COMPUTATIONAL GEOMETRY IN A CURVED WORLD

Diane L. Souvaine

(thesis)

CS-TR-094-87

October 1986

COMPUTATIONAL GEOMETRY IN A CURVED WORLD

Diane L. Souvaine

A DISSERTATION
PRESENTED TO THE
FACULTY OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE BY THE
DEPARTMENT OF
COMPUTER SCIENCE

OCTOBER 1986

To Richard

Acknowledgments

David Dobkin supervised this dissertation. He proposed the topic and has been a steady resource throughout its development. Over four years at Princeton, he as served as advisor, instructor, mentor, collaborator, and friend. I am enormously grateful for his expertise, wisdom, insight, and sensitivity, and his willingness to share all four.

Bruce Arden, Andrea LaPaugh, Richard Lipton, and Kenneth Steiglitz all taught me essential courses that formed my foundation in computer science. They continued to support and encourage me during subsequent years, making the department a congenial place in which to work.

Kennneth Supowit and Robert Tarjan were the readers for this dissertation. Each made numerous suggestions that have enhanced the quality of this work. In addition, I benefitted from timely conversations with each of them and with William Thurston during the course of my research.

Christopher Van Wyk collaborated with David Dobkin and me on the results in Sections 5.4, 5.5, and 6.2. He also read an entire draft of this dissertation in a short time, making very useful comments.

I thank Peter Honeyman, Pat Parseghian, Larry Rogers, and Marc Stavely for their help in overcoming the technical difficulties that arose in producing this document.

This research was supported in part by an Exxon Foundation Fellowship and by by National Science Foundation Grants MCS 83-03926 and DCR85-05517.

Special thanks go to Richard Horn for his constant love and support.

Abstract

Computational geometry as a field deals with the algorithmic aspects of all geometric problems. But the majority of the results obtained heretofore have been focused on objects defined with straight lines and flat faces, in part because a computational geometry of curved objects seemed significantly more complex. The major result of this dissertation is to show that curved objects can indeed be processed efficiently.

We extend the results of straight-edged computational geometry into the curved world by defining a pair of new geometric objects, the *splinegon* and the *splinehedron*, as curved generalizations of the polygon and polyhedron. We identify three distinct techniques for extending polygon algorithms to splinegons: the carrier polygon approach, the bounding polygon approach, and the direct approach. By these methods, large groups of algorithms for polygons can be extended as a class to encompass these new objects. In general, if the original polygon algorithm has time complexity $O(f(n))$, the comparable splinegon algorithm has time complexity at worst $O(Kf(n))$ where K represents a constant number of calls to a series of primitive procedures on individual curved edges. These techniques apply also to splinehedra. In addition to presenting the general methods, we state and prove a series of specific theorems. Problem areas include convex hull computation, diameter computation, intersection detection and computation, kernel computation, monotonicity testing, and monotone decomposition, among others.

Table of Contents

Chapter 1

Introduction

1.1. The Need for Algorithms on Curvilinear Objects

Computational geometry as a field focuses on the algorithmic aspects of geometric problems. Thus the span of the field should include algorithms for all objects definable in any geometry. Despite the breadth of the field as it has been defined, the majority of the results obtained heretofore have been restricted to a small class of geometric objects: points, lines, line segments, polygons, planes, and polyhedra. This narrowness of focus stems in part from the belief that a computational geometry of curved objects would be significantly more complex. The major result of this dissertation is to show that curved objects can indeed be processed efficiently.

In the early stages of computational geometry, work on problems involving two and three-dimensional objects focused primarily on convex objects defined with straight lines. The discrepancy between the results provided by the researchers in computational geometry and the problems faced by its practitioners prompted the development of decomposition algorithms. Decomposition algorithms provide efficient methods for splitting arbitrary polygons into shapes which are better understood and thus are more easily handled, such as convex, monotone, or triangular pieces. Although it would be preferable for algorithms to process diverse objects directly, the combination of decomposition algorithms with the extensive results on the restricted class of objects has produced a

sophisticated algorithmic framework for the efficient processing of arbitrary polygons.

As computational geometry matures, it is necessary to broaden the focus further, since despite this extensive body of algorithms and algorithmic techniques for objects defined on straight edges, few of its results apply directly to problems of the real world. Graphics implementors, robotics researchers, VLSI designers and pattern recognition researchers all work with objects that are neither convex nor flat. For example, solid modeling systems build objects by patching together surface patches that are defined via bicubic splines or quadratic splines [Re]. Motion planning problems that need to be solved for the advancement of robotics typically involve motion of curved objects through barriers having curved shapes [HK]. In addition, modern font design systems rely upon conic and cubic spline curves [Pa,Pr,Kn]. Both Smith and Forrest spoke fervently at the recent Computational Geometry Symposium of the need for efficient algorithms for processing curved objects directly [Sm, Fo].

Little progress has been made in this area. With the exception of a few algorithms for regular curved objects such as circles and spheres, only recently have algorithms begun to appear that treat curved objects directly [SV, HMRT]. Instead, the way to tackle arbitrary real objects has been to approximate them first as polygons or polyhedra of a sufficient number of vertices for the particular application. This process is generally quite unsatisfactory [Sm, Fo].

In this dissertation, we take a significant step towards remedying this

situation. Our major results are a set of recipes that can be used to determine if a result in the linear convex world can apply in the curvilinear world and if so to determine the best method of translating the result. We supplement this with numerous applications to existing algorithms. Furthermore, as new algorithms are developed for straight objects, it will often be possible to state them for curved objects with no extra machinery. These contributions will aid both producers and consumers of geometric algorithms.

1.2. Organization of the Dissertation

In the dissertation, we develop a computational geometry for processing curvilinear objects cleanly and efficiently. In Chapter 2, we begin by defining a new geometric object, which we call a *splinegon*. In defining the splinegon, we aim to retain the accuracy of a continuous description of an object and to encompass as many different shapes as possible. At the same time, the splinegon must capture enough discrete properties so that the techniques used in polygon algorithms can be applied to splinegons as well. In addition, the structure of the splinegon must allow analysis of algorithm performance to be clean.

Our definition allows almost every closed curve to be formulated as a splinegon, and the splinegon is amenable to a variety of computational techniques. Throughout the dissertation, we demonstrate that large groups of algorithms for polygons can be extended as a class to encompass these new objects. In general, if the original polygon algorithm has time complexity $O(f(n))$, the comparable splinegon algorithm has time complexity at worst $O(Kf(n))$ where K represents a constant number of calls to a series of primitive procedures on

individual curved edges.

We identify three general methods for translating groups of algorithms for linear objects to algorithms for splinegons. In Chapter 3, we present the carrier polygon approach. The carrier polygon is a polygon all of whose edges are chords of the splinegon. The carrier polygon often provides enough information about the behavior of the splinegon that the number of direct computations on the curved edges themselves can be reduced. Algorithms can focus on the carrier polygon, computing specific information about the behavior of the splinegon only when necessary. Algorithms extended by the carrier polygon approach include line-convex polygon intersection computation, convex polygon-polygon intersection detection, point inclusion testing, and area computation.

In Chapter 4, we present the bounding polygon approach. The bounding polygon is a polygon each of whose edges is tangent to an edge of the splinegon. The bounding polygon approximates the contour of a splinegon better than does the carrier polygon. Algorithms extended by the bounding polygon approach include convex polygon diameter computation, monotonicity testing, kernel computation, and computation of the convex hull of a simple polygon.

Chapter 5 covers the direct approach. Often edge-based algorithms can be translated to splinegons merely by updating the assumptions about possible behavior of edges. Algorithms extended by this approach, either by the author or by others, include convex polygon-polygon intersection computation, horizontal visibility computation, simplicity testing, monotone decomposition, and computation of the convex hull of a simple polygon.

Chapter 6 discusses the applicability of convex decomposition and triangulation to splinegons. Splinegons can be decomposed into convex pieces, provided that both the union and the difference operation are allowed. Splinegons can be triangulated, but the triangles are not necessarily convex. Both triangulation and convex decomposition may require a linear number of new vertices. Thus the number of triangles or convex pieces may prevent the efficient extension of polygon algorithms dependent on these decompositions. The extent to which monotone decomposition or other stratagems may compensate in algorithm development for awkward convex decomposition and triangulation remains unclear.

Three-dimensional curved geometric objects, named *splinehedra*, are defined and described in Chapter 7. It is interesting to note that while the splinegon emerged as the natural generalization of the polygon, there are two natural generalizations of the polyhedron which could be chosen as the definition of a splinehedron. We describe both and discuss the advantages and limitations of each.

Chapter 8 reviews the contributions of this work and describes the many directions in which extensions are possible.

Chapter 2

The Splinegon and its Properties

2.1. Introduction

The extension of algorithms designed for the world of straight-edged objects into the world of curved objects requires the definition of a new abstract object which can mediate between these two worlds. We call this new object a *splinegon*. First, we give a formal definition of the object as a curved extension of a straight-edged polygon. The formal definition, however, is of limited usefulness. In practice, we do not need to create new curved objects from polygons. Instead, we are given a curved object and need to process it. Thus, in the following section, we describe the process of structuring an arbitrary curved object as a splinegon, choosing vertices which relate the splinegon to an inferred polygon. We also isolate the few curved objects which cannot be formulated as splinegons.

As we show in subsequent chapters, algorithms for polygons can be extended to algorithms for splinegons with the same asymptotic time complexity except for the increased complexity of the primitive procedures. Primitive procedures on splinegons are more complicated than primitive procedures on polygons. Determining the intersection of two line segments, for example, is a well understood, and often implemented, process requiring constant time.[1] The

[1] As Forrest [Fo] and others remind us, however, in a world where round-off error exists, even line segment intersection is not a solved problem. We ignore such questions here.

complexity of determining the intersection of two curved segments depends on the complexity of the curves themselves. In the final section of this chapter, we describe our method of accounting for these primitive procedures.

2.2. Formal Definitions

A *splinegon* S can be formed from a polygon P on n vertices, $v_1, v_2, \ldots, v_n$, by replacing each line segment $\overline{v_i v_{i+1}}$ with a curved edge e_i which also joins v_i and v_{i+1} and which satisfies the following condition: the region $S\text{-}seg_i$ bounded by the curve e_i and the line segment $\overline{v_i v_{i+1}}$ must be convex.[2] The new edge need not be smooth; a sufficient condition is that the there exists a left-hand and a right-hand derivative at each point p on the splinegon. If $S\text{-}seg_i \subseteq S$, then we say that the edge e_i is *concave-in*. Otherwise, we say that the edge e_i is *concave-out*. The polygon P is called the *carrier polygon* of the *splinegon* S.

Splinegons can be categorized much as polygons are. If the only edge intersections are those between two adjacent edges at their common vertex, then the splinegon is said to be *simple*. If other edge intersections exist, then the splinegon is called *non-simple*. A splinegon may be classified as a *monotone splinegon* in some distinguished direction $\overrightarrow{z}$ if it satisfies the following criterion: let m (resp. M) represent the point on the splinegon having the smallest (resp. largest) component in the $\overrightarrow{z}$ direction; the points m and M split the splinegon into two monotone chains such that in traversing either chain from m to M the $\overrightarrow{z}$ component strictly increases. A *starshaped splinegon* contains at least one point w

[2] Subscripts are always interpreted modulo n.

in its interior so that each line segment from w to a point on the boundary of the splinegon lies within the splinegon. The collection of all such points w is called the *kernel* of the splinegon. The carrier polygons for splinegons in these four categories may or may not be simple (see Fig. 1).

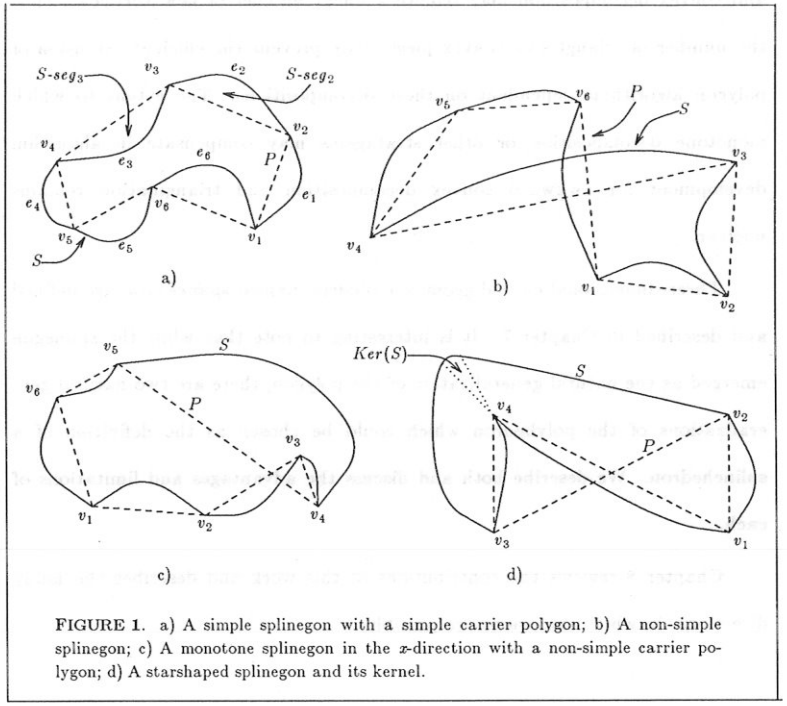

FIGURE 1. a) A simple splinegon with a simple carrier polygon; b) A non-simple splinegon; c) A monotone splinegon in the x-direction with a non-simple carrier polygon; d) A starshaped splinegon and its kernel.

A *convex splinegon* S has a convex carrier polygon P and encloses a convex region. We define a *triangle* to be a simple splinegon of three vertices. Since we have made no restriction that edges of splinegons be smooth, any arbitrary convex polygon of n vertices may be considered a splinegonal triangle. Although in

the polygonal world a triangle is necessarily convex, a splinegonal triangle has no such restriction (see Fig. 2).

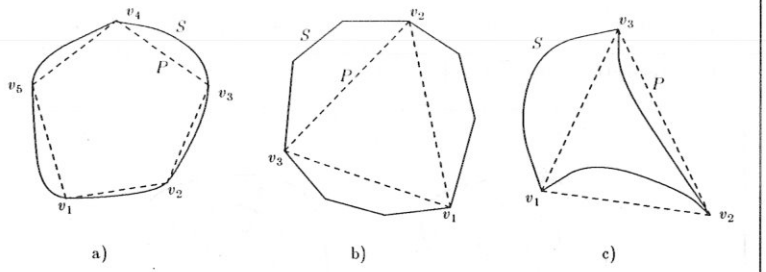

FIGURE 2. a) A convex splinegon; b) A convex triangle; c) A non-convex triangle.

2.3. Restructuring Closed Planar Curves as Splinegons

We have defined the process of creating a splinegon from an arbitrary polygon. Before describing the reverse process, we must decide which closed curves can be considered splinegons and which cannot. The following three segments define a closed curve which bounds a simple region, but the curve cannot be considered a splinegon (see Fig. 3a):

$$y = x \sin(1/x) \text{ for } x \in (0,\ 2/\pi];$$
$$y = 2/\pi \text{ for } x \in (0,\ 2/\pi];$$
$$x = 0 \text{ for } y \in [0,\ 2/\pi].$$

The first segment has an infinite number of inflection points, all lying on the x-axis. Each of them would have to be added as a vertex of the carrier polygon in order to satisfy the convexity criterion. To be a splinegon a curve must have

a finite number of inflection points: those points which separate a concave-in section of curve from a concave-out section and those points at which two concave-in (resp. concave-out) sections form an interior (resp. exterior) reflex angle.

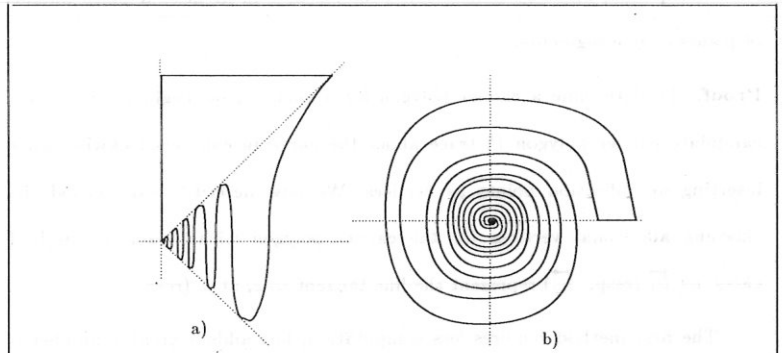

FIGURE 3. Closed curves which are not splinegons.

The following four segments also define a closed curve which bounds a simple region, but again the curve cannot be considered a splinegon (see Fig. 3b):

$$r = 1\ /\ \theta \text{ for } \theta \in [2\pi, +\infty);$$
$$r = 1\ /\ (\theta+\pi) \text{ for } \theta \in [2\pi, +\infty);$$
$$\theta = 0 \text{ for } r \in [1/3\pi,\ 1/2\pi];$$
$$(r, \theta) = (0,0).$$

Both the first and second segments wind around each other an infinite number of times. Despite the fact that the curve has only two inflection points, the infinite winding prevents any finite collection of vertices from decomposing the curve into a collection of curved segments which, together with the line segments joining their endpoints, bound convex regions.

We can now categorize the set of planar curves definable as splinegons:

Lemma. Any closed planar curve S can be considered a splinegon provided that the following two conditions are satisfied: S has only a finite number of inflection points; and any infinite line l intersects S in at most a finite number of points or line segments.

Proof. To determine a carrier polygon for a curve S, we begin by creating a candidate carrier polygon P: trace about the curve in counter-clockwise order, inserting all inflection points as vertices. We now describe two methods for choosing additional vertices for the carrier polygon within each e_i. In both cases, let $\overleftrightarrow{l_{i1}}$ (resp. $\overleftrightarrow{l_{i2}}$) represent the line tangent to e_i at v_i (resp. v_{i+1}).

The first method requires less computation, but adds a greater number of vertices. Trace S in counter-clockwise order, moving from vertex to vertex. At every edge e_i, add as a vertex of P each single point and the endpoints of any line segments of the intersection of the line $\overleftrightarrow{l_{i1}}$ with e_i. If w_i is the last such vertex added, then add each single point and the endpoints of any line segments of the intersection of the line $\overleftrightarrow{l_{i2}}$ with the portion of e_i between w_i and v_{i+1}. Add all new vertices to P in the order in which they are encountered on tracing e_i from v_i to v_{i+1}. This method guarantees that each *S-seg* is indeed convex, but it adds more vertices than necessary and yields an unwieldy carrier polygon, with many collinear edges (see Fig. 4a).

The second method requires more computation, but adds fewer vertices and yields a more manageable carrier polygon. Trace S in counter-clockwise order, moving from vertex to vertex. Stop at each edge e_i. If the line $\overleftrightarrow{l_{i1}}$ intersects e_i

in a single point (rather than a line segment) other than v_i, insert the first such point encountered on tracing e_i from v_i to v_{i+1} into the vertex list for P, making it the next vertex to be visited and splitting the current edge into two pieces. Repeat this process, tracing the curve in clockwise order and testing each edge e_i for intersection with $\overleftrightarrow{l_{i2}}$ (see Fig. 4b). □

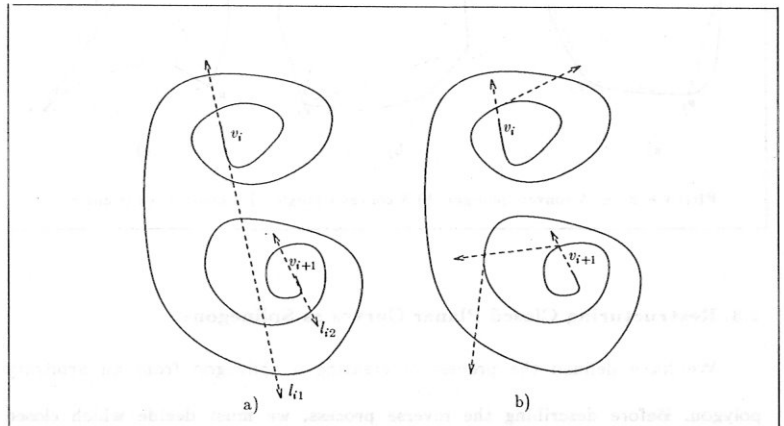

FIGURE 4. An edge e_i of S when its carrier polygon P contains only inflection points. a) The vertices added by method 1. b) the vertices added by method 2.

2.4. Primitive Operations on Splinegons

For any closed planar curve S which has a finite number of inflection points and which intersects each infinite line in a finite number of components, a carrier polygon can be found which transforms S into a splinegon. This splinegon has far fewer vertices, in general, than any polygon which would adequately approximate the curve. This is an advantage in object description, especially in

situations where some calculations can be done based on the structure of the carrier polygon.

Calculations on splinegons, however, remain more complicated than those on polygons. When designing geometric algorithms for linear objects, we assume the existence of certain primitive procedures, such as determining the intersection of two line segments or finding a line which supports a pair of adjacent edges at their shared vertex. These procedures can be performed in constant time. As the splinegon model places no restrictions on the complexity of the description of curved edges, manipulation of curved edges could become arbitrarily complicated. One route to take (e.g. [SV]) is to assume that in each instance there exists a bound on the complexity of each curved edge. Then each primitive operation can be considered to require constant time. For example, if each edge is described by a polynomial of bounded degree, then a primitive operation will require time dependent on that degree. Note, however, that the intersection of two polynomial curves of degree 5, for example, is not solvable in closed form. Thus the constant associated with this primitive procedure must represent the time required by the chosen approximation algorithm, accurate to some requisite number of bits.

Rather than ignore the time spent on individual primitive operations, however, we prefer to emulate the technique used by computational geometers studying collections of N points in d-dimensional space. Although d could be viewed as a constant and ignored, analyses of algorithm complexity are reported as functions of both N and d. The complexity of primitive operations on splinegon edges, however, cannot be easily parametrized by a single variable, as edges

need not be polynomials but may be defined in countless different ways. Our solution is to postulate the existence of a family of procedures, or oracles, which perform the primitive operations needed in the design of our algorithms. To each of these procedures, or oracles, we assign a variable respresenting the time required to provide the requested response:

A_1 (A_2) compute the intersection of two curved edges (faces) or the maximum and minimum separation between them.

B_1 (B_2) compute the intersection of a line with a curved edge (face).

C_1 (C_2) given a curved edge (face) and either a direction or a point, report both the point and the direction of a line (plane) which supports the edge (face) at that point.

D_1 (D_2) determine the line (plane) which supports a pair of curved edges (faces).

E_1 (E_2) compute the intersection of a plane with a curved edge (face).

F_1 (F_2) compute the area (volume) bounded by a curved edge (face) and by the corresponding edge (face) of the carrier polygon (polyhedron).

This set of oracles is sufficient to generate the results which follow, but also includes a redundancy which facilitates the analysis of the curvilinear algorithms, allowing the isolation of those "non-linear" computations involved. Note that the oracles A_1 and D_1 (resp. A_2 and D_2) have the greatest complexity as they operate on a pair of curved edges (resp. faces). Our analyses of algorithm complexity are given both in terms of operations and calls to oracles

for primitive procedures. We reiterate, however, that the complexity of each primitive procedure would be a constant in any domain where we restricted splinegon edges (splinehedron faces).

When choosing a splinegonal representation of a closed planar curve, the implementor will note a tradeoff. An N-sided convex polygon can be considered a triangle, but the primitive operation of determining whether an "edge" of one intersects an "edge" of another will be slow (see Fig. 4b). On the other hand, each algorithm will have a constant number of iterations. An algorithm for an N-sided polygon can run very slowly when N is very large, but the primitive operations will take constant time. This type of dichotomy has produced a class of efficient hierarchical algorithms [Ki, DK2], which we discuss further in Section 3.5.

Chapter 3

Algorithms Focusing on the Carrier Polygon

3.1. Carrier Polygon Approach

In the field of computational geometry, of all classes of polygons, convex polygons have received the greatest amount of study and are the best understood [LP3, PS1]. We begin with the definition of convexity and show how this and some key related properties have contributed to algorithm development. We then propose a technique which uses these properties for extending polygon algorithms to splinegon algorithms.

A set S is defined as being convex if and only if whenever both x and y belong to S then so does $\lambda x + (1 - \lambda)y$, for all λ such that $0 \leq \lambda \leq 1$. From a geometric standpoint, an object S is convex if and only if the line segment $\overline{xy}$ joining any two points of S, x and y, lies completely within S. From this definition, two key properties are derived.

First is the separation property. That is, given any point x on the boundary of a planar convex object S, there exists a line l through x which divides the plane into two half-planes: a closed half-plane containing all of S; and an open half-plane containing no point of S. The intersection of l and S consists either of the single point x or else of a single closed line segment containing x. Such a line l is called a *supporting line* for S at x (see Fig. 5). If S is a polygon of n vertices, then the set of n lines formed by extending each of the n edges of S is sufficient to supply a supporting line for S at each point on its boundary.

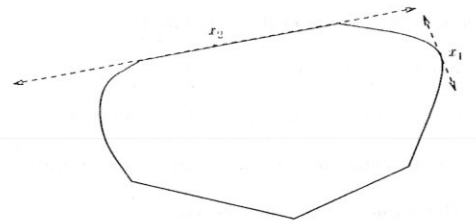

FIGURE 5. A planar convex object with supporting lines at given points. The line at x_1 intersects in a point; the line at x_2 in a segment.

The second property is the supporting line property. Given any direction in the plane, there exists a pair of lines m_1 and m_2 in that direction such that both m_1 and m_2 are supporting lines for S and the region bounded by m_1 and m_2 completely contains S (see Fig. 6). If S is a polygon of n vertices, then the intersection of both m_1 and m_2 with S must include at least one of the n vertices.

For a convex splinegon S with at least one curved edge, no finite set of lines can include a line of support for S at each point on its boundary. Moreover, no finite set of points on the boundary of S includes a point lying on each line of support for S. The necessary points and lines will have to be determined on an individual basis by calling an oracle. Note that this differs from the situation with convex polygons P where the lines of support of the edges cover the boundary of P, and every line of support of P passes through a vertex.

One of our goals, however, is to minimize the number of oracle calls. As the curved edges on splinegons become more complex, the time associated with

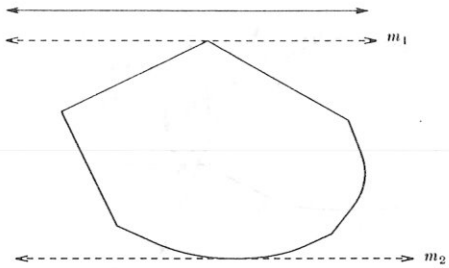

FIGURE 6. A planar convex object enclosed by a pair of supporting lines of given direction.

each oracle call increases. Wherever possible, we would like to develop algorithms which require asymptotically fewer oracle calls than they do simple operations. In other words, we prefer that the running time of an algorithm depend more on the number of edges than on the complexity of those edges.

When processing convex splinegons, we avoid many of the direct manipulations of the curved edges by focusing on the carrier polygon instead. The carrier polygon for a convex splinegon is convex. The line defined by two adjacent vertices, $\overleftrightarrow{v_i v_{i+1}}$, divides the plane into two half-planes: the "outside" half-plane contains the convex region $S\text{-}seg_i$; the "inside" half-plane contains a splinegon $S_i = S - S\text{-}seg_i$. S_i can be considered a convex polygon which is supported by $\overleftrightarrow{v_i v_{i+1}}$ along an edge (see Fig. 7).

Furthermore, we can show that the convexity of S dictates that $S\text{-}seg_i$ be enclosed by the triangle $S\text{-}tri_i$ determined by the three lines $\overleftrightarrow{v_i v_{i+1}}$, $\overleftrightarrow{v_{i-1} v_i}$, and $\overleftrightarrow{v_{i+1} v_{i+2}}$. Assume otherwise. Then a point x of $S\text{-}seg_i$ lies outside both $\overleftrightarrow{v_i v_{i+1}}$,

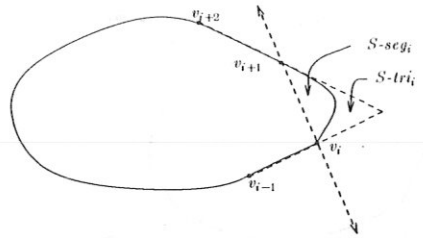

FIGURE 7. A convex splinegon S divided into S_i and $S\text{-}seg_i$.

and $\overleftrightarrow{v_{i-1}v_i}$. But the line segments $\overline{v_{i-1}x}$, $\overline{v_{i+1}x}$, and $\overline{v_{i-1}v_{i+1}}$ must all belong to S, and so must the triangle these segments define. This triangle, however, contains v_i in its interior, contradicting the fact that v_i lies on the boundary of S (see Fig. 8).

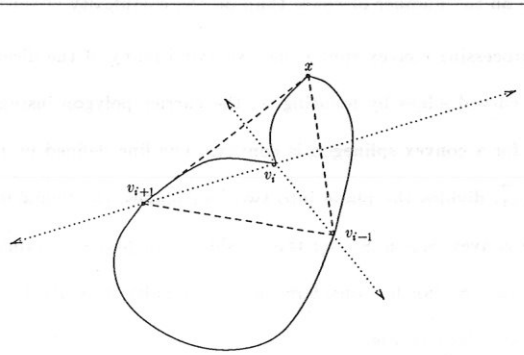

FIGURE 8. A demonstration that all points of $S\text{-}seg_i$ must lie inside both $\overleftrightarrow{v_{i-1}v_i}$ and $\overleftrightarrow{v_{i+1}v_{i+2}}$.

Without any direct manipulation of curved edges, the behavior of S can be reasonably approximated. The techniques used in a polygon algorithm can be

applied to the carrier polygon of a splinegon, with slight modifications to allow for all possible behavior of $S\text{-}seg_i$ and its bounding triangle $S\text{-}tri_i$. Only infrequently will an examination of the precise behavior of e_i be necessary. Then we can compute the line tangent to e_i and parallel to some other given line. This tangent line separates the plane into two half-planes, one containing the splinegon, and one not. The splinegon can then be viewed as a convex polygon supported by the line at a vertex. The corresponding step in the polygon algorithm can then be applied directly to the splinegon.

3.2. Computing the Intersection of a Line with a Convex Splinegon

As a first example of the utility of the carrier polygon approach, we discuss the process of computing the intersection of a line with a convex splinegon. Assuming that the vertices of the polygon are already stored in random access memory, Chazelle and Dobkin [CD] have shown that the intersection of a line and a convex polygon of N vertices can be computed in $O(\log n)$ time. In the case of no intersection, their algorithm reports the vertex of the polygon which lies closest to the line. Using the carrier polygon approach, we extend their result, also assuming that the vertices of the splinegon are stored in random access memory, with each vertex pointing to a description of the curve which joins it to its neighbor:

Theorem 1. The intersection of a line with a convex splinegon of N vertices can be computed in $O(B_1 + \log N)$ operations (see Fig. 9).

Proof. To compute the intersection of a line l with a convex splinegon S, first

run the Chazelle-Dobkin polygon algorithm [CD] on the carrier polygon P. If there is no intersection, that algorithm will report the vertex v_i of the polygon which is closest to l. Although l does not intersect P, it may still intersect one of $S\text{-}seg_{i-1}$ or $S\text{-}seg_i$. (The convexity of S dictates that l cannot intersect both.) Consequently, we use oracle B_1 to test l against e_{i-1} and e_i. In the case that the intersection of l with P consists of a single vertex v_i, again l may also intersect either $S\text{-}seg_{i-1}$ or $S\text{-}seg_i$. Testing the line against both e_{i-1} and e_i is sufficient to determine the splinegon-line intersection. If the polygon algorithm reports the intersection as $\overline{v_iv_{i+1}}$, an entire edge of P, or as $\overline{v_iv_j}$, a diagonal of P, then the intersection of l and S also consists exactly of that segment. If the polygon algorithm reports the intersection as a line segment with endpoints on two different edges of the carrier polygon, $\overline{v_iv_{i+1}}$ and $\overline{v_jv_{j+1}}$, testing the line against the corresponding curved edges e_i and e_j determines the endpoints of the segment forming the splinegon-line intersection. After running the polygon algorithm in $O(\log N)$ time, the subsequent special cases each require at most two calls to oracle B_1. Thus the entire process requires $O(B_1 + \log N)$ time, using asymptotically fewer oracle calls than simple operations. □

Corollary. The inclusion of a point within a convex splinegon of N vertices can be decided in $O(B_1 + \log N)$ operations.

Proof. To decide whether a point x lies within the splinegon P, choose an arbitrary line L which passes through x and determine the intersection of L and P. Constant time then suffices to check whether x lies within that intersection. □

In the examples given here, the carrier polygon provides sufficient constraints on the behavior of the splinegon that running the original algorithm unchanged on the carrier polygon precisely locates the two S—*segs* which decide the solution. A single primitive procedure on each produces the answer.

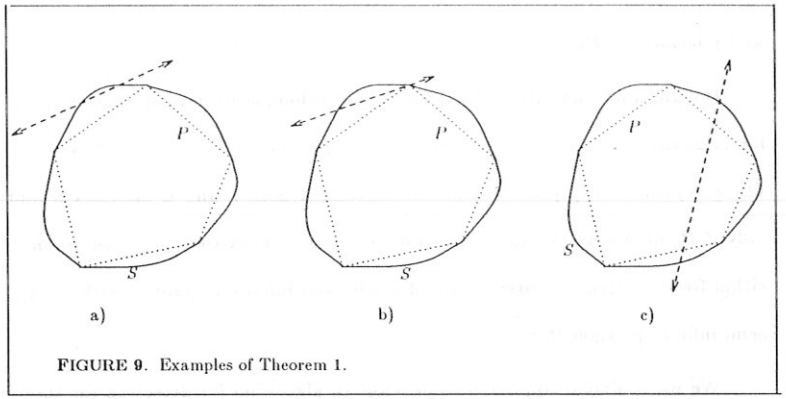

FIGURE 9. Examples of Theorem 1.

3.3. Detecting the Intersection of two Convex Splinegons

By definition, a polygon P of N vertices which is monotone in the y-direction can be split at the points having maximum and minimum y-coordinate into a left and a right monotone chain, P_L and P_R. Assuming that the vertices of the polygon are already stored in random access memory, this splitting process requires $O(\log N)$ ordinary operations [CD]. A point p having y-coordinate y_p belongs to P if and only if it lies to the right of the unique edge of P_L which intersects the line $y = y_p$ and to the left of the unique edge of P_R which intersects the same line. Alternatively, we can let the monotone chains P_L and P_R represent semi-infinite polygons which open infinitely to the left and right, respectively. Then we can say that p belongs to P if and only if p belongs to P_L

and p belongs to P_R.

Dobkin and Kirkpatrick [DK] used this technique to develop an algorithm for detecting the intersection of two convex polygons of at most N vertices in $O(\log N)$ time. They proved that two convex polygons P and Q intersect if and only if P_L intersects Q_R and P_R also intersects Q_L. They then provided an algorithm for detecting the intersection of a left semi-infinite polygon L with a right semi-infinite polygon R.

We use a similar approach to develop an algorithm for detecting the intersection of two convex splinegons. To rewrite a convex splinegon S of at most N vertices as the intersection of two semi-infinite open splinegons S_L and S_R, we first determine the pair of vertices of maximum and minimum y-coordinate, v_M and v_m, using $O(\log N)$ time. Unlike the polygon case, these vertices rarely represent the extreme points of the splinegon. Let v_M^* represent the point of maximum y-coordinate lying on either e_M or e_{M-1}. Likewise, let v_m^* represent the point of minimum y-coordinate lying on either e_m or e_{m-1}. The two points v_M^* and v_m^* are inserted as vertices into the ordered list for S, and then they become the splitting points to form S_L and S_R (see Fig. 10). The entire process takes $O(C_1 + \log N)$ time.

Given the problem of detecting the intersection of two convex splinegons P and Q of at most N vertices each, we expect that each splinegon is given as a list of vertices in a random access memory with each vertex pointing to a description of the curve which adjoins it to its neighbor. We do no preprocessing of this description. The output should consist either of a point in the intersection or of a line which supports one of the splinegons and separates it from the other.

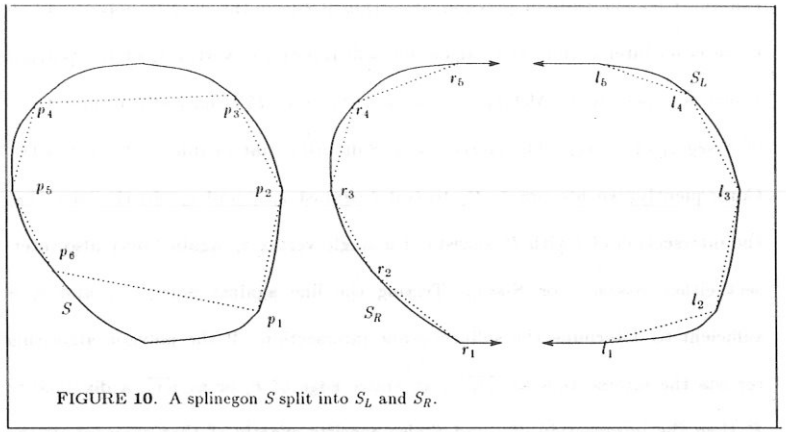

FIGURE 10. A splinegon S split into S_L and S_R.

Once P and Q have both been rewritten as the intersection of two semi-infinite splinegons, $P_L \cap P_R$ and $Q_L \cap Q_R$ respectively, we need an algorithm which detects the intersection of a left semi-infinite splinegon with a right semi-infinite splinegon. Before giving details, we introduce the necessary notation. L (resp. R) is a semi-infinite splinegon opening to the left (resp. right) and having the n (resp. m) vertices $l_1, l_2, \cdots, l_n$ (resp. $r_1, r_2, \cdots, r_m$). Let $i = \lceil m/2 \rceil$ and $j = \lceil n/2 \rceil$. R_i (resp. L_j) represents the line defined by the pair of vertices r_i, r_{i+1} (resp. l_j, l_{j+1}). Our results rely upon the subdivision of the plane induced by the lines L_j and R_i.

In general, four regions are produced as shown in Fig. 11. In each of these regions, there are limitations on the involvement of each splinegon. Together the R-region (resp. L-region) and the LR-region contain all of R (resp. L) except

for $R\text{-}seg_i$ (resp. $L\text{-}seg_j$). Together the $\emptyset$-region and the L-region (resp. R-region) contain all of $R\text{-}seg_i$ (resp. $L\text{-}seg_j$). If R_i and L_j are parallel, they divide the plane into only three regions. If L_j is left of R_i, no LR-region exists. If R_i is left of L_j, no $\emptyset$-region exists. In all cases, the L-region (R-region) lies the furthest to the left (right). If the LR-region and the $\emptyset$-region co-exist, then one lies above the other, as determined by the angles which R_i and L_j form with the horizontal.

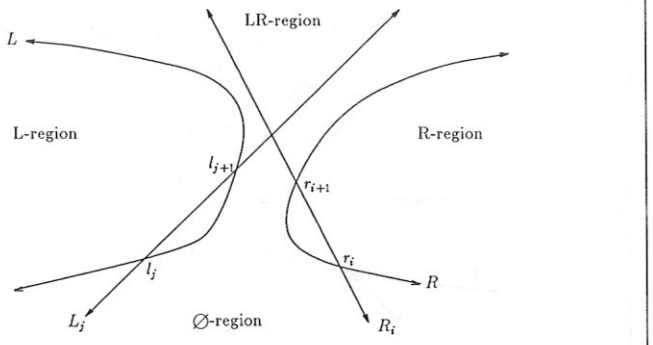

FIGURE 11. The subdivision of the plane induced by R_i and L_j.

The basis for the inner loop of our algorithm is established by the sequence of lemmas we present below. In the statement of those lemmas, we define **above**, **below** and **separation** with respect to y-coordinates: p lies **above** (resp. **below**) q if the y-coordinate of p is larger (resp. smaller) than that of q; and, r **separates** p and q if its y-coordinate lies between the other two y-coordinates. In addition, we assume that the LR-region lies above the $\emptyset$-region. Should the converse hold, symmetric lemmas can be easily stated. We

also state results in terms of R and its vertices but symmetric results can be stated for L and its vertices.

Lemma 1. If l_{j+1} and r_{i+1} both border the LR-region and if r_{i+1} separates l_j and l_{j+1}, then r_{i+1} is a witness to the intersection of L and R (see Fig. 12).

Proof. The point r_{i+1} lies to the left of the curved edge from l_j to l_{j+1} on the left semi-infinite splinegon L. Thus $r_{i+1} \in L$. □

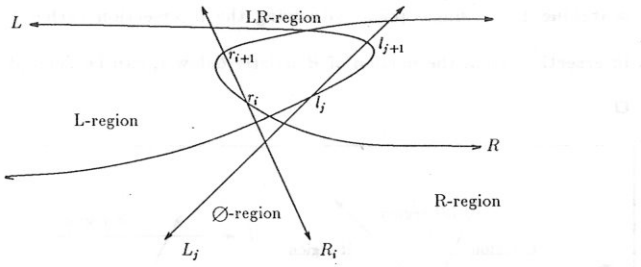

FIGURE 12. Example of Lemma 1.

Lemma 2. If r_i lies below r_{i+1}, l_j, and l_{j+1}, then L and R intersect if and only if L intersects the splinegon formed by deleting the vertices of R lying below r_i (see Fig. 13).

Proof. Suppose that r_i borders the $\emptyset$-region. Then the R-region completely contains the portion of R lying below r_i. The only portion of L which can enter the R-region is $L\text{-}seg_j$. But $L\text{-}seg_j$ lies strictly between l_j and l_{j+1} and thus lies strictly above r_i. Consequently, the portion of R lying below r_i cannot play any part in the intersection of L and R (see Fig. 13a).

Suppose that r_i borders the LR-region, and furthermore that L intersects R at a point z which lies below r_i. As no part of L below l_j can enter the R-region, z must lie in the LR-region. By convexity, both the LR-region and L contain the line segment $\overline{l_j z}$. But as l_j lies above r_i and z lies below r_i, $\overline{l_j z}$ must intersect the ray originating at r_i and extending infinitely to the right, a ray which is contained within R. We conclude that the portion of R lying strictly below r_i cannot completely contain the intersection of L and R. As we are searching for a single point witnessing the intersection rather than the entire intersection area, the portion of R strictly below r_i can be deleted (see Fig. 13b). □

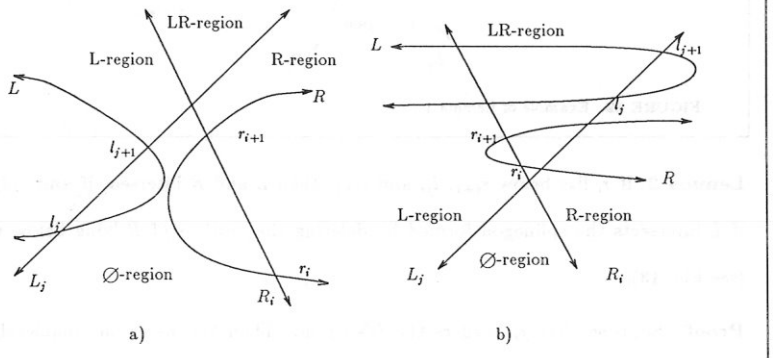

FIGURE 13. Examples of Lemma 2.

Lemma 3. If r_{i+1} and l_{j+1} both border the LR-region, and r_{i+1} lies above r_i, l_j, and l_{j+1}, then L and R intersect if and only if L intersects the splinegon formed by deleting the vertices of R lying above r_{i+1} (see Fig. 14).

Proof. Suppose that L intersects R at a point z of R lying above r_{i+1}. As no part of L above l_{j+1} can enter the R-region, z must lie in the LR-region. By convexity, both the LR-region and L contain the line segment $\overline{l_{j+1} z}$. But as l_{j+1} lies below r_{i+1} and z lies above r_{i+1}, $\overline{l_{j+1} z}$ must intersect the ray originating at r_{i+1} and extending infinitely to the right, a ray contained within R. The portion of R lying strictly above r_{i+1} cannot completely contain the intersection of L and R, so it can be deleted. □

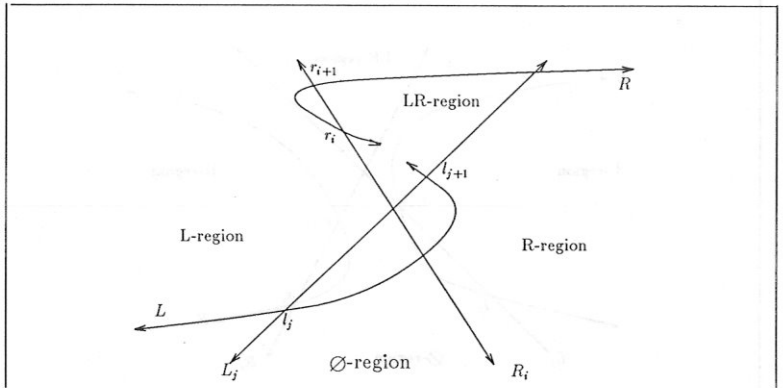

FIGURE 14. Example of Lemma 3.

Lemma 4. If l_{j+1} borders the LR-region, and r_i, r_{i+1}, and l_j all border the ∅-region, then L and R intersect if and only if L intersects the splinegon formed by deleting the vertices of R lying below r_i (see Fig. 15).

Proof. Suppose that L intersects R below r_i at a point z. Since the portion of R below r_i belongs strictly to the R-region, z also belongs to the R region. By convexity, both the R-region and L contain the line segment $\overline{l_{j+1} z}$. But as l_{j+1}

lies above r_i and z lies below r_i, $\overline{l_{j+1}z}$ must intersect the ray originating at r_i and extending infinitely to the right, a ray contained within R. The portion of R lying strictly below r_i cannot completely contain the intersection of L and R, and thus the portion of R strictly below r_i can be deleted. □

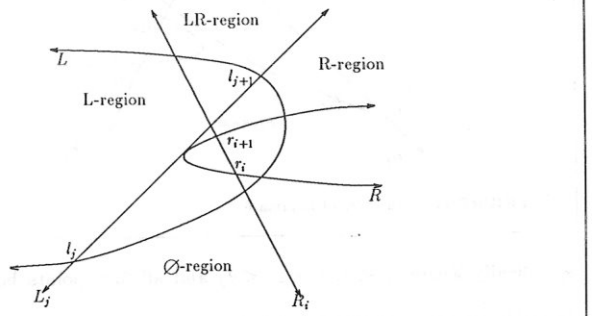

FIGURE 15. Example of Lemma 4.

Our algorithm to detect the intersection of a right semi-infinite splinegon R and a left semi-infinite splinegon L relies on these lemmas. If R and L do not satisfy the conditions for Lemma 1 or for either of its two symmetric versions, then at least one of Lemmas 2-4 or their symmetric versions will reduce the size of at least one of L or R by at least a half using only constant time. Thus, we may iteratively apply the lemmas until one of the following occurs: either a point in the intersection is determined, or one semi-infinite splinegon, say L, has at most three vertices remaining. In the former case, we are done. In the latter case, decompose L into two two-vertex left semi-infinite splinegons to test in succession against R.

The problem remaining is to detect the intersection of a two-vertex left semi-infinite splinegon L against a right semi-infinite splinegon R of m vertices. We begin by again applying Lemmas 1-4 and their symmetric versions, hoping either to locate a point in the intersection or to reduce the size of R by a half. If the only lemmas which apply, however, dictate that some portion of L be deleted, no progress has been made. Thus we need other lemmas to cover this possibility.

Lemma 5. If L has precisely two vertices, both of which border the ∅-region, and if r_{i+1} lies above l_2, then L and R intersect if and only if L intersects the splinegon formed by deleting the vertices of R lying above r_{i+1} (see Fig. 16).

Proof. Since L has only two vertices, those two vertices exactly define the y-extent of L. Consequently, any portion of R lying strictly above l_{j+1} plays no part in the intersection. □

Lemma 6. If L has precisely two vertices, both of which border the ∅-region, and if $l_1 < r_i < r_{i+1} < l_2$, then let l^i represent a point on the edge from l_1 to l_2 where a line parallel to R_i supports L. If l^i lies above r_i (resp. below r_{i+1}), then L intersects R if and only if it intersects the splinegon formed by deleting the vertices of R lying below r_i (resp. above r_{i+1}) (see Fig. 17).

Proof. Suppose that L intersects R at a point z lying below r_i (resp. above r_{i+1}). Then $z \in R$-region. But no point of L can belong to the R-region unless l^i also belongs to the R-region. Consequently, by convexity, both the R-region and L contain the line segment $\overline{l^i z}$. But as l^i lies above r_i (resp. below r_{i+1}) and z lies below r_i (resp. above r_{i+1}), $\overline{l^i z}$ must intersect the ray originating at r_i

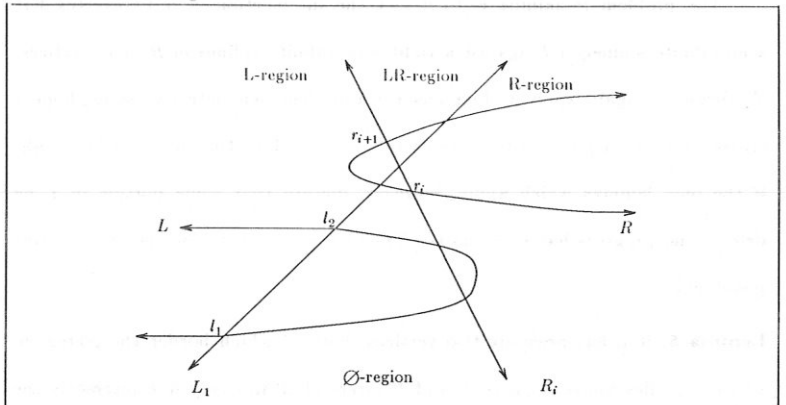

FIGURE 16. Example of Lemma 5.

(resp. r_{i+1}) and extending infinitely to the right, a ray contained within R. We conclude that the portion of R lying strictly below r_i (resp. above r_{i+1}) cannot completely contain the intersection of L and R, and thus the portion of R strictly below r_i (resp. above r_{i+1}) can be deleted. □

Theorem 2. $O(A_1 + C_1 \log N)$ operations suffice to detect the intersection of a right and a left convex semi-infinite splinegon of at most N vertices each.

Proof: These lemmas provide the basic steps for our algorithm. Whenever the hypothesis of a lemma is satisfied, either an intersection point is found or the size of one splinegon is divided in half. We repeatedly apply Lemmas 1-4 until one splinegon has at most 3 vertices. We split that splinegon into two splinegons of only two vertices each. For each two-vertex splinegon, we repeatedly apply Lemmas 1-5 until its second splinegon has been reduced to 3 vertices or until reaching an instance where not one of these five lemmas applies,

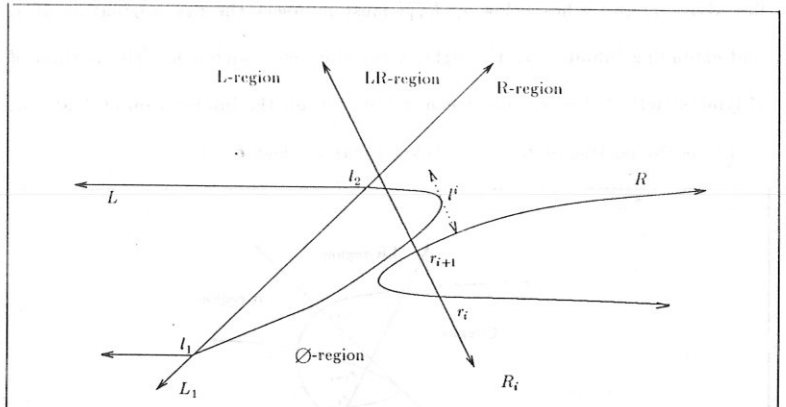

FIGURE 17. Examples of Lemma 6.

specifically where $l_1 < r_i < r_{i+1} < l_2$ and all four points border the ∅-region. In the latter case, we can apply Lemma 6.

Eventually the second splinegon is reduced to only 3 vertices. All edges remaining from one of the original splinegons can be tested for intersection against all edges remaining from the other. If an intersection point is found, it is reported. Otherwise determine the pair of points, one from each splinegon, at minimum distance from each other. Each splinegon has a supporting line at the respective point separating it from the other splinegon. $O(A_1 + C_1)$ time will detect any intersection among the remaining semi-infinite splinegons and determine a point common to both of the original splinegons or a line supporting one and separating it from the other.

In the process described above, each lemma is applied at most a logarithmic number of times. Lemmas 1-5, and their symmetric versions, each

require only constant time. Lemma 6, however, calls an oracle and thus uses time C_1. Consequently, the reducing stage of the algorithm may use time $O(C_1 \log N)$. Determining the intersection among the remaining edges requires $O(A_1)$ time. If there is no intersection, determining the supporting separating lines requires $O(C_1)$ time. □

Theorem 3. The intersection of two convex splinegons of at most N vertices can be detected in $O(A_1 + B_1 + C_1 \log N)$ operations.

Proof: Given the convex splinegons, we begin by splitting them into semi-infinite splinegons, using $O(C_1 + \log N)$ time. Then, using $O(A_1 + C_1 \log N)$ time, we apply the algorithm of the previous theorem twice: once for P_R and Q_L; and once for P_L and Q_R. If either iteration reports no intersection, then P and Q do not intersect. The supporting separating lines reported also support and separate P and Q. If both iterations report an intersection point, then P and Q do intersect. We test each point for inclusion in Q, using time $O(B_1 + \log N)$. If either point tests positive, we are done. But it is possible that neither point belongs to $P \cap Q$.

Suppose that two points of one splinegon, say P, were reported: p_1 belongs to $P_L \cap Q_R$ and p_2 belongs to $P_R \cap Q_L$. Since p_1 belongs to $P_L \cap Q_R$, p_1 lies to the right of the left boundary of Q, but since it does not belong to Q, it must also lie to the right of the right boundary of Q. Similarly, p_2 lies to the left of both boundaries of Q. Consequently, the line segment $\overline{p_2 p_1}$, which by convexity lies within P, must intersect both boundaries of Q. Thus a point in $P \cap Q$ can be found in $O(B_1 + \log N)$ time (see Fig. 18).

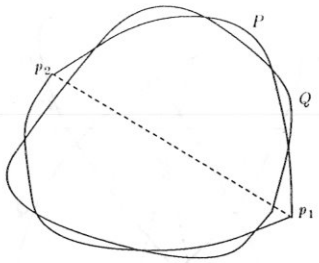

FIGURE 18. Two points of P were reported.

Suppose that one point of each splinegon were reported, say p_1 from P_L and q_1 from Q_L. Test a horizontal line through p_1 (resp. q_1) for intersection with both P and Q, using $O(B_1)$ time. If either iteration reports interleaving segments, then report a point in $P \cap Q$. If not, the segment of Q lies to the left (resp. right) of the segment of P. Consequently, a line segment joining the two segments of Q must cross any line segment joining the two segments of P, and the intersection point belongs to $P \cap Q$ (see Fig. 19). □

In the general case, this algorithm runs in time $O(A_1 + B_1 + C_1 \log N)$. Lemma 6, however, was the only one reequiring any curve manipulations. Thus a logarithmic number of curve manipulations are necessary only if there exist N^ϵ edges of one splinegon, for $\epsilon > 0$, within the y-extent of a single edge of the second splinegon.

We can make our time estimates more exact. Find the smallest index m_j and the largest index M_j such that $r_{m_j+1}, \ldots, r_{M_j+1}$ all lie within the y-extent

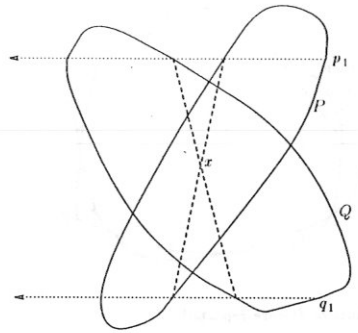

FIGURE 19. One point of P and one of Q were reported.

defined by l_j and l_{j+1}. Let s_j represent the index such that the slope of L_j lies between the slopes of R_{s_j-1} and R_{s_j}. Let h_j (resp. k_j) represent the smallest (resp. largest) index such that the line R_{h_j} (resp. R_{k_j}) has inverse slope larger than (resp. smaller than or equal to) L_j and intersects L_j below l_j (resp. above l_{j+1}) or not at all. If no index h_j (resp. k_j) satisfies these criteria, then set $h_j = s_j + 1$ (resp. $k_j = s_j - 1$). Finally, set $h = \min(h_j, M_j)$ and $k = \max(k_j, m_j)$. If in the course of the algorithm, l_j and l_{j+1} should define an extant two-vertex splinegon, then only for i such that $k_j \leq i \leq h_j$ would Lemma 6 be invoked. Define $d_L = \max_{j=1,\ldots,n} (h_j - k_j)$. Perform the symmetric calculations for R and define $d_R = \max_{i=1,\ldots,m} (j_i - k_i)$. Let $d = \max(d_L, d_R)$. Then the intersection detection algorithm runs in time $O(A_1 + B_1 + C_1 \log d + \log N)$.

At each stage in the polygon algorithm, the lines R_i and L_j divide the plane into four regions which rigidly constrain the behavior of L and R. L and

R can overlap only in the LR-region; the remainder of L lies in the L-region; the remainder of R lies in the R-region; and the $\varnothing$-region contains no portion of either semi-infinite polygon in its interior. These constraints mean that at each iteration the current edge of one semi-infinite polygon as well as all of the edges of that polygon on one side of the current edge can be deleted. Thus, a constant amount of work removes at least half of the edges of one polygon from further consideration.

The situation in the splinegon case is more murky. The two splinegons may overlap in any of the four regions defined by R_i and L_j. An intersection occurring in the R-region (resp. L-region), however, must involve $L\text{-}seg_j$ (resp. $R\text{-}seg_i$). And an intersection occuring in the $\varnothing$-region involves both $L\text{-}seg_j$ and $R\text{-}seg_i$. Thus the current edge can never be deleted.

As described in Section 3.1, however, in a convex splinegon S, each $S\text{-}seg_i$ is constrained to lie within a triangle $S\text{-}tri_i$ defined by edges of the carrier polygon P. In the first stage of the algorithm, all edges to one side of the current edge of one semi-infinite splinegon, say L, can often be determined to lie in the L-region but out of the range of $R\text{-}seg_i$. Thus these edges may all be deleted. Otherwise, we can determine that for one splinegon, say R, to intersect any portion of L to one side of the current edge, R would first have to intersect the remaining portion. One of these two tests, requiring constant time each, allows at least one less than half of the edges of one splinegon to be removed from further consideration.

At the second stage of the algorithm, when one splinegon, say L, has only

one edge, sometimes neither method applies. It is possible that all of the remaining edges of R lie within $L\text{-}tri_1$. To reduce R at all, it may be necessary to call a primitive procedure to further define the behavior of $L\text{-}seg_1$. Having done so, however, the size of R can again be reduced by at least one less than half of its edges.

The algorithm provided in this section merely detects the intersection of two convex splinegons. In Section 5.2, we present an algorithm for computing that intersection.

3.4. A Hierarchical Method for Testing Point Inclusion

In recent years, the hierarchical searching method has emerged as a potent tool in the field of computational geometry [Ki, DK2, DS]. Here, a geometric problem is preprocessed to provide a coarse representation of the entire problem. Search queries upon the whole are then used to localize the region in which the problem is to be solved. Queries of this type alternate with computations which yield continually finer descriptions of these continually smaller regions. Efficient algorithms result from balancing the two processes of localizing the search and of increasing the detail. Algorithmic efficiency is then balanced against preprocessing time and storage space requirements.

The convex splinegon is an abstract object ideally suited for hierarchical processing. As mentioned in Chapter 2, for all N, an N-sided convex polygon can be considered a splinegonal triangle, but the primitive operations on the "edges" will be slow. On the other hand, each algorithm will have a constant

number of iterations. An algorithm for an N-sided polygon can run very slowly when N is very large, but primitive operations on edges require only constant time. We now exploit this dichotomy to develop an efficient hierarchical algorithm for point inclusion.

Given a convex polygon P on N vertices, $v_1, v_2, \ldots, v_N$, we develop a hierarchy of splinegons all having the same boundary as P, but having carrier polygons of fewer vertices. We assume for ease of explanation that $N = 3(2^k)$ for some k. We recast P as a triangular splinegon S^k with a carrier polygon P^k whose three vertices $v_1{}^k, v_2{}^k, v_3{}^k$ are all original vertices of P. The vertices $v_i{}^k$ are chosen so that $N/3 - 1$ of P's original vertices lie in the interior of each of the three "curved" edges of S^k.

To create a splinegon S^{i-1} from a splinegon S^i in the hierarchy, insert each vertex $v_j{}^i$ of S^i as a vertex of S^{i-1}. In addition, insert the median original vertex $m_j{}^i$ lying in the interior of each edge $e_j{}^i$ of S^i. After performing this process k times, we achieve a splinegon S^0 which is identical to its carrier polygon P^0 and to the original polygon P (see Fig. 20).

To test whether a point x lies within the convex polygon P, we begin by determining in constant time whether x lies within the triangle P^k. If it does, we are done. If not, then x must lie outside of at least one of the lines $\overleftrightarrow{v_j{}^k v_{j+1}{}^k}$. If x lies outside of two of the lines, then by convexity it cannot belong to P, as we demonstrated in section 3.1. Suppose that x lies outside of the line $\overleftrightarrow{v_j{}^k v_{j+1}{}^k}$. Then we can determine whether x lies in the carrier polygon P^{k-1} by testing whether x belongs to the triangle $\Delta v_j{}^k m_j{}^k v_{j+1}{}^k$, which is identical to the trian-

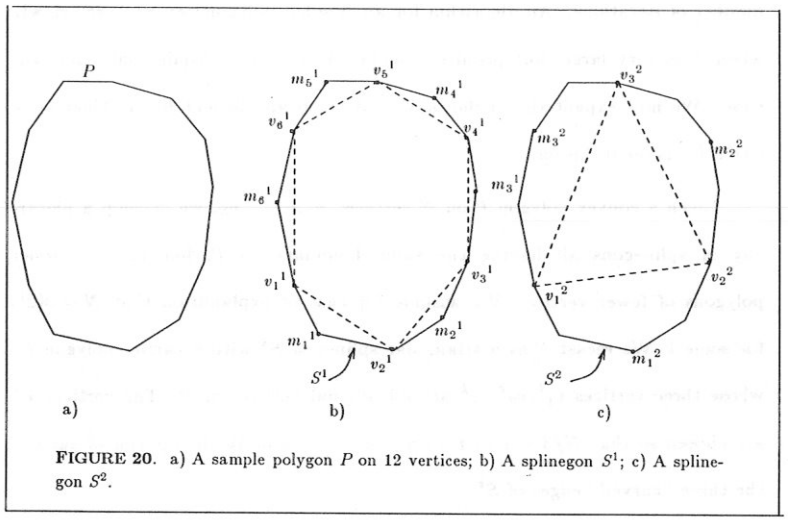

FIGURE 20. a) A sample polygon P on 12 vertices; b) A splinegon S^1; c) A splinegon S^2.

gle $\Delta v_{2j}{}^{k-1} v_{2j+1}{}^{k-1} v_{2j+2}{}^{k-1}$ defined by three adjacent vertices of P^{k-1}.

At each stage in the algorithm, we either determine that x belongs to some carrier polygon P^i, or that x lies outside of P, or select a triangular test which will determine whether x belongs to P^{i-1}. If we proceed through k stages without terminating, we know an index j such that x belongs to the original polygon P if and only if x belongs to the triangle $\Delta v_j v_{j+1} v_{j+2}$. We conclude:

Theorem 4. The inclusion of a point in a convex polygon of N vertices can be decided in $O(\log N)$ operations using the hierarchical method.

Proof. At each stage in the algorithm, the point is tested for inclusion in a triangle. The first such test reduces the size of the problem by at least $2/3$. Each subsequent test divides the problem in half. □

This approach works equally well for testing a point for inclusion in a convex splinegon. Suppose that a convex splinegon S has P as its carrier polygon. We define a hierarchy of splinegons all having the same boundary as S, but with vertices chosen as described above. In this case, the lowest-level splinegon S^0 is identically equal to S, and its carrier polygon P^0 is exactly P. At the final stage of the algorithm, we know an index j such that x belongs to the original carrier polygon P if and only if x belongs to the triangle $\Delta v_j v_{j+1} v_{j+2}$. If x lies outside both $\overleftrightarrow{v_j v_{j+1}}$ and $\overleftrightarrow{v_{j+1} v_{j+2}}$, then x does not belong to S. If x lies outside just $\overleftrightarrow{v_j v_{j+1}}$, then x belongs to S if and only if it belongs to $S\text{-}seg_j$. Similarly, if x lies outside just $\overleftrightarrow{v_{j+1} v_{j+2}}$, then x belongs to S if and only if it belongs to $S\text{-}seg_{j+1}$ (see Fig. 21).

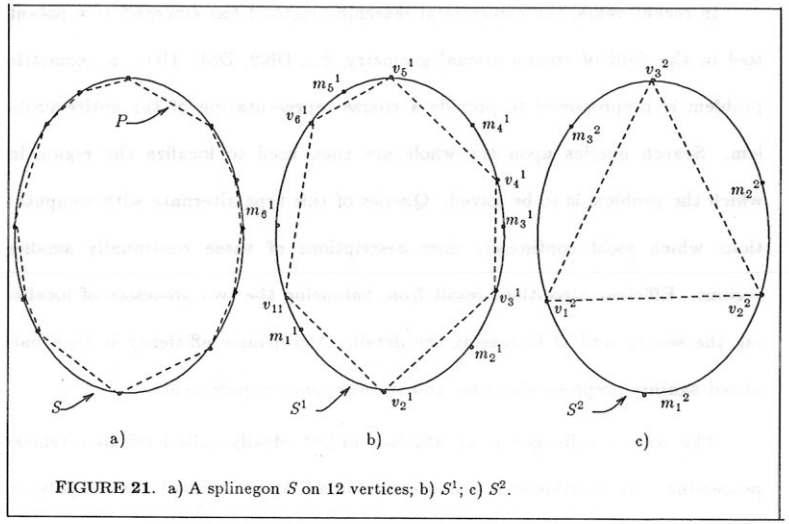

FIGURE 21. a) A splinegon S on 12 vertices; b) S^1; c) S^2.

Corollary. The inclusion of a point in a convex splinegon of N vertices can be decided in $O(B_1 + \log N)$ operations using the hierarchical method.

Proof. As described above. □

These algorithms provide an excellent example both of the adaptability of splinegons to hierarchical methods and of the efficacy of the carrier polygon approach. The desired hierarchy is readily available from the initial description of the polygon or splinegon, so no preprocessing or additional space is necessary. At each stage, the carrier polygon under consideration provides a sufficient approximation to the splinegon.

3.5. Determining the Area Enclosed by a Splinegon

In 1975, Shamos discussed the problem of determining the area of a planar polygon P of N vertices $v_1, v_2, \ldots, v_N$ in counter-clockwise order [Sh1]. The area of a non-simple planar polygon is defined as follows: the area of each simple bounded region should be added once for each time the boundary of the polygon wraps around it in counter-clockwise order; the area should be subtracted once for each time the boundary of the polygon wraps around it in clockwise order. Thus, some areas may be counted a multiple number of times, and some planar polygons may have negative area.

Shamos first cited a simple formula for the computation of the area:

$$\frac{1}{2}\left|\sum_{i=1}^{N} x_i(y_{i+1}-y_{i-1})\right| .$$

Next he showed that the formula could be altered to reduce the number of multiplications. If N is odd, the area is given by

$$\frac{1}{2}\left|\sum_{i=1}^{N-1} (x_i-x_N)(y_{i+1}-y_{i-1})\right| ,$$

a computation requiring $N-1$ multiplications. If N is even, the area is given by

$$\frac{1}{2}\left|\sum_{i=2}^{N/2} (x_{2i-1}-x_1)(y_{2i}-y_{2i-2}) + (x_{2i-2}-x_N)(y_{2i-1}-y_{2i-3})\right| ,$$

a computation requiring $N-2$ multiplications.

If the polygon is not simple, then the boundary of P divides the plane into one infinite region and a number of finite regions. In traversing the boundary of P once in the order dictated by the indices of the vertices, those finite regions whose boundaries are traversed in counter-clockwise order are deemed to have positive area. Those traversed in clockwise order have negative area. The area of P is defined as the sum of the areas of the finite regions.

Although the carrier polygon approach generally applies only to the extension of convex algorithms, it enables us to find the area of an arbitrary splinegon S of N vertices. First we calculate the area of the carrier polygon. Then we add (resp. subtract) the area of each *S-seg*$_i$ for which e_i was found to be concave-right (resp. concave-left) during the counter-clockwise traversal of S.

This method works because we are performing arithmetic, rather than set operations. There exist simple splinegons which are defined on non-simple carriers in such a way that no two of the *S-segs* are disjoint (see Fig. 22). The areas of half of the *S-segs*, however, will be added, while the other half will be subtracted. The algorithm will produce a correct value for the area, despite all of the overlapping.

Theorem 5. The area of an arbitrary splinegon of N vertices can be computed in $N-1$ multiplications and N calls to the procedure F_1 if N is odd, and in $N-2$

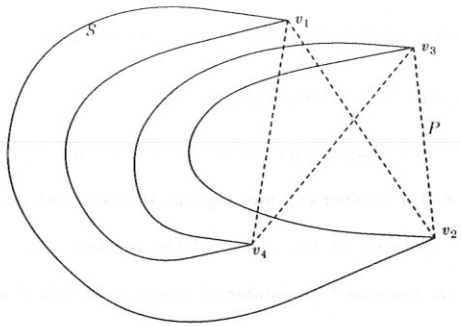

FIGURE 22. A simple splinegon S with a non-simple carrier polygon such that no two of the S-*segs* are disjoint.

multications and N calls to the procedure F_1 if N is even.

3.6. Discussion

The carrier polygon approach is particularly useful for processing convex splinegons. For each convex splinegon S, the carrier polygon P is itself convex. $P \subseteq S$. $S = P \bigcup_i S\text{-}seg_i$ where each $S\text{-}seg_i$ lies within a triangle $S\text{-}tri_i$ defined by $\overline{v_i v_{i+1}}$, $\overleftrightarrow{v_{i-1} v_i}$ and $\overleftrightarrow{v_{i+1} v_{i+2}}$. $P \bigcup_i S\text{-}tri_i$ forms a star-shaped polygon whose kernel is P.

The carrier polygon imposes sufficient structure on a convex splinegon that polygon algorithms can be extended to splinegons with the only modification being *ad hoc* procedures for the accommodation of the S-*segs*. The examples in this chapter include line-splinegon intersection computation, splinegon-splinegon

intersection detection, and a hierarchical algorithm for point inclusion.

The final algorithm in this chapter, for area computation, is somewhat of an anomaly. The splinegon need not be convex, nor even simple, but the carrier polygon approach enables the extension of Shamos' algorithm. The success of the extension, however, is due more to its dependence on arithmetic operations instead of set operations than to the power of the carrier polygon.

As described in Chapter 6, the carrier polygon can also be an important tool in processing monotone splinegons, but only when it is simple.

Chapter 4

Algorithms Focusing on the Bounding Polygon

4.1. Bounding Polygon Approach

In standard form, a splinegon S is given as a circular list of vertices, which completely define its carrier polygon P, and a pointer from each vertex to a description of the edge joining that vertex to its neighbor. A second polygon Q can be used to approximate the splinegon S. Q contains all of the vertices of S, which will be called *fixed vertices* of S. In addition, for each edge e_i of S which is not a line segment, let e_i^* represent the point of intersection of the ray tangent to e_i at v_i with the ray tangent to e_i at v_{i+1}. For some splinegon edges the two tangent rays might not intersect in the plane. The rays would intersect, however, if S were embedded on the surface of a sphere. We can allow the point e_i^* to represent the corresponding point on the projective plane at infinity (see Fig. 23a). Alternately, such splinegon edge can be broken into at most three pieces by the insertion of two new fixed vertices so that for each new edge the tangent rays will intersect at a point with finite coordinates (see Fig. 23b).

The point e_i^* is called a *pseudo-vertex* of S and is inserted into Q between the fixed vertices v_i and v_{i+1}. The polygon Q is called the *bounding polygon* of the *splinegon* S. An edge of Q which joins two fixed vertices is called a *fixed edge*. An edge joining a fixed vertex with a pseudo-vertex is called a *pseudo-edge*. A pseudo-edge $\overline{v_i e_i^*}$ (resp. $\overline{e_i^* v_{i+1}}$) is considered *loose* if its only intersection with the curved edge e_i is at the vertex v_i (resp. v_{i+1}). If $\overline{v_i e_i^*}$ (resp. $\overline{e_i^* v_{i+1}}$) intersects e_i in a line segment, then the edge is considered *tight*.

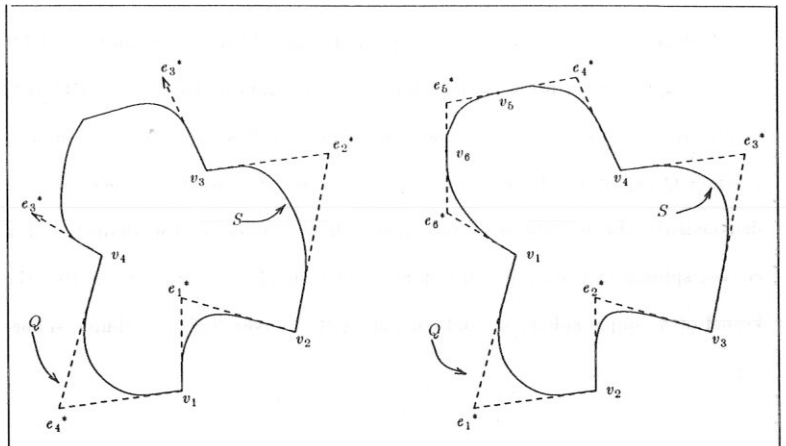

FIGURE 23. A pair of splinegons and their bounding polygons. One has an infinite pseudo-vertex, whereas the other does not.

The bounding polygon provides a useful tool for extending numerous algorithms on simple polygons to splinegons, particularly those algorithms which are vertex-based. The bounding polygon Q of a simple splinegon S of N vertices has at most $2N$ vertices iff we allow points at infinity, or at most $6N$ vertices otherwise. In general, the edge e_i of S can be viewed as a polygonal chain of an arbitrary number of vertices leading from v_i to v_{i+1} whose first edge lies on $\overline{v_i e_i^*}$, whose last edge lies on $\overline{e_i^* v_{i+1}}$, and which includes no reflex angles. During the execution of a polygon algorithm on the bounding polygon, processing of a pseudo-vertex e_i^* can include evaluation and insertion of pertinent information about the associated curve e_i.

Unfortunately, the specifications of the polygon Q are not readily available from the given description of the splinegon S. Computing each edge of Q costs $O(C_1)$ time. As a consequence, only infrequently will we specify the bounding polygon Q explicitly. Instead, each edge or vertex is computed as needed. We demonstrate the usefulness of this approach in computing the diameter of a convex splinegon, testing the monotonicity of a simple splinegon, computing the kernel of a simple splinegon, and computing the convex hull of a simple splinegon.

4.2. Computing the Diameter of a Convex Splinegon

The diameter of a convex polygon is realized by a pair of antipodal vertices. Shamos' algorithm for finding the diameter [Sh1] determines all such vertex pairs, computes all of the distances, and keeps the maximum. The diameter of a convex splinegon is also realized by a pair of antipodal points, but although those points will lie on the boundary of the splinegon, they may not be vertices. To find the diameter of a convex splinegon S of N vertices, we apply a modified version of the Shamos algorithm to the bounding polygon. After determining all antipodal vertex pairs for the bounding polygon, any pseudo-vertex can be replaced by the appropriate point on its corresponding edge to yield the pairs of antipodal points on the splinegon itself.

To run this algorithm, the entire bounding polygon must be computed, using $O(C_1 N)$ time. Next, each edge of the bounding polygon is oriented as a vector and translated in turn to the origin: pseudo-edges become vectors of the form $\overrightarrow{v_i e_i^*}$ and $\overrightarrow{e_i^* v_{i+1}}$; and fixed edges become vectors of the form $\overrightarrow{v_i v_{i+1}}$. Due

to the convexity of S, these vectors will be in non-decreasing order of polar angle.[3] The inclusive sector from a vector whose head is a fixed vertex v_i to a vector whose tail is v_i corresponds to that fixed vertex. If the two vectors have the same direction, then the fixed vertex will correspond to a sector consisting of a single ray. The sector strictly between a vector whose head is a pseudo-vertex e_i^* and a vector whose tail is e_i^* corresponds to that pseudo-vertex. A pseudo-vertex e_i^* was added to the bounding polygon only if the edge e_i was not straight. Thus, each pseudo-vertex corresponds to a non-empty open sector (see Fig. 24).

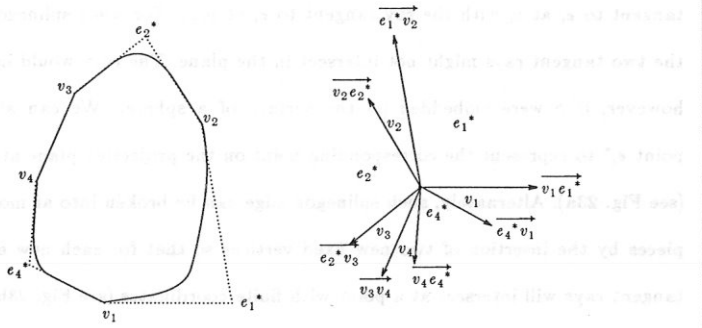

FIGURE 24. A splinegon and its bounding polygon and the corresponding vertex sectors.

Pick a line l passing through the origin, and in time $O(\log N)$ determine the two sectors in which it lies. To find all antipodal pairs, rotate the line l

[3] If the boundary of S is smooth at a vertex v_i, then the vectors $\overrightarrow{e_{i-1}^* v_i}$ and $\overrightarrow{v_i e_i^*}$ have the same orientation.

counter-clockwise. An antipodal pair changes only when l enters a new sector. Divide the vertex pairs reported into three groups: pairs having two fixed vertices; pairs with one fixed vertex and one pseudo-vertex; and pairs having two pseudo-vertices. The first group can be processed as in the original algorithm by computing the distance between the two vertices. The next two groups require the use of oracle A_1. For the second group, determine the point on that curved edge associated with the pseudo-vertex which lies at maximum distance from the fixed vertex. For the last group, determine the pair of points, one per curved edge, at maximum distance from each other. The maximum distance over the three groups is the diameter.

Theorem 6. The diameter of a convex splinegon of N vertices can be computed in $O((A_1 + C_1)\, N)$ time.

Proof. Finding the bounding polygon and translating all vectors to the origin requires $O(C_1 N)$ time. Since there are at most $2N$ sectors, there are at most $2N$ antipodal vertex pairs. By scanning sequentially, the corresponding pairs of antipodal points on S and the distances between them can all be determined in $O(A_1 N)$ time. Thus, the entire algorithm runs in $O((A_1 + C_1)\, N)$ time. □

Given that the object focused upon here is a convex splinegon S, one might expect to apply the carrier polygon approach. But to determine the maximum interior distance of S, we need to focus not on a polygon contained in S, but a polygon containing S. The polygon defined by $P \bigcup_i S\text{-}tri_i$ is not convex, so it is useless. But the bounding polygon Q of a convex splinegon P is always convex. Running the original polygon algorithm on Q produces a list of antipodal pairs

of vertices. At this point, the original algorithm would compute the Euclidean distance defined by each pair of vertices and select the maximum. The splinegon algorithm provides special procedures for antipodal pairs involving pseudo-vertices.

4.3. Determining the Monotonicity of a Simple Splinegon

A splinegon, like a polygon, is monotone in a direction $\vec{z}$ if it can be decomposed into two chains each monotone in direction $\vec{z}$. But to determine if a polygonal chain is monotone, it is not necessary to examine every edge. Suppose that the points $v_1, p_1, p_2, \ldots, p_n, v_2$ form a convex polygonal chain M and that the points v_1, w, v_2 form an acute triangle such that p_1 lies on the line segment $\overline{v_1 w}$ and p_n lies on $\overline{v_2 w}$. Then the chain M is monotone in a direction $\vec{z}$ if and only if the chain H defined by v_1, w, v_2 is also monotone in the direction $\vec{z}$ (see Fig. 25a). Similarly, a splinegon edge e_1 extending from v_1 to v_2 for which $\overline{v_1 w}$ and $\overline{v_2 w}$ are tight pseudo-edges is monotone in a direction $\vec{z}$ if and only if H is monotone. If the pseudo-edge $\overline{v_1 w}$ (resp. $\overline{v_2 w}$) is loose, however, then in general the monotonicity of e_1 coincides with the monotonicity of H. But e_1 is monotone in the direction orthogonal to $\overline{v_1 w}$ (resp. $\overline{v_2 w}$), and H is not (see Fig. 25b).

Consequently, if all edges of the bounding polygon Q associated with a splinegon S are tight, then S is monotone in a direction $\vec{z}$ if and only if Q is monotone in the same direction. But if Q is not monotone in any direction and if $\overline{v_i e_i^*}$ (resp. $\overline{e_i^* v_{i+1}}$) is loose, then S could still be monotone in the direction orthogonal to $\overleftrightarrow{v_i e_i^*}$ (resp. $\overleftrightarrow{e_i^* v_{i+1}}$). Thus to determine the monotonicity of a

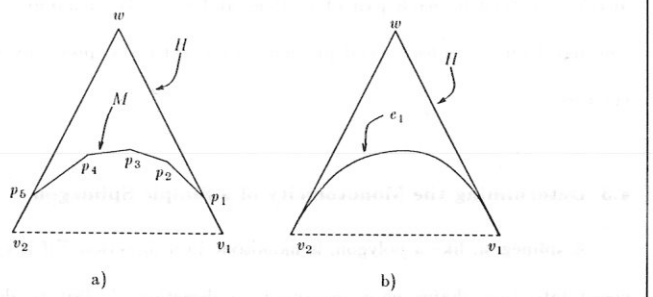

FIGURE 25. a) Convex polygonal chain inscribed in a triangle; b) Curved edge inscribed in a triangle.

simple splinegon, the Preparata-Supowit [PS] method can be applied to its bounding polygon with the following modification. Instead of processing all of the directions from $\overrightarrow{v_i e_i^*}$ to $\overrightarrow{e_i^* v_{i+1}}$, inclusive, as a group, process the directions between the two vectors as a group and process the vectors $\overrightarrow{v_i e_i^*}$ and $\overrightarrow{e_i^* v_{i+1}}$ separately according to whether the corresponding edges were tight or loose. We describe the algorithm below.

As in the previous example, orient all edges of the bounding polygon as vectors. Assign each fixed edge a key of 0. If pseudo-edge $\overrightarrow{v_i e_i^*}$ (resp. $\overrightarrow{e_i^* v_{i+1}}$ is tight, associate with it a key of 0; otherwise, assign it a key of 1. We calculate the edges of the bounding polygon, one by one, and push the vectors as translated to the origin, onto the queue L. After pushing all of the vectors, push a second copy of the first. Creating the queue L costs $O(C_1 N)$ time.

We process the list of at most $2N + 1$ vectors one by one, retaining the significant information in a new list M of vectors ordered by polar angle. Pop

the first vector from L and insert it into the empty list M along with its key and with three tags all initialized to 0: the forward, the backward, and the self. These tags may assume values in the set $\{0,1,2\}$. If a tag having a value of 2 is "incremented", it retains the value 2. The last vector inserted into M is called the current vector.

If the top vector of L describes the same polar angle as does the current vector, compare their keys. If both vectors have the same key, delete the top vector. Otherwise keep as the current vector whichever one has the larger key and delete the other. Consider the angle from the current vector to the top vector. If it belongs to the interval $(0,\pi)$ (resp. $(-\pi,0)$), we shall move forward (resp. backward) through M in order to insert the top vector. Begin by incrementing the forward (resp. backward) tag on the current vector. Increment the self tag only if the current vector has a key of 1 or if this move does not represent a change in direction. Then move forward (resp. backward) through M, incrementing all three tags on every vector and deleting any vector having three identical tags, until locating the position for the new vector. Pop it from L and insert it into M, making the backward (forward) tag match the forward (backward) tag of the vector preceding it, and the forward (backward) and self tags match the backward (forward) tag of the vector ahead of it (see Fig. 26).

Each vector is inserted into M once, requiring constant time. Each subsequent time it is processed, all three of its tags are incremented using constant time. But when all three tags on a vector equal 2, the vector is deleted. As each of the $O(N)$ vectors will be processed at most three times, the cost of

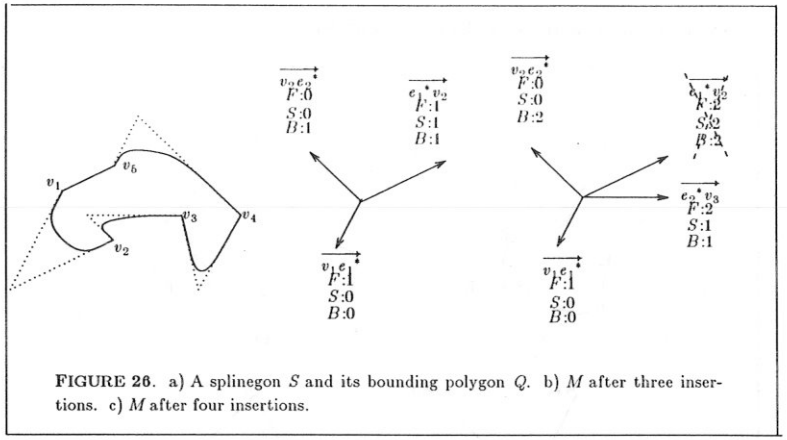

FIGURE 26. a) A splinegon S and its bounding polygon Q. b) M after three insertions. c) M after four insertions.

creating the list M is at most $O(N)$.

At the termination of the above algorithm, we have the partition of the polar range $[0,2\pi)$ by $O(N)$ vectors, each labeled with either a 1 or a 2, into $O(N)$ sectors which can identified as a 1 or a 2 by the forward and backward tags on the vectors bounding it. Pick a pair of rays r_1, r_2 which form a straight angle at the origin. In $O(\log N)$ time, determine whether r_1 (r_2) contains a vector in M or lies in a sector between two vectors and assign it the appropriate label. Rotate r_1 and r_2 in tandem counter-clockwise around the origin, changing the labels whenever either intersects a new vector or enters a new sector and recording every polar-angle interval in which both rays are assigned a 1. Since the labels change at most $O(N)$ times, there are at most $O(N)$ intervals reported. Thus this process requires $O(N)$ time. Whenever both rays are assigned a 1, T is monotonic in the direction normal to the two rays. If there is no angle at which both rays are assigned a 1, then T is not monotone.

Theorem 7. The directions in which a simple splinegon of N vertices is monotone can all be determined in $O(C_1 N)$ time.

Proof. As described above. □

As noted at the beginning of this section, if all edges of the bounding polygon Q associated with a splinegon S are tight, then S is monotone in a direction $\vec{z}$ if and only if Q is monotone in the same direction. But for each loose pseudo-edge, S could be monotone in the direction orthogonal to that edge, even though Q is not. Thus, to determine the monotonicity of S, we run the original polygon algorithm on Q, but perform separate calculations on the at most $2N$ directions determined by the loose pseudo-edges.

4.4. Computing the Kernel of a Simple Splinegon

To compute the kernel of a simple polygon of N vertices, Lee and Preparata [LP2] developed a vertex-based algorithm which runs in $O(N)$ time. The Lee-Preparata algorithm assumes that the first vertex v_1 of the polygon P is reflex, for if no vertex were reflex, the polygon would be convex and thus serve as its own kernel. It also assumes that the vertices are numbered in counterclockwise order around the boundary of P. The algorithm begins with the first vertex and then moves from vertex to vertex. Upon reaching a vertex v_i, the following information is available:

1) A doubly linked list of vertices which describes the boundary of the convex region K which is visible to all edges from v_N to v_i. If K is unbounded, the list is linear, and the vertices at the list tail and at the list head are both

points in the projective plane at infinity associated with a particular direction. If K is bounded, then the list is circular and all of its vertices are finite.

2) A pair of vertices F and L from K at maximum distance from v_i such that $\overrightarrow{v_iF}$ and $\overrightarrow{v_iL}$ both support K and such that the clockwise wedge from $\overrightarrow{v_iF}$ to $\overrightarrow{v_iL}$ contains K. If K is bounded, then F and L always represent finite points in the plane. If K is unbounded, however, F (resp. L) may represent the point at infinity at the tail (resp. head) of K's list.

The computation to be performed at a vertex v_i depends upon whether that vertex is reflex or convex.

Suppose that v_i is reflex (resp. convex) and that F (resp. L) lies on or to the left of $\overrightarrow{v_{i+1}v_i}$ (resp. on or to the right of $\overrightarrow{v_iv_{i+1}}$). Then trace the boundary of K from F to L in the counterclockwise direction (resp. from L to F in the clockwise direction). Stop upon finding a point k' where $\overrightarrow{v_{i+1}v_i}$ (resp. $\overrightarrow{v_iv_{i+1}}$) intersects the boundary of K. If no such point is found, then the kernel of P is null, so the algorithm halts. Otherwise, insert k' in the appropriate position as a vertex of K.

Next, trace the boundary of K in the clockwise (resp. counterclockwise) direction from k' until reaching a second point k'' of the intersection of K and $\overrightarrow{v_{i+1}v_i}$ (resp. $\overrightarrow{v_iv_{i+1}}$). If we reach a point at infinity at the list tail without discovering a point k'', then let k'' be the point at infinity having direction $\overrightarrow{v_{i+1}v_i}$ (resp. $\overrightarrow{v_iv_{i+1}}$). In either case, insert k'' in the appropriate position as a vertex of K, and set $F = k''$ (resp. $L = k''$). Delete all vertices of K between k' and k'' in

clockwise (resp. counterclockwise) order (see Fig. 27).

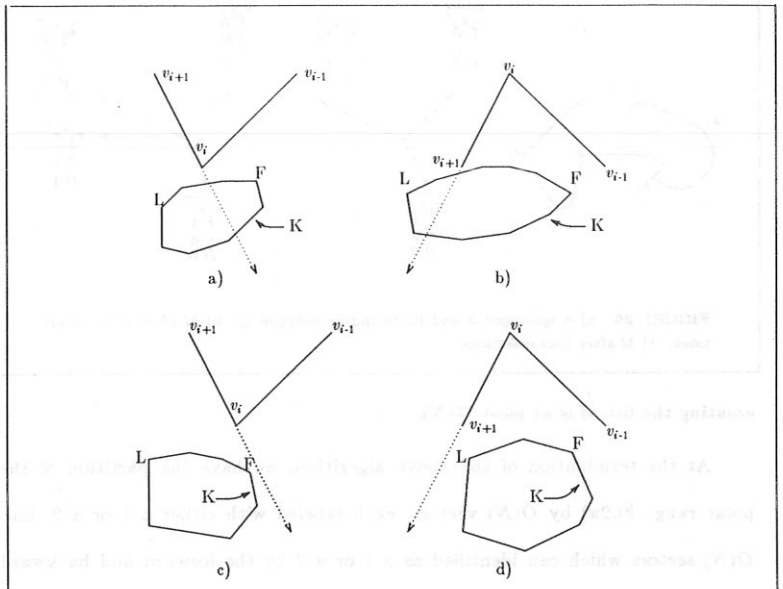

FIGURE 27. a) v_i is reflex and F lies on or to the left of $\overrightarrow{v_{i+1}v_i}$. b) v_i is convex and L lies on or to the right of $\overrightarrow{v_iv_{i+1}}$. c) v_i is reflex and F lies to the right of $\overrightarrow{v_{i+1}v_i}$. d) v_i is convex and L lies to the left of $\overrightarrow{v_iv_{i+1}}$.

Suppose that v_i is reflex (resp. convex) and that F (resp. L) lies to the right of $\overrightarrow{v_{i+1}v_i}$ (resp. $\overrightarrow{v_iv_{i+1}}$). In this case, K remains unchanged.

In all cases, before proceeding to the next vertex v_{i+1}, the algorithm performs a final update on both L and F.[4] Trace the boundary of K counterclockwise beginning with L (resp. F) until finding a vertex k_j such that either k_{j+1}

[4] In general, whenever one of F or L was set to k'' above, this final update will leave that value unchanged. The exception is the special case where v_i is convex and the line segment $\overline{v_iv_{i+1}}$ contains both k' and k''. In this instance, L must be revised a second time.

lies to the left of (resp. on or to the right of) $\overrightarrow{v_{i+1}k_j}$ or such that k_j is the point at infinity at the list head. Set $L = k_j$ (resp. $F = k_j$).

Lee and Preparata show that the algorithm runs in linear time since all but two of the edges traced in attempting to revise K are always removed, since F and L move around K only in a counterclockwise direction, and since for each v_{i+1} for which there exists a $p \in K$, $\sum_{j=1}^{i} \alpha_j < 3\pi$, where α_j represents the interior angle of the triangle $\Delta p v_j v_{j+1}$ at p.

We can compute the kernel of a simple splinegon S also in linear time by applying a modified version of their algorithm to its bounding polygon Q. The modifications necessary to accommodate pseudo-vertices and their related curved edges are derived from the following observations. For a point within a splinegon to be visible from an edge e_i which is concave-out, the point must lie within the wedge at the pseudo-vertex e_i^* defined by the extension of the rays $\overrightarrow{v_i e_i^*}$ and $\overrightarrow{v_{i+1} e_i^*}$ (see Fig. 28a). Thus, a concave-out curved edge from v_i to v_{i+1} defines the same visible region as would the pair of straight pseudo-edges $\overline{v_i e_i^*}$ and $\overline{e_i^* v_i}$. If the edge is concave in, then a visible point must lie within the convex region defined by the rays $\overrightarrow{e_i^* v_i}$ and $\overrightarrow{e_i^* v_{i+1}}$ and by the curved edge e_i from v_i to v_{i+1} (see Fig. 28b). Thus, a concave-in curved edge from v_i to v_{i+1} defines a somewhat smaller visible region than that determined by the pair of straight pseudo-edges $\overline{v_i e_i^*}$ and $\overline{e_i^* v_i}$.

The only required modifications to the algorithm pertain to convex pseudo-vertices. The efficacy of those modifications depends on the following lemma:

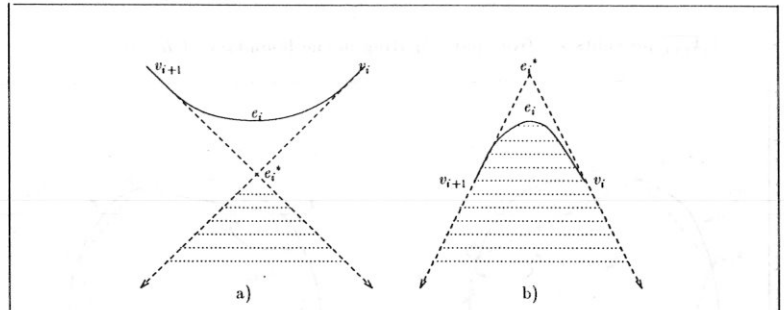

FIGURE 28. a) Region visible from an edge e_i which is concave-out. b) Region visible from an edge e_i which is concave-in.

Lemma. Given a simple splinegon S, at most one segment of a curved edge e_i can lie on the boundary of the kernel of S.

Proof. Suppose that e_{i1} and e_{i2} are two distinct segments of e_i, in counterclockwise order, both of which belong to the boundary of the kernel of S. The two segments are joined by a convex chain of straight edges. Let $\overline{k_j k_{j+1}}$ represent the straight edge which immediately follows e_{i1}. The line $\overleftrightarrow{k_j k_{j+1}}$ must contain some vertex q of the bounding polygon Q lying on a chain of edges which extends in counterclockwise order from v_{i+1} to v_i.

1) Suppose that $\overrightarrow{k_{j+1} k_j}$ contains q (see Fig. 29a). Then some chain of edges must join v_i to q. At best, q lies nearly at the point at infinity and a single edge connects v_i and q. Thus $\overleftrightarrow{v_i q}$ is nearly parallel to $\overleftrightarrow{k_j k_{j+1}}$. Even so, the edge $\overline{v_i q}$ prevents e_{i1} from participating in the boundary of K.

2) Suppose instead that $\overrightarrow{k_j k_{j+1}}$ contains q (see Fig. 29b). Then some chain of edges joins v_{i+1} to q. But for that chain of edges to permit e_{i1} to participate in the boundary of K, q must lie to the left of $\overrightarrow{k_j v_{i+1}}$. Then, however,

$\overline{k_j k_{j+1}}$ prevents e_{i2} from participating in the boundary of K. □

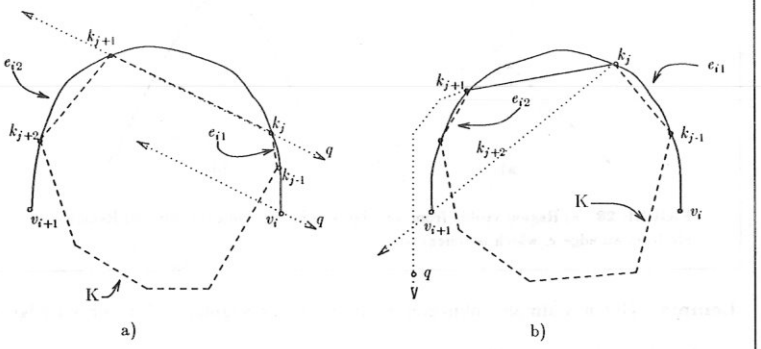

FIGURE 29. Examples for Lemma.

We make the following three modifications to the Lee-Preparata algorithm:

1) Upon reaching a fixed vertex v_i which precedes a convex pseudo-vertex e_i^*, stop after revising K but before making final revisions to F and L. Perform the following computation before continuing. Suppose that the current value of L is k_j. Test k_j in constant time to determine whether it lies to the right of $\overrightarrow{v_i v_{i+1}}$. If not, then mark the curved edge e_i in the representation of S as vacuous, for in no way can it participate in defining the kernel of S (see Fig. 30a). If it does, then test k_j in $O(B_1)$ time to determine whether it lies either on or to the right of e_i. If this second test fails, then mark the curved edge e_i in S as potent and assign it a pointer to the edge $\overline{k_j k_{j+1}}$ in K. Also mark the edge $\overline{k_j k_{j+1}}$ in K as the tail edge for e_i. The curved edge e_i may or may not participate in defining the kernel of S, but

a search beginning at k_j and moving in the counterclockwise direction will yield the answer (see Fig. 30b).

If the second test succeeds, trace the boundary of K in the clockwise direction from k_j until discovering an edge $\overline{k_m k_{m+1}}$ which intersects either e_i or $\overrightarrow{v_i v_{i+1}}$. If no such edge exists, then the kernel of S is null and we halt (see Fig. 30c). If the reported edge crosses $\overrightarrow{v_i v_{i+1}}$ but does not cross e_i, then mark e_i in S as vacuous (see Fig. 30d). If the reported edge does cross e_i, then mark e_i in S as potent, and mark $\overline{k_m k_{m+1}}$ in Q as the tail edge for e_i (see Fig. 30e). In either of these last two cases, delete all vertices of K strictly between k_j and k_{m+1} in the clockwise direction. If e_i is vacuous, then these edges would have been deleted anyway in the processing of pseudo-vertex e_i^*. If e_i is potent, then e_i prevents these edges from contributing to the boundary of the kernel of S.

2) When reaching a fixed vertex v_{i+1} after having just processed a convex pseudo-vertex e_i^*, determine whether e_i in S has been marked potent. If so, perform the following computation before proceeding with the algorithm. Suppose that the current value of F is k_j. Since e_i has been marked potent, k_j must lie to the right of $\overrightarrow{v_i v_{i+1}}$. Test k_j in $O(B_1)$ time to determine whether it lies either on or to the right of e_i.

   a) If not, then add an extra pointer in the representation of the curved edge e_i in S to the edge $\overline{k_{j-1} k_j}$ in K, and label the edge $\overline{k_{j-1} k_j}$ as the head edge for e_i. The curved edge e_i may or may not participate in defining the kernel of S, but a search beginning at the tail edge, mov-

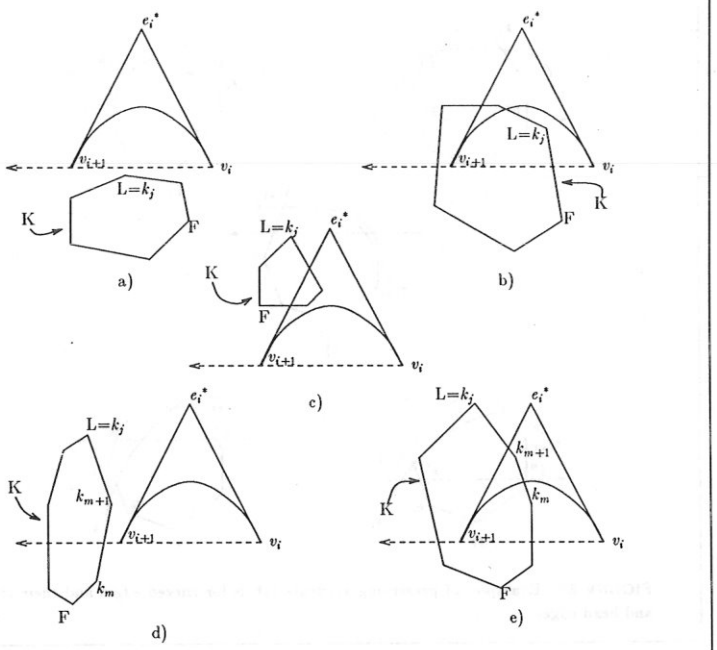

FIGURE 30. The five possible scenarios after revising K at a fixed vertex which precedes a convex pseudo-vertex.

ing in the counterclockwise direction, and ending at the head edge will yield the answer (see Fig. 31a).

b) If so, trace the boundary of K in the counterclockwise direction from k_j until discovering an edge $\overline{k_m k_{m+1}}$ which intersects e_i. Such an edge must exist. Add a pointer from e_i in S to the edge $\overline{k_m k_{m+1}}$ and also mark $\overline{k_m k_{m+1}}$ in Q as the head edge for e_i (see Fig. 31b). In this case,

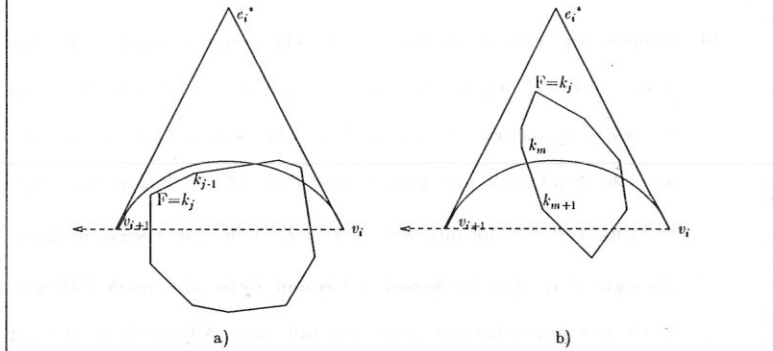

FIGURE 31. The two possible scenarios after processing a pseudo-vertex associated with a potent curved edge of S.

delete all vertices of K strictly between k_j and k_m in the counterclockwise direction. The curved edge e_i prevents these edges from contributing to the boundary of the kernel of S.

3) Next, we must guarantee that, as K is repeatedly revised, labels and pointers to tail edges and head edges of potent curved edges are updated. Also, edges which become vacuous must be so identified. The only instances in which these updates must be made are those in which the deleted edges of K include some portion of either one or both of the tail edge $\overline{k_t k_{t+1}}$ and head edge $\overline{k_h k_{h+1}}$ for some curved edge e_i.

a) Suppose the deleted portion runs in the counterclockwise direction from a point k', which lies between k_{h+1} and k_t, and ends at a point k'', which lies between k_{h+1} and k'. In other words, the tail edge and the head edge and all intervening edges are all deleted. In this case,

mark the curved edge e_i as vacuous (see Fig. 32a).

b) Suppose the deleted portion runs in the counterclockwise direction from a point k', which lies between k_{t+1} and k_h, and ends at a point k'', which lies between k_{t+1} and k'. In other words, both the tail edge and the head edge are deleted, but some of the intervening edges remain. Test $\overline{k'k''}$ for intersection with e_i. If the entire segment lies to the right of e_i, then the kernel of S is null. Otherwise, mark $\overline{k'k''}$ both as the new head edge and as the new tail edge. Adjust the pointers at e_i (see Fig. 32b).

Use as many of the following as pertain, if and only if neither of the above cases apply.

c) Suppose the deleted portion runs in the counterclockwise (resp. clockwise) direction from a point k', which lies between k_t and k_{t+1} (resp. k_h and k_{h+1}). In other words, the interior part of the tail edge (resp. head edge) is deleted. In this case, mark $\overline{k_tk'}$ (resp. $\overline{k'k_{h+1}}$), the remaining portion, as the new tail (resp. head) edge for e_i. Update the pointer at e_i (see Fig. 32c).

d) Suppose the deleted portion runs in the clockwise (resp. counterclockwise) direction from a point k', which lies between k_t and k_{t+1} (resp. k_h and k_{h+1}), to a point k''. In other words, the exterior part of tail edge (resp. head edge) is deleted. Test the point k' to determine whether it lies to the right of e_i. If so, then mark $\overline{k'k''}$ as the new tail (resp. head) edge. If not, then mark $\overline{k'k_{t+1}}$ (resp. $\overline{k'k_h}$) as the new

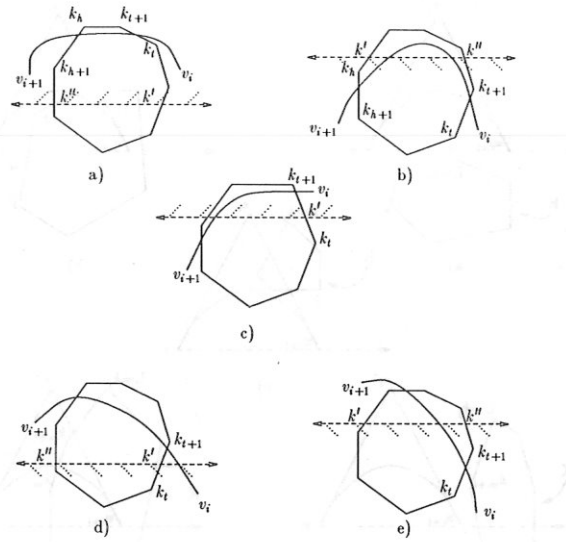

FIGURE 32. Examples of preserving accurate labels for curved edges and their tail and head edges.

tail (resp. head) edge. Update the pointer at e_i (see Fig. 32d).

e) Suppose the deleted portion runs from a point k' through both k_t and k_{t+1} (resp. k_h and k_{h+1}), and ends at a point k''. In other words, the entire tail (resp. head) edge is deleted. Mark $\overline{k'k''}$ as the new tail (resp. head) edge and update the pointer at e_i (see Fig. 32e).

These three routines provide the basis for the following theorem:

Theorem 8. The kernel of a simple splinegon P of N vertices can be determined in $O((B_1 + C_1)\, N)$ time.

Proof. Determining the bounding polygon requires $O(C_1 N)$ time. The entire Lee-Preparata algorithm runs in $O(N)$ time. If no tracing is done, then modification #1 requires constant time. If n edges are traced, then $n-1$ edges are deleted. Thus the tracing and the deleting may be charged to those edges, and only $O(B_1)$ time needs be charged to each call to modification #1. The same argument applies to modification #2. Whenever the original algorithm revises K by deleting all vertices in a particular direction between k' and k'', it explicitly traces all of the intervening edges. Thus the information about the relative positioning of head and tail edges can be computed at the same time and all marks and pointers may be updated, incurring at most a constant charge per edge. When the main algorithm is complete, we perform one final trace around K. We test the edges between each tail-head pair for intersection with the respective curved edge, and update K accordingly. This single pass around K and S requires $O(B_1 N)$ time. □

The kernel of the bounding polygon is a subset of the kernel of the splinegon. A concave-out curved edge defines the same visible region as do the pair of associated pseudo-edges, but a concave-in curved edge defines a somewhat smaller visible region than that determined by the associated pseudo-edges. To compute the kernel of a splinegon, therefore, we run the original algorithm on the bounding polygon, but provide special processing for pseudo-vertices, and the adjacent pseudo-edges, associated with concave-in splinegon edges.

4.5. Computing the Convex Hull of a Simple Splinegon *(revised 6/87)*

Schäffer and Van Wyk [SV] have already extended the Graham-Yao algorithm for computing the convex hull of a simple polygon [GY] to process piecewise smooth Jordan curves. To do so, they first revise the Graham-Yao vertex-based algorithm to run as an edge-based algorithm instead. Then they use the direct approach described in the next chapter to extend the algorithm. Using the bounding polygon and special procedures for pseudo-vertices, we can extend the vertex-based algorithm directly.

The Graham-Yao algorithm assumes that the vertices $v_1, v_2, \cdots v_m, v_{m+1}, \cdots v_N$ of the simple polygon P are given in clockwise order around the boundary, that v_1 is the vertex of minimum x-coordinate, and that v_m is the vertex of maximum x-coordinate. Define the path along P from v_i to v_j as a *pocket* of P if no vertex along the path lies to the left of the directed line segment $\overrightarrow{v_i v_j}$; call $\overrightarrow{v_i v_j}$ the *top* of the pocket; say that a vertex lies inside (resp. outside) the pocket if it lies (resp. does not lie) in the closed region bounded by the pocket and its top. Graham and Yao characterize the task of finding the convex hull of P as that of identifying a circular list of vertices such that each consecutive pair delimits a pocket of P and such that the pocket tops and the vertices form a convex polygon (see Fig. 33a). The set of vertices of the convex hull must include both v_1 and v_m and may not include any vertex lying inside a pocket of P, except for its endpoints. Thus the convex hull problem can be divided into two symmetric pieces: compute the top hull (resp. bottom hull) of P, which corresponds to the *left hull* of the oriented chain $v_1, v_2, \ldots v_{m-1}, v_m$

(resp. $v_m, v_{m+1}, \ldots v_N, v_1$).

The *left hull* algorithm maintains a stack Q of candidate hull vertices, where q_0 (resp. q_t) represents the bottom (resp. top) element of the stack, with the invariants that $q_0, q_1, \cdots, q_t$ always form a convex polygon and for $2 \leq i \leq t$, q_{i-1}, q_i always delimit a pocket of P. To find the top hull of P, the algorithm begins by setting $q_0 = v_m$, $q_1 = v_1$, and q_2 to be the first vertex lying to the left of the directed line segment $\overrightarrow{v_1 v_m}$. After pushing a vertex v_i onto the stack, the algorithm moves from vertex to vertex along the chain, searching for the first vertex x outside the current convex polygon. If v_{i+1} lies to the left of $\overrightarrow{q_{t-1} q_t}$, then it automatically becomes x. Otherwise, the algorithm tests whether v_{i+1} belongs to the pocket with endpoints q_{t-1} and q_t. If so (resp. not), then x will be the first successor of v_{i+1} to lie to the left of $\overrightarrow{q_{t-1} q_t}$ (resp. $\overrightarrow{q_t q_0}$). Before inserting x into the stack, as many vertices are popped from the stack as necessary so that x lies to the right of the new $\overrightarrow{q_{t-1} q_t}$. The algorithm uses linear time and space, as each vertex not rejected outright is inserted into the stack exactly once and deleted at most once. At termination, the path from q_t to q_0 along P also forms a pocket, and thus Q describes the top hull of P.

The splinegon algorithm is nearly identical to the polygon algorithm, but it must consider both the fixed vertices and the pseudo-vertices of the bounding polygon. In this application, however, we never compute the coordinates of the pseudo-vertices explicitly; each pseudo-vertex merely points to a description of the corresponding curved edge. Thus, we need to define what we mean by the directed line segment $\overrightarrow{vw}$ where at least one of v and w is a pseudo-vertex:

1) $\overrightarrow{v_i e_j^*}$ represents the directed line segment of maximum length which extends from v_i to a point y on e_j and which supports e_j so that each point of e_j lies on or to the right of $\overrightarrow{v_i e_j^*}$.
2) $\overrightarrow{e_i^* v_j}$ represents the directed line segment of maximum length which extends from a point x on e_i to v_j and which supports e_i at x so that each point of e_i lies on or to the right of $\overrightarrow{e_i^* v_j}$.
3) $\overrightarrow{e_i^* e_j^*}$ represents the directed line segment which extends from a point x on e_i to a point y on e_j and which supports e_i at x and e_j at y so that each point of either e_i or e_j lies on or to the right of $\overrightarrow{e_i^* e_j^*}$.

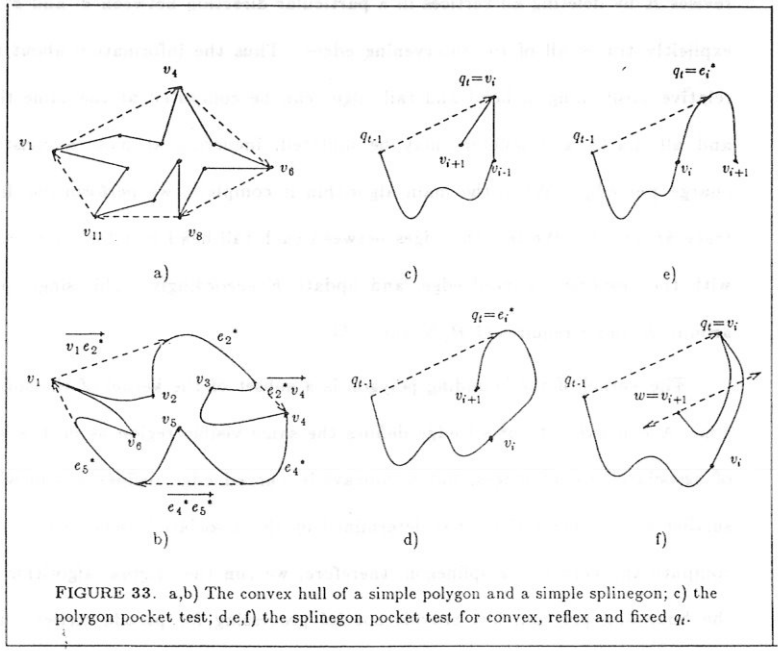

FIGURE 33. a,b) The convex hull of a simple polygon and a simple splinegon; c) the polygon pocket test; d,e,f) the splinegon pocket test for convex, reflex and fixed q_t.

Next, we augment our definitions of what it means for a vertex x to lie to the left of $\overrightarrow{vw}$:

1) e_j^* lies to the left of $\overrightarrow{vw}$ if any portion of e_j lies to the left of $\overrightarrow{vw}$.
2) If $\overrightarrow{v e_j^*}$ (resp. $\overrightarrow{e_j^* w}$) intersects e_j only at v_{j+1}, we shall consider v_{j+1} to lie to the left of the respective directed line segment.

Given these augmented definitions, the convex hull of a simple splinegon P can

now be identified by a circular subsequence of the vertices of the bounding polygon Q such that each consecutive pair delimits a pocket and such that a convex splinegon is formed by the pocket tops, the fixed vertices, and the portions of those edges identifed by pseudo-vertices which join adjacent pocket tops (see Fig. 33b).

As in the polygon case, the circular list of vertices is determined by two applications of the *left hull* algorithm. However, we must first insert the points of minimum and maximum x-coordinate as fixed vertices and renumber the vertices accordingly. We must also define the test to determine whether a vertex v immediately succeeding q_t and lying on or to the right of $\overrightarrow{q_{t-1}q_t}$ lies inside a pocket delimited by q_{t-1} and q_t. In the polygon case, the pocket test is simple: for $q_t = v_i$, v_{i+1} lies inside (resp. outside) the pocket if it does (resp. does not) lie to the left of the directed line segment $\overrightarrow{v_{i-1}v_i}$ (see Fig. 33c). In the splinegon case, we have multiple cases. If q_t is a reflex pseudo-vertex, then v belongs to the pocket (see Fig. 33d). If q_t is a convex pseudo-vertex, then v does not belong to the pocket (see Fig. 33e). If q_t is a fixed vertex v_i, let w represent whichever of v_{i-1} and v_{i+1} lies closest to the line l containing $\overrightarrow{q_{t-1}q_t}$. Find the intersection with both e_{i-1} and e_i of the line passing through w parallel to l. The intersection either consists of one component from each edge, or of both w and a second point from one edge and one component from the other. In the latter case, discard w. Now, if the one component from e_i is to the left (resp. right) of the one component from e_{i-1}, then v does (resp. does not) belong to the pocket (see Fig. 33f).

Theorem 9. The convex hull of a simple splinegon of N vertices can be computed in time and space $O((B_1 + C_1 + D_1)\, N)$.

Proof. As modified, the Graham-Yao algorithm will provide a list of vertices describing the left hull of each half of the splinegon. That list will include both fixed vertices and pseudo-vertices. A single transversal of that final list can determine which portion of each of the curved edges associated with a listed pseudo-vertex actually lies on the convex hull. Then the convex hull is formed by linking fixed vertices and curved segments with straight segments. □

In this algorithm, the structure of the bounding polygon was used more than the polygon itself. In fact, neither the pseudo-edges nor the pseudo-vertices were ever explicitly determined, for the approximation they would have provided for curved edges would not have been sufficiently accurate to decide which edges participate in the convex hull. The vertex list for the bounding polygon, however, did contain an entry for each non-straight edge of the splinegon. Thus we could apply the original polygon algorithm to the bounding polygon and yield the convex hull of the splinegon merely by adding special procedures for processing those vertices which were really pointers to curves rather than vertices.

4.6. Discussion

The bounding polygon approach is more versatile than the carrier polygon approach. When computed explicitly, it provides a good approximation for the splinegon. A convex splinegon has a convex bounding polygon. A splinegon is

monotone if its bounding polygon is monotone. A splinegon has a kernel only if the edges of its bounding polygon define a non-empty visible region. The efficacy of the bounding polygon is due in large part to the fact that for many calculations on polygons, or on splinegons, attention need be given only to vertices at reflex angles, interior angles measuring more than 180 °. A splinegon whose edges are all concave-in has reflex angles only at fixed vertices, and the adjacent pseudo-edges accurately determine the angle at a fixed vertex. A curved edge which is concave-out corresponds to a polygonal chain composed entirely of reflex angles. The associated pseudo-vertex and the adjacent pseudo-edges adequately approximate it.

As demonstrated in Section 4.5, the bounding polygon provides an alternative tool for expressing the splinegon itself. It allows a vertex-based polygon algorithm to be extended to splinegons with its structure intact. It also allows the creators of new algorithms to write them in a general format which encompasses splinegons, but with the more restricted polygon algorithm clearly visible within. In either case, separate procedures exist for the processing of fixed vertices and for the processing of pseudo-vertices.

It is important to note that the bounding polygon of a simple polygon need not be simple. Nonetheless, algorithms on simple polygons extend to simple splinegons using the bounding polygon technique, whether or not the bounding polygon is simple. For example, we may compute the kernel of a simple starshaped splinegon using a non-simple bounding polygon which clearly has no kernel.

Chapter 5

Algorithms Featuring the Direct Approach

5.1. Direct Approach

In the previous two chapters, we have presented two distinct methods for extending polygon algorithms. The carrier polygon approach primarily applies to extensions of algorithms on convex polygons. The bounding polygon approach has particular application in the extension of vertex-based algorithms. In general, however, edge-based algorithms need neither the artifice of focusing on the carrier polygon nor the artifice of focusing on the bounding polygon. Where the original algorithm considered the line segment from v_i to v_{i+1}, the revised algorithm considers the curved edge e_i which joins v_i to v_{i+1}. All that is needed is a revised procedure for processing the individual edges which accounts for the greater freedom enjoyed by curves.

In this chapter, we show how this direct approach enables the computation of the intersection of two convex splinegons, the computation of the horizontal visibility information for a simple splinegon, the decomposition of a simple splinegon into monotone pieces, the detection of the intersection of two simple splinegons, and the determination of the convex hull of a simple splinegon.

5.2. Computing the Intersection of two Convex Splinegons

As our first example of the success of the direct approach, we extend the method of [Sh1] to compute the intersection of the two convex splinegons P and

Q, each of at most N vertices. In the original algorithm, Shamos finds a point in the interior of one polygon and develops separate procedures for the cases where the point lies within the second polygon or lies in its exterior. To simplify the algorithm, we first use the technique of section 3.3 to determine a point in the intersection of the two splinegons. This process requires $O(C_1 \log N + A_1 + B_1)$ operations. In a constant number of additional operations, we can move this intersection point to get a point x which is interior to both P and Q or we can determine that the splinegons are tangent and no such point exists.

Let x be the origin of a polar coordinate system. Draw rays from x through each of the vertices of P, dividing the plane into sectors. For each vertex of Q, determine in which sector of P it lies. Although the first search may cost $O(\log N)$ time, the entire process may be completed in $O(N)$ time by proceeding sequentially around Q. Once the sector in which a vertex lies has been identified, $O(B_1)$ time determines whether the vertex is interior or exterior to P.

Scan around Q once, examining all pairs of consecutive vertices q_i, q_{i+1}. If both vertices are in the same sector, then use A_1 time to test the arc joining them for intersection with the bounding arc of P. The number of possible solutions is unlimited by the convexity restraint, but must be bounded by $O(A_1)$. If the vertices lie in different sectors, then the arc joining them must be tested for intersection against the bounding arcs of all of the intervening sectors. Since no backtracking is done, all intersection points can be determined in $O(A_1 N)$ time. The intersection consists of chains taken alternately from splinegons P and Q with the intersection points in between.

Theorem 10. The intersection of two convex splinegons of at most N vertices each can be computed in $O(A_1 N + C_1 \log N + B_1)$ time.

Proof. As described above. □

Shamos' original algorithm made many assumptions about the possible intersections between edges. If a pair of consecutive vertices q_i, q_{i+1} lay in the interior of P, he concluded that the edge joining them also belonged to the interior to P and could not intersect the boundary of P. If one vertex were inside P and one were outside, but the two vertices lay in the same sector, then exactly one intersection could occur between the edge $\overline{q_i q_{i+1}}$ and the bounding edge of the sector. If the two vertices lay in different sectors, then $\overline{q_i q_{i+1}}$ would intersect the boundary of P extending from one sector to the other exactly once. If the two vertices both lay outside of P and in the same sector, then the edge $\overline{q_i q_{i+1}}$ also lay outside of P. If the two vertices belonged to different sectors, however, then the edge $\overline{q_i q_{i+1}}$ could intersect the boundary of P extending from one sector to the other up to two times.

All of these assumptions hold when edges are line segments, but cease to be true when edges are allowed to be curves. Our modification of Shamos' algorithm consists of removing all of these assumptions and inserting additional tests for the intersection of pairs of edges. Despite these additional tests, the time required by the algorithm remains logarithmic in N.

5.3. Computing the Horizontal Visibility Information for a Simple Splinegon and Detecting the Simplicity of an Arbitrary Splinegon

Tarjan and Van Wyk [TV] give an $O(N\log\log N)$-time algorithm for computing the internal horizontal vertex visibility information for a simple polygon of N vertices. The horizontal line segments which join a vertex to its visible edge or edges define a partition of the polygon into trapezoids. Assuming that no polygon vertices have the same y-coordinate, each trapezoid contains exactly two polygon vertices: one on its top edge and one on its bottom edge. By using their $O(N\log\log N)$-time algorithm to compute both the internal and the external horizontal vertex visibility information for a polygon, they can detect whether a polygon is simple in $O(N)$ additional time. Both algorithms extend to splinegons using the direct approach [TV].

In its original form, the horizontal visibility algorithm assumes that edges possess the following two properties:

1) each edge crosses any horizontal line at most once;
2) if a set of edges crosses two horizontal lines, the order in which they cross the horizontal lines is the same on both lines.

Splinegon edges will also satisfy these two properties provided that they are all monotone in the y-direction. Thus, before applying the Tarjan-Van Wyk algorithm, we trace around the splinegon, determining the maximum and minimum point of each edge e_i in the vertical direction. Whichever of these points is not already a vertex of the splinegon must be inserted as a vertex in the appropriate position. This modification requires $O(C_1N)$ time, and at most $2N$ new vertices are added. Once this modification has been made, the algorithm runs unchanged except for the fact that it computes the intersection of horizontal lines with curved edges rather than straight edges and reports trapezoids bounded by a pair of horizontal line segments and a pair of y-monotone curved edges. The algorithm runs in $O(N\log\log N + (B_1 + C_1)N)$ time.

The original algorithm for simplicity testing possesses a different assumption about the behavior of edges: two edges intersect in their interiors if and only if when the endpoints are ordered by y-coordinate the edges intersect the horizontal lines through the middle two endpoints in different order. Although this property holds when all edges are line segments, it is invalid for curved edges. Tarjan and Van Wyk accommodate this discrepancy by adding a final stage to the splinegon version. If the splinegon still appears to be simple after running the original algorithm, they test the pair of curved side-edges from each of the trapezoids reported in either iteration of visibility testing for intersection. If no intersections are found, the splinegon is indeed simple. This revised algorithm runs in $O(N\log\log N + (A_1 + B_1 + C_1)N)$ time.

5.4. Decomposing a Simple Splinegon into Monotone Pieces[5]

Given the horizontal-vertex-visibility partition of a simple polygon, a y-monotone decomposition of the polygon can be computed in linear time by adding an edge between the polygon vertices of any trapezoid which has a polygon vertex lying in the interior of one of the parallel sides. Such a vertex is

[5] Joint work with D.P. Dobkin and C.J. Van Wyk [DSV].

called a *y-notch*. In decomposing a simple splinegon into monotone pieces, we must recognize that the two vertices we wish to join may not be visible to each other.

Theorem 11. A simple splinegon of N vertices can be decomposed into the union of monotone pieces with simple carriers in $O(N\log\log N + (B_1 + C_1)N)$ time. The total number of vertices in the decomposition is $O(N)$.

Proof. To decompose a simple splinegon S of N vertices into the union of monotone pieces, we begin by picking a distinguished direction. Next, we adjust the coordinate system so that the distinguished direction is the y-direction. Applying the Tarjan & Van Wyk horizontal-visibility algorithm to S will then yield trapezoids with curved side-edges. For each trapezoid containing a vertex v of S identified as a y-notch, a new edge is added. As in the polygon case, we add the line segment l which connects the two vertices of S lying on the trapezoid, provided that the interior of l does not intersect either side-edge. Otherwise, we determine the point x closest to v at which l intersects a side-edge and add the line segment $\overline{vx}$. In the second instance, we add a new vertex to the splinegon, but the total number of vertices added in this way is at most the number of y-notches.

After this decomposition, each splinegonal region is monotone, but individual carrier polygons need not be simple. To guarantee simplicity, trace each splinegon, top to bottom, around both sides simultaneously. Whenever two polygon edges cross each other, match the nearest vertex on one side with a vertex of the same y-coordinate on the other side (see Fig. 34). □

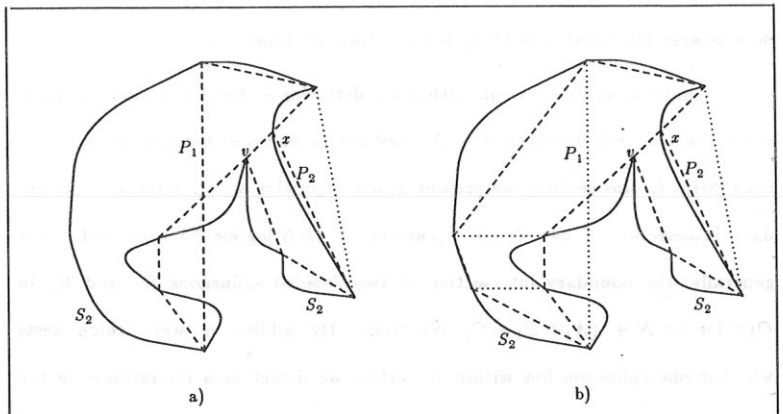

FIGURE 34. The decomposition of a simple splinegon S into the union of monotone pieces: a) one carrier polygon is non-simple; b) vertices have been added to make all carrier polygons simple.

5.5. Detecting the Intersection of two Simple Splinegons[6]

Detecting area intersection can be an easier problem than detecting boundary intersection. Chazelle and Dobkin showed that $\Omega(N)$ is a lower bound on the time required to detect the boundary intersection of two convex polygons of at most N vertices, even if preprocessing is allowed, making the Shamos algorithm for computing the intersection optimal for boundary intersection detection as well. Area intersection detection, however, can be completed in $O(\log N)$ time [CD2]. The algorithm presented in Section 5.2 will detect the boundary intersection of two convex splinegons of at most N vertices in $O(A_1N + C_1\log N + B_1)$ time, while the algorithm presented in Section 3.3

[6] Joint work with D.P. Dobkin and C.J. Van Wyk [DSV].

detects area intersection in $O(A_1 + B_1 + C_1 \log N)$ time.

Up until now, the best algorithm for detecting either the boundary intersection or the area intersection of N-sided simple polygons required $O(N \log N)$ time [SH]. In this section, we present a new algorithm which detects the boundary intersection of two N-sided polygons in $O(N \log \log N)$ time and, more generally, the boundary intersection of two N-sided splinegons K_1 and K_2 in $O(N \log \log N + (A_1 + B_1 + C_1) N)$ time. By adding a step which tests whether one splinegon lies within the other, we detect area intersection in the same amount of time.

Our approach is to create from K_1 and K_2 a merged splinegon M such that the boundaries of K_1 and K_2 are disjoint if and only if M is simple. Splinegon M consists of the edges of K_1 and K_2 together with a "bridge" between them which is composed of a constant number of edges. One way to find such a bridge was proposed by Hershberger [He]; it requires a linear-time algorithm for computing the convex hull of a simple splinegon [SV], and uses two cases depending on whether the convex hull of one splinegon contains the other. Our method for finding a bridge uses linear-time Jordan sorting [HMRT], an algorithm that plays a crucial role in the algorithms discussed in Section 5.3 for computing horizontal visibility information and for detecting simplicity.

Theorem 12. The boundary intersection of two N-sided simple splinegons can be detected in $O(N \log \log N + (A_1 + B_1 + C_1) N)$ time.

Proof. Our algorithm begins by using $O(C_1 N)$ time to determine the y-extent (minimum and maximum y-coordinates of any point) of K_1 and K_2. If the y-extents of K_1 and K_2 do not overlap, then K_1 and K_2 do not intersect. Otherwise, let y_{cut} represent a value which lies within both y-extents. Without loss of generality, we assume that each edge of K_1 and K_2 intersects the line $y = y_{cut}$ in no more than two points.[7]

For each i, we use $O(B_1 N)$ time to compute the sequence of points (p_{ij}) at which K_i crosses $y = y_{cut}$ in the order in which they appear around the boundary of K_i. Next we use the linear-time Jordan sorting algorithm [HMRT] to determine σ_i, the sequence (p_{ij}) sorted in increasing order of x-coordinate. A linear-time scan of σ_i in order of increasing x-coordinate allows us to label each point: the label $touch_i$ is assigned to the point p_{ij} if the boundary of K_i does not cross $y = y_{cut}$ at p_{ij}; the label in_i (resp. out_i) means that the interior of K_i lies to the right (resp. left) of p_{ij}.

Next, we merge σ_1 and σ_2 to form a single sorted sequence σ. Let σ' represent the sequence σ with all points labelled $touch_i$ removed. We now perform a single scan of σ and σ'. If σ' contains a sequence of the form $<in_1, in_2, out_1, out_2>$, then the boundaries of K_1 and K_2 must intersect. Otherwise, we pick any consecutive pair of crossing points $q_1 \in K_1$ and $q_2 \in K_2$, points that are labelled with different subscripts. Since y_{cut} belongs to the y-extent of both splinegons, we are guaranteed that such a pair exists.

The following process creates a new splinegon M such that the boundaries of K_1 and K_2 intersect if and only if M is not simple: pry each splinegon K_i

[7] We can remove this assumption by keeping only the leftmost and rightmost points of each connected component of the intersection. This technique is a generalization of that used by Van Wyk [V].

slightly open at q_i, so that q_i is split into two points q_i^+ and q_i^-, with q_i^+ above q_i^-; form M by joining the "slightly opened" K_1 and K_2 with the two line segments $\overline{q_1^+ q_2^+}$ and $\overline{q_1^- q_2^-}$. These edges do not cross each other, and they are chosen so that they do not cross any edge of K_1 or K_2 (see Fig. 35).

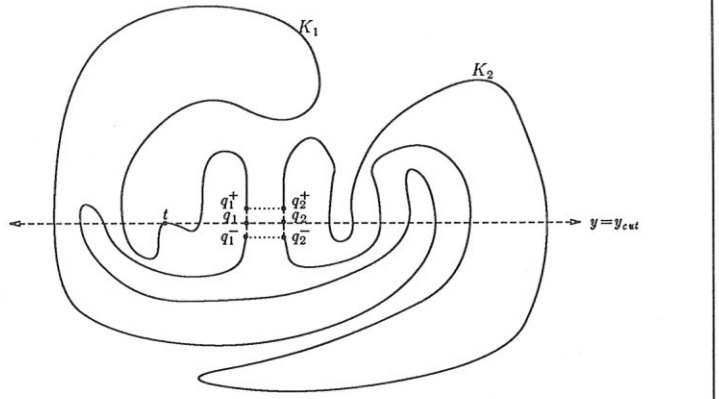

FIGURE 35. Merging K_1 and K_2 to form a single splinegon.

Finally, we use the algorithm described in Section 5.3 to test the simplicity of M in $O(N \log \log N + (A_1 + B_1 + C_1)\, N)$ time. This time represents the dominating factor in the overall time complexity of the algorithm. □

Corollary. The area intersection of two N-sided simple splinegons can be detected in $O(N \log \log N + (A_1 + B_1 + C_1)\, N)$ time.

Proof. Use the above algorithm to detect whether the boundaries of the two splinegons intersect. If they do not, we can determine whether one splinegon lies inside the other as follows. If the interiors of the two splinegons are

disjoint, then σ' must consist of repeated pairs of the form $\langle in_i, out_i \rangle$. If K_1 lies inside K_2, then σ' contains sequences of pairs of the form $\langle in_1, out_1 \rangle$ nested within pairs $\langle in_2, out_2 \rangle$. If K_2 lies inside K_1, then σ' contains sequences of pairs of the form $\langle in_2, out_2 \rangle$ nested within pairs $\langle in_1, out_1 \rangle$. These three cases exhaust the possible relationships between K_1 and K_2 since their boundaries are disjoint. □

5.6. Determining the Convex Hull of a Simple Splinegon

To extend the algorithm of Graham and Yao [GY] to compute the convex hull of piecewise-smooth Jordan curves, a subset of the simple splinegons, Schäffer and Van Wyk [SV] first revised the Graham-Yao vertex based algorithm to run as an edge-based algorithm. Thus, instead of maintaining a stack of vertices which belong to the convex hull, the Shäffer-Van Wyk algorithm maintains a stack of edges which participate in the convex hull. The main calculation on edges in the revised algorithm consists of computing a half-plane of desired orientation which contains both edges and whose bounding line supports both edges. All half-planes possessing the final two properties have bounding lines defined by endpoints of the edges. In the curved world, however, determining the desired half-plane will require computing tangents to curves. Nonetheless, this algorithm is extended directly to piecewise-smooth Jordan curves by adapting the procedures for processing edges.

5.7. Discussion

In this chapter, we have demonstrated that an edge-based polygon algorithm can be extended directly to splinegons, without using the artifice either of the carrier polygon or the bounding polygon. The generalization of the assumptions about the behavior of edges is the only modification necessary. Two edges of a polygon intersect at most in a single point, or perhaps a single line segment, whereas two splinegon edges may intersect arbitrarily often. A line supporting two edges of a polygon passes through at least one vertex of each edge, whereas a line supporting two splinegon edges may contain just one interior point from each edge. A polygon edge is always monotone in every direction but one. Some splinegon edges are not monotone in any direction, but every splinegon edge can be divided into at most three pieces such that each piece is monotone in the chosen direction. Two non-horizontal polygon edges intersect in their interiors if and only if, when the endpoints are ordered by y-coordinate, the edges intersect the horizontal lines through the middle two endpoints in different order. Even for y-monotone splinegon edges, this test proves nothing.

Revision of the assumptions about the behavior of edges does enable polygon algorithms to be extended directly to splinegons. We began the chapter by extending the Shamos algorithm for intersection computation. Next, we cited the Tarjan-Van Wyk results in horizontal visibility computation and simplicity testing, which apply both to polygons and to splinegons. In Section 5.4, we extended the polygon algorithm to decompose simple splinegons into monotone pieces. In Section 5.5, we presented a new, more efficient algorithm for detecting the intersection of simple polygons and/or splinegons. Finally, we

cited the work of Schäffer and Van Wyk in extending the Graham-Yao algorithm for computing the convex hull of a simple polygon to piecewise smooth Jordan curves.

Chapter 6

Decomposition Algorithms and the Limitations of Splinegons

6.1. Introduction

Many geometric algorithms begin by decomposing simple polygons into a disjoint set of monotone pieces, convex pieces, or triangles without adding any new vertices. Fournier and Montuno [FM] show that polygon decomposition into the union of convex polygons, of star-shaped polygons, of monotone polygons, and of triangles are all linear-time equivalent to solving the all vertex-edge visibility problem. As discussed in Chapter 5, Tarjan and Van Wyk [TV] have recently demonstrated that all horizontal-visibility information can be computed in time $O(N\log\log N)$. Thus, decomposition of a simple polygon into convex polygons, starshaped polygons, monotone polygons, and triangles can all be accomplished in $O(N\log\log N)$ time.

In general, however, splinegons do not have the same flexibility. We can determine whether a given simple splinegon is convex, star-shaped, or monotone in $O(N)$ time, as described in Chapters 3 and 4. The splinegon extensions of the Tarjan-Van Wyk algorithms allow us either to decide whether a given splinegon is simple or decompose a simple splinegon into the union of monotone pieces all having simple carriers in $O(N\log\log N)$ time, as discussed in Chapter 5. But some splinegons are inherently non-convex and can never be decomposed as the union of convex pieces. In addition, many splinegons cannot be triangulated without the addition of Steiner points.

6.2. Decomposition into Convex Pieces[8]

The problem of decomposing a simple polygon into convex pieces has received attention in various forms for several years [FP, LP1, Sc, CD1, KS2]. The motivation for this decomposition is to solve problems on more complicated general polygons by combining solutions to the problem on convex subpolygons, so decomposition into an infinite number of convex pieces is not interesting. Thus we use the term "convex decomposition" to mean a decomposition into a finite number of convex pieces.

Any polygon can be decomposed into a union of convex pieces, $\bigcup_i A_i$ [FP, LP1, Sc, CD1, KS2]. Although as mentioned above, a polygon can be decomposed into a union of convex pieces in $O(N\log\log N)$ time, convex decompositions which are optimal in some particular sense require more computation.

Decomposing a simple splinegon into the union of convex pieces, however, is problematic. A splinegon with a single edge which is concave-out can never be decomposed as a union of convex pieces. At best, such a splinegon could be expressed as the union of the convex decomposition of the carrier polygon and the union and difference, as appropriate, of the convex *S-segs*. This scheme has several problems. First of all, the *S-segs* may not be disjoint. Thus the union of the convex decomposition of the carrier polygon would have to be followed by a carefully ordered chain of unions and differences of convex *S-segs* in order to be accurate. Computing the order could be costly, the ordering prevents the parallelization of algorithms using the decomposition, and the size of the

[8] Joint work with D.P. Dobkin and C.J. Van Wyk [DSV].

decomposition must be $\Omega(N)$.

The second flaw, however, is more critical. A polygon can be decomposed as the union of convex pieces if and only if it is simple, but carrier polygons for simple splinegons need not be simple. Creating a simple carrier for a splinegon whose carrier polygon is non-simple may require quadratic time and space. We demonstrate this fact by constructing an N-sided splinegon S whose smallest simple carrier polygon has $\Omega(N^2)$ vertices.

We begin by constructing an equilateral, equiangular polygonal path C of k segments, with vertices $v_0, v_1, \ldots, v_{k-1}$, $k > 2$, such that C together with the line segment $\overline{v_0 v_{k-1}}$ bounds a convex region. Let R be the (possibly infinite) open region bounded by $\overline{v_0 v_{k-1}}$ and the lines that contain $\overline{v_0 v_1}$ and $\overline{v_{k-2} v_{k-1}}$, whose intersection with C is empty. Let h_1 be a point interior to the region bounded by C and $\overline{v_0 v_{k-1}}$ and h_2 be a point in region R; let $H = \overline{h_1 h_2}$. Let p and q be points in R such that $\overline{pv_0}$ and $\overline{qv_{k-1}}$ do not intersect H, but $\overline{pq}$ does intersect H. The following lemma implies that we can construct a splinegon edge from p to q that fits C "very tightly" in that any inscribed path must contain at least k segments:

Lemma. There exists a curve D that joins p and q such that

1) D does not intersect H,
2) $D \cup \overline{pq}$ bounds a convex region, and
3) any polygonal path inscribed in D that does not intersect $H \cup C$ contains at least k segments.

Proof. Erect perpendicular bisectors to each of the segments of C. Define points w_i on these perpendicular bisectors as follows: $w_{-1} = p$; for $0 \leq i < k-1$, let w_i be the intersection of the line through w_{i-1} and v_i and the perpendicular bisector of the edge $\overline{v_i v_{i+1}}$. Take D to be any convex curve between p and q that passes between each point w_i and the corresponding segment of C.

The perpendicular bisectors containing the points w_i define sectors with respect to the center of the curve C. The key observation to the proof of the lemma is that any inscribed path in edge D that does not cross H must have a vertex in each of these sectors: any segment with endpoints on D that are not in adjacent sectors must intersect C by the way that the points w_i were chosen. Since there are k sectors, any inscribed path in D that does not cross H has at least k vertices (see Fig. 36). □

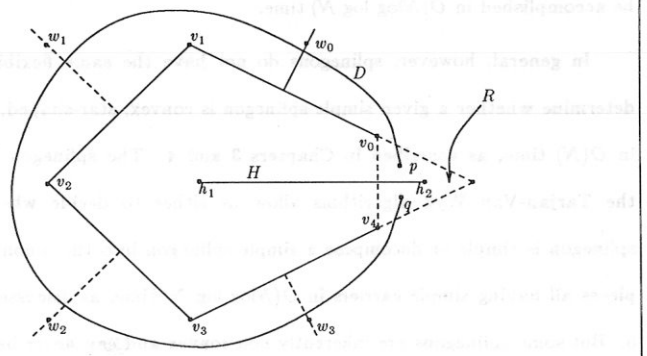

FIGURE 36. Any path inscribed in D that does not intersect either the polygonal path $v_0, \ldots, v_4$ or the line segment H must contain at least 5 segments.

Notice that any curve that lies between D and C has the same inscribed-paths property as D. The way to construct splinegon S is now clear: we pack many curved edges between C and D; each edge adds only one vertex to S, but adds at least k vertices to any simple carrier polygon.

To be more precise, put k vertices on the line segment $\overline{v_0 p}$ and k vertices on the line segment $\overline{v_{k-1} q}$. Join v_0 to h_1 and q to h_2 with line segments. Construct $2k+1$ curved edges that complete the boundary of S by joining v_{k-1} to p so that

1) the boundary of S is simple;

2) each vertex of $\overline{v_0 p}$, except for v_0, is adjacent to two vertices on on $\overline{v_{k-1} q}$;

3) each vertex of $\overline{v_{k-1} q}$, except for q, is adjacent to two vertices on on $\overline{v_0 p}$.

Splinegon S has $3k+4$ vertices. Let P be a simple carrier polygon for S. By the above lemma, P has at least k vertices on any curved edge of S, of which $k-2$ are not original vertices of S. Since S has $2k+2$ curved edges, P has at least $(2k+2)(k-2)+3k+4 = 2k^2+k$ vertices. This construction can obviously be modified to construct splinegons whose number of vertices when divided by three leaves a remainder of 0 or 2. Thus we have the following theorem:

Theorem 13. For any N, there exists a simple splinegon with a non-simple carrier polygon of N vertices for which any simple carrier polygon has $\Omega(N^2)$ vertices.

Proof. As described above. □

Given a simple splinegon with a non-simple carrier polygon, however, we have another option. By adding $O(n)$ vertices and edges, we can decompose a simple splinegon into the union of a collection of simple, monotone splinegons all defined on simple carrier polygons, as we demonstrated in Section 5.4. If convex pieces are needed, then each of these splinegons can be expressed as the union of the convex decomposition of the carrier polygon and the union and difference, as appropriate of the convex *S-segs*.

Theorem 14. A simple splinegon of N vertices can be decomposed into the union of the union and difference of a collection of convex pieces in $O(N \log\log N + (B_1 + C_1)N)$ time.

Proof. Decomposition of a simple splinegon into the union of a collection of monotone splinegons on simple carriers requires $O(N \log\log N + (B_1 + C_1)N)$ time, as demonstrated in Section 5.4. Determining a convex decomposition of a simple polygon requires $O(N \log\log N)$ time, as explained in Section 6.1. □

This approach provides *a* convex decomposition for any simple splinegon, but the size of the decomposition may be unnecessarily large. When a y-monotone splinegon S with a simple carrier polygon P is decomposed naively by forming a convex decomposition of P, then uniting it with the concave-in *S-segs*, and then subtracting the concave-out *S-segs*, the resulting decomposition has size $\Omega(N)$.

An alternative approach is to form a splinegon S' by replacing each concave-out edge of the y-monotone S by the corresponding edge of the carrier polygon. The splinegon S' can be efficiently decomposed into the smallest possi-

ble number of convex pieces, $opt(S')$.

Theorem 15. Splinegon S' can be decomposed optimally into the union of convex pieces (with or without Steiner points) using existing polygon algorithms [Gr, CD1, KS].

Proof. The existing polygon algorithms all focus on reflex angles. Thus, the edges added to decompose the splinegon S' are identical to the edges added to decompose its bounding polygon Q'. Consequently, $opt(S') = opt(Q')$. □

Given the convex decomposition of S', the concave-out *S-segs* can be subtracted from the result to give a convex decomposition of S.

Corollary. A monotone splinegon S can be decomposed into the union of the convex decomposition of S' (as defined above) and the difference of the concave-out *S-segs*. The number of pieces in the convex decomposition is $opt(S') + \nu(S)$, where $\nu(S)$ is the number of concave-out *S-segs* in S.

6.3. Triangulation

Dividing an N-sided convex polygon P into triangles is a simple linear-time procedure. By convexity, any diagonal, an open line segment joining two nonadjacent vertices, lies in the interior of P. Any collection of $N-2$ nonintersecting diagonals divides P into triangles. Triangulating a convex splinegon S is equally easy. Any collection of diagonals which triangulates P also triangulates S, and each triangle in the decomposition is convex.

Triangulation of simple splinegons, however, is complicated. The splinegonal trapezoid depicted in Figure 37a can never be triangulated merely by

adding straight edges between existing vertices. Nor will curved edges without inflection points between existing vertices suffice. The minimum-size triangulation results from stretching copies of the two longer curved edges toward each other until they touch at a point. Insert that point and the resulting four curved edges.

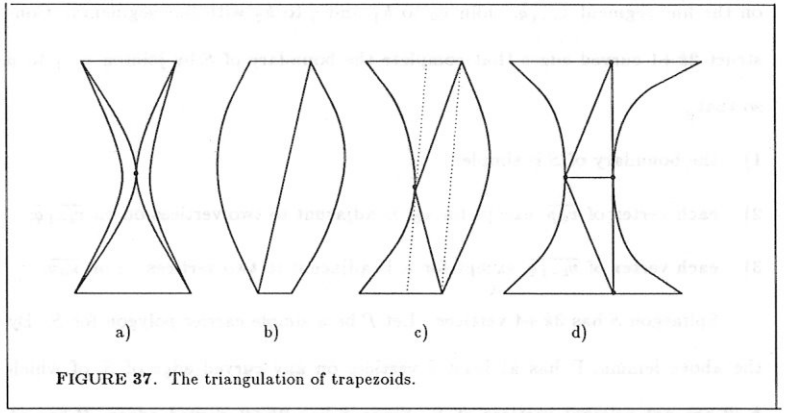

FIGURE 37. The triangulation of trapezoids.

If triangulation requires the creation of new curved edges, then its usefulness becomes questionable. Each splinegon, however, can be triangulated using a linear number of line segments and a linear number of new vertices. Computing horizontal visibility information yields a decomposition of an arbitrary splinegon into a linear number of trapezoids. Each y-notch produces at most three new vertices in the trapezoidal decomposition; the other vertices each produce at most one. A trapezoid whose side-edges are both concave-in is convex and can be triangulated by adding either diagonal; no new vertex is necessary (see Fig. 37b). If one side-edge is concave-in and one is concave-out, determine

the point on the concave-out edge which is closest to the line segment defined by the vertices of the concave-in edge; add line segments from those vertices to the new point (see Fig. 37c). If both side-edges are concave-out, add the points of minimum separation on the two curved edges as vertices. Find a line supporting one of the side-edges at its new vertex, and add the portion of that line which connects the top and bottom edge of the trapezoid. Add the two points of contact with the trapezoid as vertices; finally, triangulate the interior polygon formed by the new vertices (see Fig. 37d). At most four vertices are added to triangulate any trapezoid.

In polygon algorithms, triangulation has two major selling points: all regions can be triangulated without adding any new vertices; triangles are always convex. A prime example of the usefulness of triangulation is Kirkpatrick's optimal algorithm for planar point location [Ki].[9] He begins by triangulating every region of the given planar subdivision. A hierarchy is then established by removing an independent set of vertices and retriangulating. The efficiency of the algorithm depends on the fact that the number of vertices constantly decreases, and for each vertex discarded, the number of triangles decreases by 2. In the splinegon case, even allowing curved triangulations, there is no such guarantee.

Suppose we wish to locate a query point within a planar collection of convex objects. Using Kirkpatrick's approach, we would first determine a rectangle

[9] Lipton and Tarjan developed an optimal algorithm for this problem some years earlier [LT1, LT2]. A significant theoretical achievement, their algorithm is much harder to implement than Kirkpatrick's.

containing all of the given objects. Then we would triangulate each of the objects, as well as the region inside the rectangle and outside all of the objects. Convex splinegons can be triangulated as easily as convex polygons, but the rectangular region with holes might require a linear number of new vertices. This increase in the vertex set would be tolerable if it were a one-time expense. However, as vertices are discarded, we can still get arbitrarily complicated new regions to triangulate. The retriangulation might well add more vertices than had been discarded. Thus, this approach will not work.

Where the objects are convex, we do have another option. We can run the intersection-detection algorithm on each pair of splinegons, determining a supporting-separating line. Then using these lines, we can determine an enclosing convex polygon for each of the splinegons (see Fig. 38). Then we can run Kirkpatrick's algorithm. Once it has been determined that a point lies within a particular polygon, we can test it for inclusion in the associated splinegon. This fix works only in this very restricted case.

6.4. Limitations of splinegons

Triangulation and convex decomposition are techniques which have been used often to create efficient algorithms in the polygonal world. In the splinegon world, a convex decomposition can be obtained efficiently, provided that we allow both union and difference. The decomposition will be expressed in the form $\bigcup_j(\bigcup_i A_{ij} - \bigcup_i B_{ij})$, where j ranges over the number of monotone splinegons and the A's and B's describe the decomposition of each individual monotone

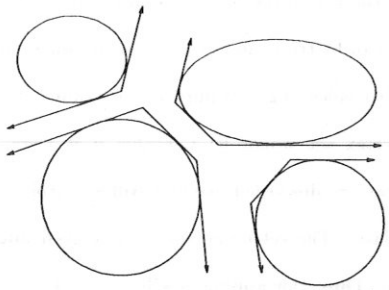

FIGURE 38. Example.

splinegon. As a linear number of vertices may be added in the process, the size of the minimum decomposition does not depend solely on the number of reflex angles. Algorithms dependent on convex decompositions have been designed to handle unions well. In many, difference can be easily accomodated. The restricted ordering of the union and difference operations, however, raises questions about the usefulness of this decomposition.

A splinegon can also be triangulated efficiently, but the triangles may not be convex, and a linear number of new vertices may be required. Algorithms dependent on triangulation often focus heavily on the convexity of the triangles. Others (e.g. [Ki]) depend on the fact that each vertex discarded decreases the number of triangles by 2. The lack of convexity and the potential size of the new decomposition may prevent the efficient extension of polygon algorithms dependent on triangulation.

Triangulation and convex decomposition represent a class of algorithms which may not extend profitably to splinegons. In the graphics world the

extension of polygon edges (resp. polyhedron faces) form a convex decomposition of the plane (resp. space). From any viewing point within one convex region, the same edges (resp. faces) are visible. The list of visible edges (resp. faces) need be updated only when the viewing point crosses a boundary. One technique used in motion planning entails unfolding polyhedral objects until they are planar. A straight path can be chosen in the plane, and then wrapped back along the surface of the polyhedron. Duality transformations which map lines to points, or planes to points have become an increasingly powerful tool within computational geometry. None of these methods extend easily into the curved world.

Does the existence of methods which do not extend to the splinegon world mean that there are a class of problems which require asymptotically more time in the splinegon world than they do in the polygonal world? Not necessarily. There is some indication that alternative methods can be substituted which allow the asymptotic complexity to remain unchanged. Both Edelsbrunner, Guibas and Stolfi [EGS] and Sarnak and Tarjan [ST] have provided optimal algorithms for planar point location which do in fact extend to splinegons. Not only do these algorithms equal the Lipton-Tarjan algorithm and the Kirkpatrick algorithm in time and space complexity, but they surpass the older algorithms in practicality. The Edelsbrunner et. al. result depends on monotone pieces rather than triangles [EGS]. The monotone decomposition of splinegons is efficient and clean. Consequently, their algorithm extends directly to splinegons, as they expect. The Sarnak-Tarjan algorithm depends solely on the monotonicity of the individual edges.

It is hard assess the degree to which alternative methods can compensate for those methods which extend poorly from the straight world to the curved world. It may be that monotone decomposition, for example, can make up for whatever power convex decomposition and triangulation will lack and thus will emerge as an increasingly powerful tool in the polygonal world as well as in the splinegonal world. Further study is needed.

Chapter 7

Splinehedra

7.1. Definition and Discussion of Features

The splinegon concept extends to three dimensions. We define a *splinehedron* S as a modification of a *carrier polyhedron* P. Each face of P is replaced by a curved surface bounded by the same vertices and edges. The i^{th} face f_i of S together with the corresponding face p_i of P must enclose a convex region $S\text{-}seg_i$. A *convex splinehedron* both encloses a convex region and has a convex carrier polyhedron.

This splinehedron model allows direct extension of the three main splinegonal methods into the three-dimensional world. The carrier polyhedron approach still works. Given a convex splinehedron S, the plane defined by the i^{th} face p_i of the carrier polyhedron P divides space into two half-spaces. The "outside" half-space contains the convex region $S\text{-}seg_i$, and the "inside" half-space contains the convex splinehedron $S_i = S - S\text{-}seg_i$. S_i can be considered a convex polyhedron which is supported by the given plane along a face. Furthermore, the convexity of S dictates that $S\text{-}seg_i$ is enclosed in the solid defined by the "outside" half-space determined by p_i of P and by the "inside" half-spaces determined by the faces adjacent to p_i. Consequently, without any direct manipulation of curved faces, the behavior of S can be reasonably approximated.

This splinehedron model readily accommodates the bounding polyhedron approach. Given an arbitrary splinehedron S, we create a bounding polyhedron

Q. Q contains all of the original vertices and all of the original edges of S, the fixed edges and fixed vertices. For each triangular face f_i of S which is not a planar polygon, let f_i^* represent the point of intersection of the three planes each of which is tangent to f_i at an edge.[10] As in the two-dimensional case, this point may have finite coordinates, or may represent a point at infinity. The pyramid defined by p_i and the pseudo-vertex f_i^* contains $S\text{-}seg_i$. Insert the pseudo-vertex f_i^* into Q along with pseudo-edges joining it to each of the fixed vertices of f_i. A face bounded only by fixed edges is called a fixed face. Otherwise, it is a pseudo-face. A pseudo-face is considered *loose* if its only intersection with the curved face it supports is the fixed edge. If the intersection has positive area, the pseudo-face is considered *tight*.

A face-based polyhedron algorithm can be extended to a face-based splinehedron algorithm using the direct approach. A face of the splinehedron resembles a face of a polyhedron in that it is bounded by a collection of vertices and line segments. In extending a polyhedron algorithm, however, all assumptions based on the flatness of the faces (e.g. the monotonicity of faces, that the intersection of two faces consists of a single component) must be updated.

[10] If f_i has more than three vertices, then the tangent planes defined by its edges may not intersect in a single point. This irregularity does not present a problem. We still insert a single pseudo-vertex f_i^* into Q, but f_i^* will represent the collection of vertices defined by the intersection of the tangent planes as well as the line segments which connect them.

7.2. Detecting the Intersection of two Convex Splinehedra using the Carrier Polyhedron Approach

We extend here the results of [DK1] to the problem of detecting whether two convex splinehedra intersect. The standard representation of a convex splinehedron resembles that of a polyhedron: a vertex list, an edge list, and a face list. The intersection-detection algorithm requires, however, that each splinehedron be represented as a sequence of parallel splinegonal cross-sections, one per vertex, and all their connecting faces and edges. Each cross-section of the splinehedron forms a splinegon having the corresponding cross-section of the carrier polyhedron as a carrier polygon. This alternative representation is uneconomical. The conversion process may require $O((E_1+E_2)n^2)$ time and the new format may require $O(n^2)$ space.[11] Since the algorithm does not presume that the two splinehedra share the same xyz-coordinate system, however, the conversion process for each splinehedron in a collection need be performed only once.

Each pair of adjacent splinegonal cross-sections and all of their connecting edges and faces describe a splinedrum whose side faces are curved patches. The carrier polygons for these adjacent splinegonal cross-sections together with their connecting edges and faces describe a carrier drum for the splinedrum (see Fig. 39). Thus a splinehedron can be viewed as a sequence of splinedrums. Each splinedrum can be specified by a circular list of its side-edges, pointers to the

[11] As mentioned in [DK1], special techniques may be applied to reduce the time and space, but they are accompanied by a time penalty for retrieving information about the individual cross-sections, faces and edges.

description of the individual curved side-faces, and the planes containing the top and bottom faces. The algorithm for detecting the intersection of two splinehedra follows that of the polyhedron algorithm of [DK1] and centers around detecting the intersection of the two middle splinedrums. In each instance in which the two splinedrums do not intersect, half of one splinehedron may be removed from future consideration.

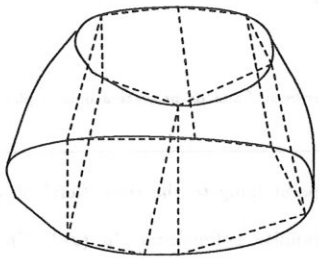

FIGURE 39. A splinedrum defined on two adjacent splinegonal cross-sections and its carrier drum defined on two adjacent polygonal cross-sections.

Before proceeding to intersection detection for two splinedrums, we begin by discussing splinegon-splinedrum intersection detection:

Theorem 16. Given a splinedrum and a splinegon, each of at most N vertices, $O((C_1+E_2)\log N + A_1 + B_1 + E_1)$ operations suffice to compute either

(a) a point common to both, or

(b) a line supporting an edge of the splinegon or a plane supporting a face of the splinedrum which separates the two objects.

Proof. Let P be a splinegon lying in the plane T, and let J be a splinedrum. Define Q as the cross-section of J determined by T. P and J intersect if and only if P and Q intersect. To determine Q explicitly would require a linear number of operations, so an implicit representation of Q is preferable. If T intersects the top (bottom) face of J, the intersection consists of a line segment, possibly degenerate, with two endpoints v_i, v_j (v_k, v_l), not necessarily distinct, lying on the boundary of the top (bottom) face of J. These endpoints can be determined in $O(\log N + E_1)$ time and all serve as vertices of the convex splinegon Q. The intersection of T with the edges between v_i and v_k (v_j and v_l) provide the remaining vertices of Q. Since each of these edges is straight, a needed vertex can be found in constant time. Each edge of Q is computed when needed in E_2 time as the intersection of T with a curved face of J (see Fig. 40). The intersection of P and Q can be detected using the splinegon-splinegon algorithm described in Section 3.3. □

To detect splinedrum-splinedrum intersections, we use an analog of the two-dimensional splinegon-splinegon intersection approach described in Section 3.3. We decompose each splinedrum into left and right semi-infinite splinedrums relative to a plane W orthogonal to the tops of both splinedrums. We assume that each splinedrum J is represented by an edge list $j_1, j_2, \ldots, j_n$ ordered counterclockwise around its top face. Denote the half-space on one side of the plane, the positive half-space; the other, the negative half-space. Consider the positive half-space to lie to the **right** of the negative half-space. The projections of the top and bottom faces of J onto W would form a pair of line segments lying on parallel lines which we shall call J_T and J_B respectively and

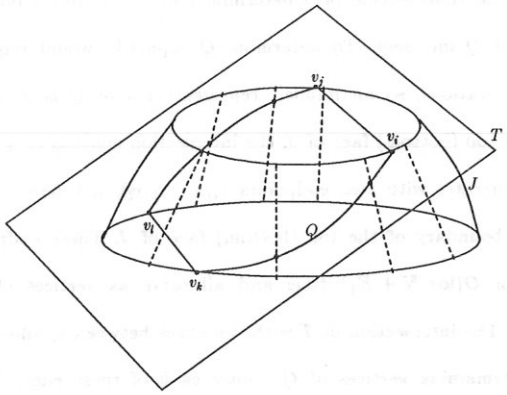

FIGURE 40. A splinedrum J intersects a plane T in a splinegon Q.

to which we assign an arbitrary orientation. The image of each j_i would be a line segment joining the lines J_T and J_B. Locate the edge j_k (j_l) with the maximum (minimum) projection relative to the orientation of J_T and J_B. Due to the convexity of J, this task requires only logarithmic time. These edges may not represent the side boundaries, however, of the projection of J onto W (see Fig. 41). The face on either side of j_k (resp. j_l) may have a curve whose projection forms the maximum (minimum) boundary. Determine which face, if any, has this characteristic and insert a new pseudo-edge j_M (resp. j_m) to represent that curve into the circular list for J. We do not compute the new pseudo-edge explicitly, as it may even represent a non-planar curve.

The splinedrum J can now be split at the edges j_m and j_M. The edges from j_m to j_M in clockwise (counter-clockwise) order, the curved faces which

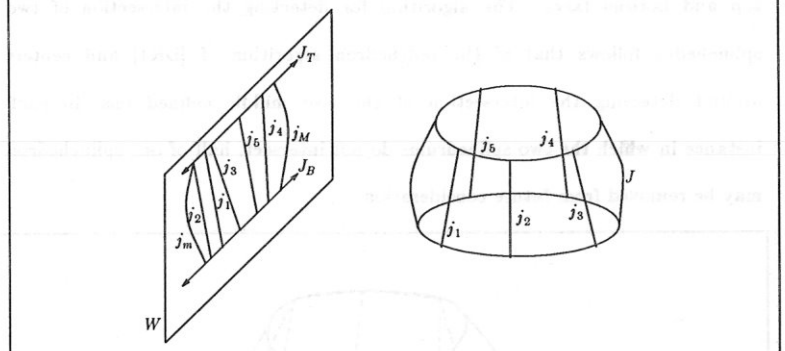

FIGURE 41. A splinedrum J and its projection onto a plane W orthogonal to the top of J.

join them, and every point lying to the right (left) of some face together form the right (left) semi-infinite splinedrum J_R (J_L). The boundary of J_R (J_L) includes the edges and faces defined above as well as infinite end-faces extending from j_m and j_M to $+\infty$ ($-\infty$) and infinite top and bottom faces defined similarly. This decomposition requires $O(C_1 + \log N)$ time. The decomposition of a splinedrum J into semi-infinite splinedrums J_R and J_L satisfies the following conditions: $J = J_R \cap J_L$ and $J \subseteq J_L, J_R$. Consequently, two splinedrums J and K intersect if and only if J_R intersects K_L and J_L also intersects K_R.

Let L (resp. R) be a semi-infinite splinedrum opening toward the left (resp. right) half-space defined by W and having its top and bottom faces contained in the respective planes L_T and L_B (R_T and R_B). Any potential intersection of L and R is confined to the infinite parallelepiped defined by the four planes L_B, R_B, L_T, and R_T. Consequently, we restrict our search to that region. The first

step of the algorithm consists of deleting all faces of R lying strictly below L_B and strictly above L_T and all faces of L lying strictly below R_B and strictly above R_T. Assume that L (resp. R) has n (resp. m) remaining side-edges $l_1, l_2, \ldots, l_n$ ($r_1, r_2, \ldots, r_m$) and that the middle edges of L and R are l_j and r_i. Let R_i (resp. L_j) represent the plane defined by the pair of segments r_i, r_{i+1} (resp. l_j, l_{j+1}). The planes R_i and L_j divide the parallelepiped into three or four regions comparable to those defined in the two-dimensional case: LR-region, L-region, R-region, and $\varnothing$-region. If R_i intersects L_j in a line, the projection of that line onto W is labelled W_{ij}.

In what follows, we say that the face $r_i r_{i+1}$ (resp. edge r_i) **overlaps** the face $l_j l_{j+1}$ (resp. edge l_j) if their projections onto W intersect. We say that a face (resp. an edge) **borders** the LR-region if some portion of it within the parallelepiped defined by L_B, R_B, L_T, and R_T does. If no pair of edges, one from r_i, r_{i+1} and one from l_j, l_{j+1}, overlap, then the closest pair form the **inner** edges; the remaining two are **outer** edges. In cases where $r_i r_{i+1}$ overlaps $l_j l_{j+1}$ but only in the $\varnothing$-region, we call the pair of edges, one from r_i, r_{i+1} and one from l_j, l_{j+1}, whose projections intersect nearest to the line W_{ij}, the **near** edges; the remaining two are **far** edges. The edges **beyond** r_i (resp. r_{i+1}) consist of $r_1, \ldots, r_{i-1}$ (resp. $r_{i+2}, \ldots, r_m$). The terms **overlap** and **beyond** replace the terms **separate** and **above** or **below** in the two-dimensional algorithm.

Lemma 1. If $l_j l_{j+1}$ overlaps $r_i r_{i+1}$ in the LR-region, then in constant time we can determine either a point l on the boundary of $l_j l_{j+1}$ or a point r on the boundary of $r_i r_{i+1}$ which is a witness to the intersection of R and L.

Proof. The projections of the faces $l_j l_{j+1}$ and $r_i r_{i+1}$ onto the plane W are trapezoids. Calculate the intersection of this pair of trapezoids in constant time, and determine a point on the boundary of one of the trapezoids, say the trapezoid associated with $r_i r_{i+1}$, which belongs to the intersection. Find the intersection of the line through that point and orthogonal to W with the boundary of $r_i r_{i+1}$. The point reported lies directly to the left of $l_j l_{j+1}$ and thus serves as a witness to the intersection of L and R (see Fig. 42). □

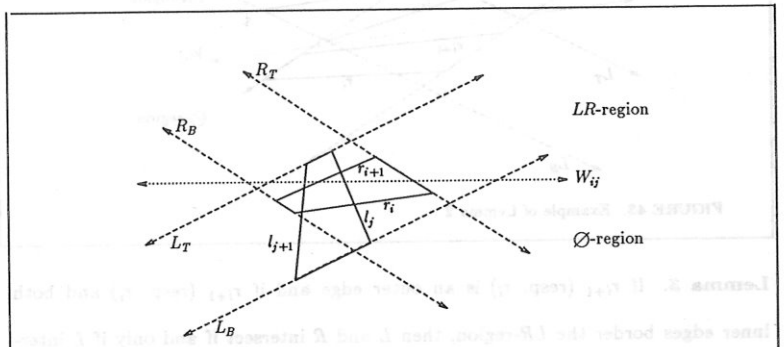

FIGURE 42. Example of Lemma 1.

Lemmas 2-6 below state results in terms of R and its edges where a symmetric result can be stated for L and its edges.

Lemma 2. If r_i (resp. r_{i+1}) is an outer edge and if neither r_i nor any edge beyond r_i (resp. r_{i+1}) borders the LR-region, then L and R intersect if and only if L intersects the splinedrum formed by deleting all edges of R beyond r_i (resp. r_{i+1}).

Proof. If L and R intersect, either they intersect in the LR-region or L-seg_j

(resp. $R\text{-}seg_i$) intersects R (resp. L). No part of R beyond r_i borders the LR-region, nor can $L\text{-}seg_j$ intersect any part of R beyond r_i. Consequently, all edges of R beyond r_i can be removed from further consideration (see Fig. 43). □

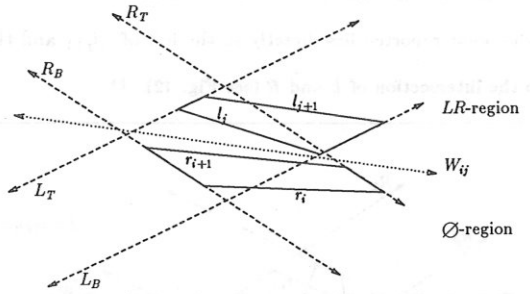

FIGURE 43. Example of Lemma 2.

Lemma 3. If r_{i+1} (resp. r_i) is an outer edge and if r_{i+1} (resp. r_i) and both inner edges border the LR-region, then L and R intersect if and only if L intersects the splinedrum formed by deleting all edges of R beyond r_{i+1} (resp. r_i).

Proof. L may indeed intersect R beyond r_{i+1}. If it does, however, it must also intersect the semi-infinite rectangle extending from r_{i+1} to $+\infty$ which lies within R. Thus the portion of R lying strictly beyond r_{i+1} cannot completely contain the intersection of L and R. As we are searching for a single point witnessing the intersection rather than the entire intersection area, the portion of R beyond r_{i+1} can be deleted (see Fig. 44). □

Lemma 4. If l_jl_{j+1} overlaps r_ir_{i+1} only in the ∅-region, if l_jl_{j+1} borders the LR-region, and if r_i (resp. r_{i+1}) is a far edge, then L intersects R if and only if L

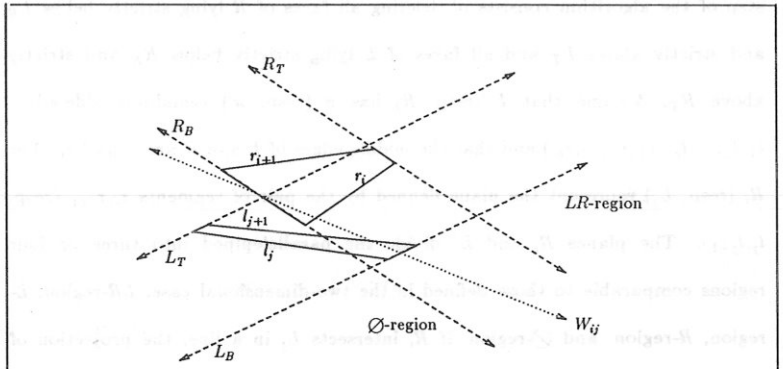

FIGURE 44. Example of Lemma 3.

intersects the splinedrum formed by deleting all edges of R beyond r_i (resp. r_{i+1}).

Proof. L intersects R beyond r_i (resp. r_{i+1}) if and only if $L\text{-}seg_j$ penetrates the R-region. If so, then L intersects the semi-infinite rectangle extending from r_i (resp. r_{i+1}) to $+\infty$ which lies within R (see Fig. 45). □

Lemma 5. If l_jl_{j+1} overlaps r_ir_{i+1} only in the ∅-region, and neither l_jl_{j+1} nor r_ir_{i+1} border the LR-region, let l^i represent a point on the face l_jl_{j+1} where a plane parallel to R_i supports the portion of the face l_jl_{j+1} which lies within the parallelepiped. If l^i lies within the R-region and beyond the far edge r_i (resp. r_{i+1}), then L intersects R if and only if L intersects the splinedrum formed by deleting all edges of R beyond the near edge r_{i+1} (resp. r_i). If l^i lies within the R-region and between r_i and r_{i+1}, then l^i is a witness to the intersection of L and R. Otherwise, L intersects R if and only if L intersects the splinedrum formed by deleting all edges of R beyond the far edge r_i (resp. r_{i+1}).

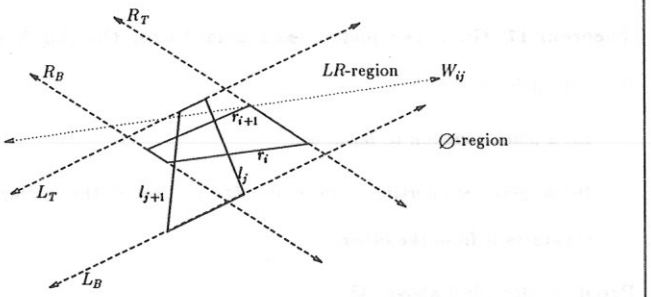

FIGURE 45. Example of Lemma 4.

Proof. If l_jl_{j+1} overlaps r_ir_{i+1} in the $\emptyset$-region, then $L\text{-}seg_j$ could intersect $R\text{-}seg_i$ or could intersect R beyond r_i or could intersect R beyond r_{i+1}. In addition, the portions of L and R beyond the near edges could intersect in the LR-region. Thus no part of either splinedrum may be deleted immediately. We compute l^i, the extreme point on L in the direction orthogonal to R_i. If l^i lies in the R-region beyond the far edge r_i (resp. r_{i+1}), then L intersects R beyond r_{i+1} (resp. r_i) only if L also intersects the semi-infinite rectangle extending from r_{i+1} (resp. r_i) to $+\infty$ which lies within R. If l^i lies in the R-region between r_i and r_{i+1}, then l^i lies directly to the right of a face of R and is thus a witness to the intersection of L and R. If l^i does not lie in the R-region, then L does not intersect R beyond the far edge r_i (resp. r_{i+1}). If l^i lies in the R-region beyond the near edge r_{i+1} (resp. r_i), then L intersects R beyond r_i (resp. r_{i+1}) only if L also intersects the semi-infinite rectangle extending from r_i (resp. r_{i+1}) to $+\infty$ which lies within R (see Fig. 46). □

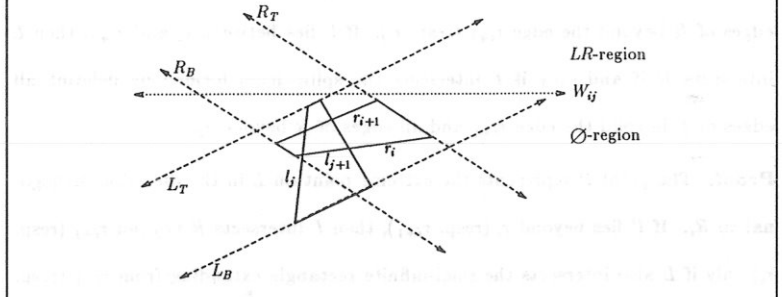

FIGURE 46. Example of Lemma 5.

We apply Lemmas 1-5 and their symmetric versions repeatedly until either a point in the intersection has been determined or until one semi-infinite drum, say L, has only 3 edges. In the former case, we are done. In the latter case, decompose L into two left semi-infinite splinedrums of only one face and two edges each.

The problem remaining is to detect the intersection of a two-edge left semi-infinite splinedrum L with a right semi-infinite splinedrum R of m edges. It is possible, however, that the single face of L may be a special face created by the process of splitting a finite splinedrum into two semi-infinite splinedrums. Thus either l_1 or l_2 may not have been computed explicitly and may even represent a non-planar curve. As a result, the plane L_1 may not be well-defined. Consequently, we apply a new lemma in order to reduce the size of R:

Lemma 6. If L has precisely two edges, let l^i represent a point on the face l_jl_{j+1} where a plane parallel to R_i supports the portion of the face l_jl_{j+1} which lies within the parallelepiped. If l^i lies beyond the edge r_i (resp. r_{i+1}), then L

intersects R if and only if L intersects the splinedrum formed by deleting all edges of R beyond the edge r_{i+1} (resp. r_i). If l^i lies between r_i and r_{i+1}, then L intersects R if and only if L intersects the splinedrum formed by deleting all edges of R beyond the edge r_{i+1} and all edges of R beyond r_i.

Proof. The point l^i represents the extreme point on L in the direction orthogonal to R_i. If l^i lies beyond r_i (resp. r_{i+1}), then L intersects R beyond r_{i+1} (resp. r_i) only if L also intersects the semi-infinite rectangle extending from r_{i+1} (resp. r_i) to $+\infty$ which lies within R. If l^i lies between r_i and r_{i+1}, then either $L\text{-}seg_1$ intersects $R\text{-}seg_i$ (or the flat face $r_i r_{i+1}$ if $R\text{-}seg_i$ is null) or L and R do not intersect (see Fig. 47). □

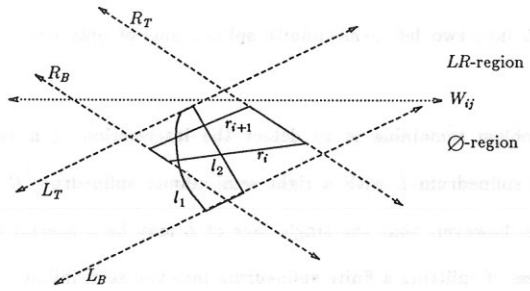

FIGURE 47. Example of Lemma 6.

Repeated iterations of Lemma 6 will reduce the second splinedrum to at most 3 edges. A constant number of oracle calls and constant additional time will detect any intersection among the remaining splinedrums.

The algorithm is repeated twice: once for J_R and K_L; and once for J_L and K_R. The results are merged, as in the two-dimensional case.

Theorem 17. Given two preprocessed splinedrums, $O(C_2 \log N + A_2)$ operations can determine either

(a) a point common to both, or

(b) a plane supporting a face or edge of one of the splinedrums which separates it from the other.

Proof. As described above. □

The algorithm described above for the splinedrum-splinedrum intersection problem must be expanded for the splinehedron-splinehedron intersection problem. We use the notation of [DK1] and call the middle splinedrum of a splinehedron its *waist*. In addition, the *cone* of a splinedrum J of a splinehedron P is formed by extending infinitely the trapezoid defined by the two side edges of each face and determining the intersection of all of the half-spaces H_i which each contain all of J except for a curved $J\text{-}seg_i$. The cone of a splinedrum is the largest convex polyhedron which contains the straight-edged carrier of the splinedrum as a cross-section. The cone of J includes all of P except for the P-*segs* which participate in J. Finally, we define a new term. The *ball* of a splinedrum J of a splinehedron P is the union of all of the half-spaces containing J defined by planes which support both a face or side-edge of J and the corresponding face or edge of P. In general, there are an infinite number of such half-spaces. Consequently, we do not compute the *ball* of J, but it remains a useful abstract object. The ball of J is the largest convex splinehedron which contains J as a cross-section. The ball of J includes P.

Theorem 18. The intersection of two preprocessed convex splinehedra of at most N vertices each can be detected in $O((A_2+ B_1) \log N + (C_2+ E_2) \log^2 N)$ operations.

Proof. Let P and Q represent the two preprocessed convex splinehedra. Let J (K) be the waist of P (Q); let C_J (C_K) represent the cone of J (K); and let B_J (B_K) represent the ball of J (K). Determine whether J and K intersect by following the algorithm of Theorem 17 and using $O((C_2) \log N + A_2)$ operations. If they intersect, the point returned by the algorithm is also a witness to the intersection of P and Q. If not, the algorithm returns a supporting separating plane. If possible, it chooses a plane T which separates J from K by supporting both a face or side-edge of J (resp. K) and the corresponding face or edge of P (resp. Q). Otherwise, the algorithms reports a plane T which supports a top or bottom face of J (resp. K).

Suppose that T supports a face or a side-edge of J. Then T also separates P from K. Using $O(\log N)$ time, we can determine whether T intersects C_K above K, below K, or not at all. If no intersection is found, then $O(E_2)$ time suffices to determine whether T intersects some Q-seg_j above K, below K, or not at all. If still no intersection is found, then T separates the two splinehedra. Otherwise, either the bottom half or the top half of Q may be removed from future consideration.

Suppose that T supports the top (resp. bottom) face of J, and thus K lies "above" ("below") J. Since T supports the top (resp. bottom) face of J, no plane supporting a face or side-edge of J separates it from K. Thus, B_J must

intersect K above (resp. below) J, say at some point z. Suppose that Q intersects P at a point w which lies below (resp. above) J. By convexity, the line segment $\overline{wz}$ must belong to Q. But since $P \subseteq B_J$, $\overline{wz}$ also belongs to B_J. Consequently, $\overline{wz}$ must intersect J and thus Q intersects J. Thus P and Q intersect if and only if Q intersects the splinehedron formed by deleting all splinedrums of P lying below (resp. above) J.

In either instance, half of the splinedrums of one splinehedron can be removed from future consideration after $O((C_2+ E_2)\log N + A_2+ B_1)$ operations. □

7.3. An Alternative Splinehedron Model and its Features

A splinehedron is defined from its carrier polyhedron. The vertex list remains unchanged. Each face entry is modified to contain an equation of the surface in which the face lies. In the general case, each edge entry must be modified to include an equation of the planar curve which joins its two vertex-endpoints and which separates the two faces.[12] The i^{th} face of the splinehedron S together with the corresponding face of the carrier polyhedron P and each plane defined by a curved edge of S and the corresponding straight edge of P bound a convex region S-seg_i.

Allowing each face of P to be replaced by a curved surface containing the vertices of the original face but having curved edges more adequately reflects the real world. This model, however, dramatically alters the efficacy of the three

[12] We restrict an edge to being a planar curve because a non-planar edge would dramatically complicate the model.

methods of polyhedron extension. A splinehedron in this model still has a carrier polyhedron, but it no longer approximates the splinehedron as well as it did in the restricted model. Given a convex splinehedron S, the plane defined by the i^{th} face p_i of its carrier polyhedron P divides space into two half-spaces. The "outside" half-space contains the convex region $S\text{-}seg_i$, but also some portion of each of the adjacent S-*seg*s and possibly some part of an unlimited number of other neighboring S-*seg*s. The "inside" half-space contains the remainder, a convex splinehedron which cannot be defined explicitly without specific calculations for each instance of S and i. In addition, the solid defined by the "outside" half-space determined by p_i of P and by the "inside" half-spaces determined by the faces adjacent to p_i no longer contains $S\text{-}seg_i$. Whenever f_i represents a curved face all of whose edges are curved, then each of those edges as well as a neighboring region will be excluded from the solid.

The bounding polyhedron approach is even more problematical. A bounding polyhedron should contain all of the original vertices of a splinehedron as well as a collection of pseudo-vertices which approximate the faces. The previous method of defining the bounding polyhedron is no longer valid. Once edges are defined as curves, there no longer exists a single plane which supports a face along an entire edge, approximating the face in the neighborhood of that edge. Only one alternative method seems promising. For each triangular face f_i of S which is not a planar polygon, let f_i^* represent the point of intersection of the three planes each of which is tangent to f_i at a vertex. The pseudo-vertices of the form f_i^* only form a subset of the new vertices which must be inserted into Q. Neighboring pyramids will intersect each other forming numerous new

vertices and edges. The bounding polyhedron defined in this fashion could have as many vertices as the sum of the number of faces of S with three times the number of vertices.

The direct approach is also adversely affected, but not to the same extent. A face of a splinehedron under this model has a smaller resemblance to a face of a polyhedron. Not only is the face not flat, it also does not have a planar boundary, let alone a piecewise-straight planar boundary. The modifications which must be made to a polyhedron algorithm become far more complicated.

7.4. The Effect of the Alternative Model on Intersection Detection

The process for converting a splinehedron into a sequence of splinedrums is more complicated in the alternative model. In the original model, each cross-section of the splinehedron formed a splinegon whose carrier polygon was the corresponding cross-section of the carrier polyhedron. Each adjacent pair of splinegonal cross-sections together with their carrier polygons defined a splinedrum and its carrier drum. In the alternative model, a cross-section of the polyhedron no longer serves as a carrier for a cross-section of the splinehedron (see Fig. 48a). The carrier polygon for each individual cross-section is determined by explicitly computing the vertices of the intersection of the cutting plane with the curved edges of the splinehedron (see Fig. 48b). To achieve a candidate carrier drum, two vertices of adjacent carrier polygons are connected by a straight edge if the same points are connected by a curved edge in the splinehedron. But the four vertices delimiting a face, two points from each carrier polygon, may not be coplanar. In general, diagonal edges must be

added to yield a convex carrier drum with triangular faces.

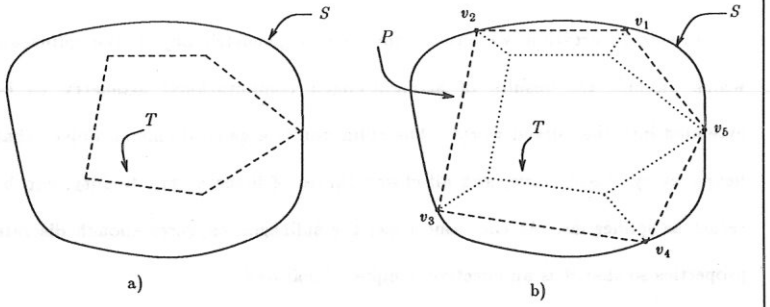

FIGURE 48. a) The cross-sections S and T of a splinehedron and its carrier polyhedron, respectively; b) The carrier polygon P for the splinegonal cross-section S.

Thus, in the alternative model, each splinehedron may still be viewed as a sequence of splinedrums. These splinedrums, however, have different features. The side faces of the splinedrums are triangular curved patches and the top and bottom faces are splinegons. The side edges of the splinedrums will be planar curves rather than line segments. Consequently, many of the presumptions in the original algorithm do not hold.

Again, let L (resp. R) represent a semi-infinite splinedrum opening toward the left (resp. right) half-space defined by W and having its top and bottom faces contained in the respective planes L_T and L_B (resp. R_T and R_B). Here, let $l_1, l_2, \ldots, l_n$ (resp. $r_1, r_2, \cdots, r_m$) represent the side-edges of the carrier drum for L (resp. R). Let L_j (resp. R_i) represent the plane defined by the edges l_j, l_{j+1} (resp. r_i, r_{i+1}). In the previous model, L_j (resp. R_i) divided the world into two half-spaces: one containing $L\text{-}seg_j$ (resp. $R\text{-}seg_i$), and the other containing

$L - L\text{-}seg_j$ (resp. $R - R\text{-}seg_i$). In this model, one half-space contains $L\text{-}seg_j$ (resp. $R\text{-}seg_i$) as well as portions of some number of contiguous L-segs (resp. R-segs) on either side. It may require $O(\log N)$ time to determine the full range of the L-segs (resp. R-segs) involved. Thus, the four regions defined by R_i and L_j are far less useful than in the previous model, and it remains unclear how best to define the inner loop of the algorithm to reduce the size of each semi-infinite splinedrum. In any case, the new algorithm should be far more complicated than the original and run much more slowly. Given the added complications of this alternative model, we have chosen to focus on the more restricted model defined in Section 7.1.

7.5. Summary and Open Questions

In this chapter, we have defined a geometric object, the splinehedron. Under the definition given in Section 7.1, each of the approaches for extending polygonal algorithms to splinegons also applies to extending polyhedron algorithms to splinehedra. We use the carrier polyhedron approach to prove that the intersection of two preprocessed convex splinehedra of at most N vertices each can be detected in time $O((A_2 + B_1) \log N + (C_2 + E_2) \log^2 N)$.

As computation in three dimensions is far more difficult than the corresponding computation in two dimensions, far fewer three-dimensional algorithms have been developed. As more polyhedron algorithms are developed, however, analogs of Chapters 3, 4, and 5 of this thesis can be written, extending those new algorithms to splinehedra. Alternately, generators of new algorithms might ponder the extension principles presented here and develop directly

general algorithms applicable both to polyhedra and splinehedra.

The definition of a splinehedron which we have currently chosen, however, restricts us to a "soccer ball" version of the world: every face of a three-dimensional object is bounded by a planar polygon composed of a collection of straight line segments. Although certainly more general than a polyhedron itself, this splinehedron is hardly as general a model for the three-dimensional world as the splinegon is for two dimensions. Even the alternative definition which we suggest here provides at best an awkward framework in which to model real-world objects which have saddle points. The question remains how best to modify the definition of a splinehedron to enlarge the class of objects for which it is an effective model but also to retain an adaptability for computation.

Chapter 8

Conclusions and Future Work

In this dissertation, we have defined a new geometric object, the *splinegon*, which enables the results of straight-edged computational geometry to be extended into the curved world. The splinegon is a general enough object that nearly every closed curve, and all closed curves of inherent practicality, can be recast as splinegons. At the same time, the splinegon captures enough discrete properties so that it is an effective computational tool.

We have presented three distinct techniques for extending polygon algorithms to splinegons: the carrier polygon approach, the bounding polygon approach, and the direct approach. The carrier polygon is a polygon whose edges are all chords of the splinegon, often yielding sufficient information about the behavior of the splinegon that direct curvilinear computations can be reduced; specific information about the behavior of the splinegon is computed only when necessary. The carrier polygon imposes sufficient structure on a convex splinegon that polygon algorithms can be extended to splinegons with the only modification being *ad hoc* procedures for the accommodation of the *S-segs*.

The bounding polygon is a polygon each of whose edges is tangent to an edge of the splinegon. The bounding polygon approximates the contour of a splinegon better than does the carrier polygon, and thus is more versatile. When computed explicitly, it provides a good approximation for the splinegon. Without actually computing its edges and vertices, however, it can still allow a vertex-based polygon algorithm to be extended to splinegons with its structure

intact. It also allows the creators of new algorithms to write them in a general format which encompasses splinegons, but with the more restricted polygon algorithm clearly visible within. In either case, separate procedures exist for the processing of fixed vertices and for the processing of pseudo-vertices.

TABLE OF RESULTS

Problem	Approach	Time Complexity
Intersection detection of a line with a convex splinegon	Carrier	$O(B_1 + \log N)$
Intersection detection of two convex splinegons	Carrier	$O(A_1 + B_1 + C_1 \log N)$
Intersection detection of convex splinegon and splinedrum	Carrier	$O((C_1 + E_2)\log N + A_1 + B_1 + E_1)$
Intersection detection of two convex splinedrums	Carrier	$O(C_2 \log N + A_2)$
Intersection detection of two convex splinehedra	Carrier	$O((A_2 + B_1)\log N + (C_2 + E_2)\log^2 N)$
Testing point inclusion for a convex polygon using hierarchy	Carrier	$O(\log N)$
Testing point inclusion for a convex splinegon	Carrier	$O(B_1 + \log N)$
Area computation for an arbitrary splinegon	Carrier	$O(F_1 N)$
Diameter computation for a convex splinegon	Bounding	$O((A_1 + C_1) N)$
Monotonicity determination for a simple splinegon	Bounding	$O(C_1 N)$
Kernel computation for a simple splinegon	Bounding	$O((B_1 + C_1)N)$
Convex hull computation for a simple splinegon	Bounding	$O((B_2 + C_1 + D_1)N)$
Intersection computation for a convex splinegon	Direct	$O(A_1 N + C_1 \log N + B_1)$
Horizontal visibility computation for a simple splinegon [TV]	Direct	$O(N\log\log N + (B_1 + C_1)N)$
Simplicity testing for an arbitrary splinegon [TV]	Direct	$O(N\log\log N + (A_1 + B_1 + C_1)N)$
Monotone decomposition of a simple splinegon [DSV]	Direct	$O(N\log\log N + (B_1 + C_1)N)$
Intersection detection for two simple splinegons [DSV]	Direct	$O(N\log\log N + (A_1 + B_1 + C_1)N)$

Often edge-based algorithms can be translated to splinegons merely by updating the assumptions about possible behavior of edges. We call this type of extension the direct approach. All three types of extension presented allow polygon algorithms to be extended to splinegon algorithms of the same asymptotic time complexity, except for the more complicated primitive procedures (see Table).

Despite the success of these three methods, some tools of straight-line geometry project poorly into the curvilinear world. As shown in Chapter 6, whether either convex decomposition or triangulation of splinegons can be used to produce efficient algorithms remains uncertain. Splinegons can be efficiently decomposed into convex pieces, but only into the union and difference of convex pieces under a restrictive ordering. Splinegons can also be triangulated, but the process may require a linear number of new vertices and the resulting triangles may not be convex.

Triangulation and convex decomposition represent a class of algorithms which may not extend profitably to splinegons. In graphics, one often extends object edges or faces to form a convex decomposition of the plane or of space, respectively. In motion planning, one may unfold a two or three dimensional object until it is linear or planar. Neither of these techniques extend readily to curved objects. Likewise, duality transformations which map lines to points, or planes to points, have no obvious counterpart in the curved world.

Although in the planar point location example, monotonicity proved to be an equally powerful tool as triangulation, it is hard to assess the degree to which alternative methods can compensate for those methods which extend poorly from the straight world to the curvilinear world. Further study is needed.

Finally, in Chapter 7, we define and describe three-dimensional curved

geometric objects, named *splinehedra*. It is interesting to note that while the splinegon emerged as the natural generalization of the polygon, there are two natural generalizations of the polyhedron which could be chosen as the definition of a splinehedron. One serves as a better model of real objects, while the other has more computational power. Neither is capable of modelling three-dimensional objects with saddle points. More work in three dimensions is needed.

This dissertation differs from the standard model in that the major contribution is a collection of methods, rather than a few distinct results. As such, this contribution should have ongoing impact, allowing algorithm designers of the future to extend their new results easily to the curved world.

REFERENCES

[BO] Bentley, J. and Ottmann,T., Algorithms for reporting and counting geometric intersections, Carnegie-Mellon University, August 1978.

[Br] Brown, K. Q., Geometric transforms for fast geometric algorithms. Rep. CMU-CS-80-101, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1980.

[CD1] Chazelle, B., and Dobkin, D., Optimal convex decompositions, *Machine Intelligence and Pattern Recognition 2: Computational Geometry,* G.T. Toussaint, ed., Elsevier Science Publishers, North Holland, 1985, pp. 63-133.

[CD2] Chazelle, B., and Dobkin, D., Intersection of convex objects in two and three dimensions, *Journal of the ACM,* to appear. A preliminary version of this paper, entitled "Detection is easier than computation," appeared in the proceedings of the ACM Symposium on Theory of Computing, Los Angeles, Ca, May, 1980, 146-153.

[CI] Chazelle, B., and Incerpi, J., Triangulation and shape-complexity, *ACM Transactions on Graphics, 3, 1984, pp. 135-52.*

[DE] Dobkin, D. and Edelsbrunner, H. Space searching for intersecting objects, IEEE FOCS, Singer Island, Fla, October, 1984.

[DK1] Dobkin, D. and Kirkpatrick, D., Fast detection of polyhedral intersections, *Theoretical Computer Science*, 27, 1983, 241-253.

[DK2] Dobkin, D. and Kirkpatrick, D., A linear algorithm for determining the separation of convex polyhedra, *Journal of Algorithms*, 6, 1985, 381-92.

[DM] Dobkin, D. and Munro, J. I., Efficient uses of the past, *Journal of Algorithms,* 6, 1985, 455-65.

[DS] Dobkin, D. and Souvaine, D., Computational Geometry -- A User's Guide, *Advances in Robotics 1: Algorithmic and Geometry Aspects of Robotics*, J. T. Schwartz and C. K. Yap, eds., Lawrence Erlbaum Associates, 1986, pp. 43-93.

[DSV] Dobkin, D., Souvaine, D., and Van Wyk, C., Decomposition and intersection of simple splinegons, submitted for publication.

[EGS] Edelsbrunner, H., Guibas, L. J., and Stolfi, J., Optimal point location in a monotone subdivision, DEC System Research System Report, October, 1984.

[FP] Feng, H. and Pavlidis, T., Decomposition of polygons into simpler components: feature generation for syntactic pattern recognition, *IEEE Transactions on Computing* C-24, 1975, pp.636-50.

[FvD] Foley, J. D. and van Dam, A., *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, MA, 1982.

[Fo] Forrest, A.R., Invited talk on computational geometry and software engineering, Second Annual Symposium on Computational Geometry, Yorktown Heights, New York, June, 1986.

[FM] Fournier, A. and Montuno, D. Y., Triangulating simple polygons and equivalent problems, *ACM Transactions on Graphics*, 3, 1984, 153-74.

[GY] Graham, R. L. and Yao, F. F., Finding the convex hull of a simple polygon, *Journal of Algorithms*, 4, 1983, 324-31.

[Gr] Greene, D. H., The decomposition of polygons into convex parts, *Advances in Computing Research*, F. Preparata, ed., JAI Press, 1984, pp.235-59.

[He] Hershberger, J., private communication.

[HMRT] Hoffman, K., Melhorn, K., Rosensthiehl, P., and Tarjan, R., Sorting Jordan sequences in linear time using level-linked search trees, *Information and Control, 68, 1986, 170-84*.

[HK] Hopcroft, J. and Kraft, D., The challenge of robotics, *Advances in Robotics 1: Algorithmic and Geometry Aspects of Robotics*, J. T. Schwartz and C. K. Yap, eds., Lawrence Erlbaum Associates, 1986.

[KS1] Kedem, K. and Sharir, M., An efficient algorithm for planning collision-free translational motion of a convex polygonal object in 2-dimensional space amidst polygonal obstacles, Computational Geometry Conference, Baltimore, Md., June, 1985, pp. 75-80.

[Ke] Keil, J. M., Decomposing a polygon into simpler components, *SIAM Journal of Computing*, 14, 1985, pp. 799-817.

[KS2] Keil, J. M. and Sack, J. R., Minimum decompositions of polygonal objects, *Machine Intelligence and Pattern Recognition 2: Computational Geometry*, G. T. Toussaint, ed., Elsevier Science Publishers, North Holland, 1985, pp. 197-216.

[Ki] Kirkpatrick, D., Optimal search in planar subdivisions, *Siam Journal on Computing* 12, 1, 1983, pp. 28-35.

[Kn] Knuth, D., *Computers and Typesetting, Vol. C: The METAFONTbook*, Addison-Wesley, Reading, MA, 1986.

[LP1] Lee, D. T. and Preparata, F., Location of a point in a planar subdivision and its applications, *SIAM Journal of Computing*, 6, 3, 1977, pp. 594-606.

[LP2] Lee, D. T. and Preparata, F., An optimal algorithms for finding the kernel of a polygon, *Journal of the ACM*, 26, 1979, 415-21.

[LP3] Lee, D. T. and Preparata, F., Computational geometry--a survey, *IEEE Transactions on Computers*, C-33, 1984, pp. 1072-1101.

[LT1] Lipton, R. and Tarjan, R., A separator theorem for planar graphs, *SIAM Journal of Applied Mathematics* 36, 1979, 177-89. A preliminary version of this paper was presented at the Waterloo conference on theoretical computer science, Waterloo, Ontario, August, 1977.

[LT2] Lipton, R. and Tarjan, R., Applications of a planar separator theorem, IEEE FOCS Conference, Providence, Rhode Island, October, 1977, pp. 162-170.

[Lo] Lozano-Perez, T., Invited talk on robotics, Second Annual Symposium on Computational Geometry, Yorktown Heights, New York, June, 1986.

[O] Overmars, M. H., Searching in the past, Parts I and II, University of Utrecht Technical Reports, 1981.

[OvL] Overmars, M. H., and van Leeuwen, J., Maintenance of configurations in the plane, *JCSS*, 23, 2, 1981, pp. 166-204.

[Pa] Pavlidis, T., Curve fitting with conic splines, *ACM Transactions on Graphics*, 2,1,1985, pp.1-31.

[Pr] Pratt, V., Techniques for conic splines, *Computer Graphics*, 19,3, 1985, pp. 151-9.

[PS1] Preparata, F. and Shamos, M. I., *Computational Geometry: An Introduction*, Springer-Verlag, 1985.

[PS2] Preparata, F. and Supowit, K., Testing a simple polygon for monotonicity, *Information Processing Letters*, 12, 4, 1981, pp. 161-64.

[Re] Requicha, A., Representations for rigid solids: theory, methods and systems, *Computing Surveys*, 12,4,1980, pp. 437-464.

[ST] Sarnak, N. and Tarjan, R. E., Planar point location using persistent search trees, *Communications of the ACM*, 29, 1986, 669-79.

[SB] Saxe, J. B. and Bentley, J. L., Decomposable searching problems, I. Static-to-dynamic transformation, *Journal of Algorithms*, 1, 4, 1980, pp. 301-358.

[Sc] Schacter, B. Decomposition of polygons into convex sets, *IEEE Transactions on Computers*, C-27, 1978, pp. 1078-82.

[SV] Schäffer, A. A. and Van Wyk, C. J., Convex hulls of piecewise-smooth Jordan curves, *Journal of Algorithms*, to appear.

[Sh1] Shamos, M., Geometric complexity, ACM Symposium on Theory of Computing, Albuquerque, New Mexico, May, 1975.

[Sh2] Shamos, M., Computational geometry, PhD Thesis, Yale University, May, 1978.

[SH] Shamos, M., and Hoey, D., Geometric intersection problems, IEEE FOCS Conference, Houston, Texas, October, 1976.

[Sm] Smith, A. R., Invited talk on the complexity of images in the movies, Second Annual Symposium on Computational Geometry, Yorktown Heights, New York, June, 1986.

[TV] Tarjan, R. E., and Van Wyk, C. J., An $O(n\log\log n)$-time algorithm for triangulating simple polygons, *SIAM Journal of Computing*, submitted.

[Va] Van Wyk, C. J., Clipping to the boundary of a circular-arc polygon, *Computer Vision, Graphics, and Image Processing*, 25, 1984, pp. 383-92.