Computational Geometry on a Systolic Chip

BERNARD CHAZELLE

Abstract — **This paper describes systolic algorithms for a number of geometric problems. For the sake of realism we restrict our investigation to one-dimensional arrays whose communication links with the outside are located at the end cells. Implementations yielding maximal throughput are given for solving dynamic versions of convex hull, inclusion, range and intersection search, planar point location, intersection, triangulation, and closest-point problems.**

Index Terms — **Analysis of algorithms, computational geometry, convolution, parallel computation, pipelining, real-time algorithms, systolic arrays, VLSI.**

I. INTRODUCTION

THE pervasive influence of VLSI in the computer science community has given research on parallel computation its second wind. In contrast with the traditional conception of parallel systems, where several computers are each assigned complicated tasks, VLSI computation, especially of systolic nature, involves the simultaneous use of a great number of very simple processors [18], [26].

As commonly referred to, systolic arrays are one- or two-dimensional arrangements of simple cells locally connected [18]–[21]. The essential features of systolic cells are their *simplicity, regularity,* and *modularity*. Performance-wise, these characteristics are definite assets, as they ensure high levels of *pipelining* and *multiprocessing,* hence providing massive parallelism. They also affect the economics of the approach by making circuit development more cost effective. Indeed, with dropping costs of electronic components and increasing levels of circuit integration, systems designers are facing the prospect of putting hundreds of thousands of gates on a single chip, which so far constitutes a formidable challenge. Systolic architectures are one answer to this challenge. Their modularity permits the designer to decompose the system's architecture into building blocks which can be used repetitively with simple interfaces.

From the origin, the epithet *systolic* has been reserved to special-purpose devices, such as multipliers, priority queues, pattern matchers, etc. With this perspective, systolic arrays were built with wired-in cell implementations, which were not to be a handicap as long as the overall reconfigurability of the array, an essential feature of a systolic architecture, was preserved. Thus, the user was essentially given the freedom to tailor the array to the size of his problem, without having the possibility of modifying the cell definition. If one wishes, however, to optimize the cell specifications or to allow a more versatile use of the systolic device, it is essential that the cell behavior be made programmable [9]. By doing so, it becomes possible to experiment with different systolic implementations of a same scheme without having to build different chips and be caught in the bottleneck of fabrication turnaround. Also, programming the array allows the user to make it fulfill not just one function, but a whole range of related tasks. The merit of this approach partly resides in the combination *versatility and high performance* which it affords. It must also be mentioned that it serves pedagogical purposes by putting systolic design into the hands of the laymen, thus making the conception and use of very high performance devices more accessible.

The purpose of this work is to present a *class-related* systolic processor based on the approach just described. This processor is a programmable systolic array aimed for solving a wide class of geometric problems in a highly unifying manner. This *class* of problems contains many of the most basic questions of computational geometry. Among others, we will find dynamic versions of convex hull, inclusion, range and intersection search, planar point location, intersection, triangulation, and closest-point problems. Whenever possible, we will insist on the dynamic aspect of the problem, for it is often where systolic solutions are at their best. On the other hand, many applications areas involve problems of an inherently dynamic nature, with which we must cope. For example, air traffic control necessitates the real-time solution of closest-point problems on an ever-changing set of points.

After discussing the advantages of systolic architectures in terms of increased adaptability and cost effectiveness, we should investigate the gains in performance to expect from a systolic treatment of computational geometry. To begin with, let us roughly describe our systolic architecture. We consider only one-dimensional arrays, i.e., arrays with a single string of cells, each connected to their one or two neighbors. Furthermore, communication with the outside world (typically, a host computer) takes place solely at either of the end cells. It results from this configuration that although there may be full parallelism in the arrays, the number of I/O operations at any time is always bounded by a constant. We do not make this assumption for the sake of simplicity, but for the sake of *realism*. Indeed, in most applications, the systolic device will receive its data from a sequential computer, therefore the assumption we are making is not a choice but an inevitable reality.

Being now ready to turn our attention to performance considerations, we immediately derive, from the assumption

Manuscript received November 3, 1982; revised November 10, 1983. This work was supported by the Defense Advanced Research Projects Agency under ARPA Order 3597, and the Air Force Avionics Laboratory under Contract F33615-81-K-1539.

The author was with Carnegie-Mellon University, Pittsburgh, PA 15213. He is now with the Department of Computer Science, Brown University, Providence, RI 02912.

0018-9340/84/0900-0774$01.00 © 1984 IEEE

above, that N pieces of data cannot be processed in fewer than N systolic steps. This may seem like a serious handicap, when compared to the $O(N^2)$ or $O(N \log N)$ running times typically offered by sequential geometric algorithms. One may hope at best the gain of a factor N or $\log N$; however, asymptotic figures based on big-Oh considerations are not too relevant in the matter. Indeed, the sole performance goal in our case is to maximize the throughput, i.e., have the systolic array keep up as closely as possible with the *host/device* data rate. This data rate is dependent on the pin bandwidth of the chip, or sometimes in real-time applications, on the rate at which data are made available to the host by the outside (e.g., radar, sensor). Note that the new emphasis made here reflects yet another departure from the traditional study of computational complexity.

It is often the case that a circuit will receive streams of data, each of them pertaining to a different instance of the problem. In this case, maximizing the throughput is called *pipelining*, and to measure the adequacy of the circuit to respond to a stream of requests, we look at its *period*, a concept introduced in [37]. Roughly, the period of a circuit is the minimum necessary delay between two consecutive sets of inputs. Of course, it is highly desirable that our systolic designs have period $O(1)$, which often involves preventing the occurrence of clusters or of the presence of cells waiting for others in order to complete execution. We will discuss these issues in detail later on.

In the next section, we describe the general features of the geometric systolic array, then proceed with a description of the algorithms in the remaining sections.

II. The Geometric Systolic Chip

Most of the systolic arrays which we will describe in this paper have the basic outlook of Fig. 1. Interaction with the outside world takes place solely at the end cells, called *boundary* cells. All of the other cells, called *generic*, are alike, and although boundary cells are assigned additional tasks for I/O purposes, they usually do not differ drastically from the generic cells. Each cell contains a small amount of memory, in the form of a few registers. We distinguish two kinds of registers.

1) Working registers for either storing data (point, edge, angle, $\cdots$,) or for providing temporary storage for the computations.

2) I/O registers for communicating data between adjacent cells.

To avoid dealing with implementation details at this point, we may regard I/O registers as being conceptually "located" on the connection wires between adjacent cells. This shows that the intercell links are critical sections of the computation, and therefore special care must be given to the way they are used. These registers are protected by gates which can be either *open* or *locked* according to the current clock phase. We assume that the whole systolic array is synchronous, and that each cell operates in lock-step. For simplicity, we also assume the existence of two clocks ϕ_1 and ϕ_2 beating in opposition. This allows us to separate input and

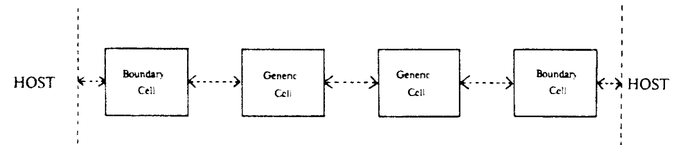


Fig. 1. The one-dimensional systolic array.

output stages easily by requiring that input (respectively, output) gates should all be open (respectively, locked) at ϕ_1 and vice versa at ϕ_2 (Fig. 2).

The lapse of time between two phases ϕ_1 is called a *systolic cycle*. It is to be distinguished from the clock cycle internal to each cell, which is likely to be much shorter. Indeed, a systolic cycle must correspond at least to a number of internal clock cycles necessary for a cell to complete the execution of its stored program. We should observe that this clocking arrangement is not unique; systolic arrays with asynchronous and/or adjacent cells operating in opposite cycles are perfectly feasible, so the choice made here serves only explanatory purposes. The only bit of notation, used throughout, that needs be introduced here concerns the representation of points by capital letters $A, M, X, \cdots$, with a_i denoting the ith coordinate of point A in a Cartesian system of coordinates.

III. Convex Hull Problems

Estimating a population parameter in statistics, or simulating chemical reactions, often requires computing the convex hull of a set of points in a dynamic fashion [35]. For static and dynamic solutions to convex hull problems on a conventional machine, see [16], [17], [29], [30], [35]. Throughout this paper we will assume that all problems are cast in $\mathfrak{R}^2$. To fulfill our purposes, we will devise a systolic structure which supports the following operations.

1) Insert/delete point M.

2) Report all the vertices of the convex hull in clockwise (or counterclockwise) order.

3) Determine whether an arbitrary point M lies inside or outside the convex hull.

Note that M is either a new point or a point already in the structure, depending on the type of query considered. The operation "delete point M" always refers to a vertex of the convex hull, however. As usual with dynamic convex hull routines, deletions and insertions proceed in very different ways. To cope with this problem, we will describe two systolic arrays, **CH1** and **CH2**, supporting the following operations.

*Array **CH1**:*

1) Insert/delete point M.

2) Report all the vertices of the convex hull (in arbitrary order).

3) Determine whether point M lies inside or outside the convex hull.

*Array **CH2**:*

1) Insert point M.

2) Report all the vertices of the convex hull in clockwise (or counterclockwise) order.

3) Determine whether point M lies inside or outside the convex hull.

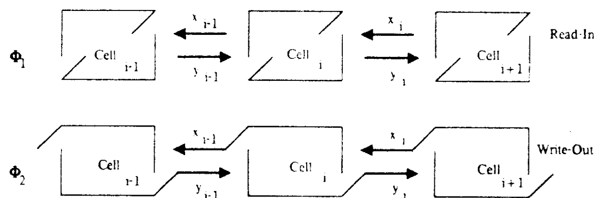


Fig. 2. Handling critical paths.

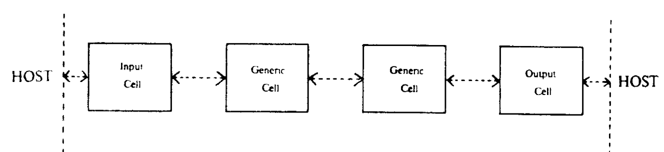


Fig. 3. The overall structure of the array **CH1**.

As we will see in detail later on, the data representation differs significantly between **CH1** and **CH2**. Indeed, while the first array keeps track of the hull vertices in arbitrary order, the latter stores the edges of the convex hull in clockwise order. This distinction will be important in order to achieve optimal throughput. Note, however, that in both cases the input is given in the form of query signals, query points, or new points to be inserted or deleted. We will see later on that in order to support the operations listed at the beginning, it suffices to concatenate **CH1** and **CH2** together.

A. The Array *CH1*

CH1 consists of N cells, so as to handle up to N points at any given time, each cell storing one point. We will assume for the time being that the number of points stored p is very near N (not so near, though, that insertions would immediately cause overflow). At the end of the section, we will consider the case where p may become significantly smaller than the capacity of the array. All operations (updates and queries) are initiated at the input cell with the answers emanating from the output cell (Fig. 3). Each cell stores at most one point, and the order of the cells in the array will not, in general, coincide with the order of the points around the convex hull. Because of deletions, the array may have *holes*, i.e., cells that do not store any point, although surrounded by cells which do.

Implementing Operation 1 is straightforward. Points to be inserted are pumped into the left cell, and travel from left to right stopping at the first vacant cell. A point to be deleted is input in the same way, moving from left to right until it encounters the cell where its copy is stored, which it then marks as vacant. Note that, as mentioned, the array does not keep track of the order of the vertices around the convex hull. Operation 3 relies on the following geometric fact, whose proof we may omit. We define a *wedge* as either of the two regions comprised between two half-lines emanating from the same point. A wedge can be either reflex or convex depending on whether the area chosen forms a reflex or nonreflex angle.

Observation 1: Let $M_0, \cdots, M_{p-1}$ be a set of p points in the plane. A point M lies outside the convex hull of $M_0, \cdots, M_{p-1}$ if and only if there exists a convex wedge centered at M which contains all the points $M_0, \cdots, M_{p-1}$.

This observation shows that we simply have to make the query point M travel from left to right, keeping track of the smallest convex wedge centered at M that contains all the points encountered so far. Note that this wedge represents the widest nonreflex angle (MM_i, MM_j) which can be formed so far. Let T denote this angle which, for simplicity, we denote (M, M_i, M_j). Updating T at each step can be done by a trivial case analysis, as illustrated in Fig. 4. To alleviate the notation, we define the function $F(M, A, B)$ to indicate on which side of AB the point M lies. $F(M, A, B)$ is the sign of the expression $um_1 + vm_2 + w$, where $uX + vY + w = 0$ is an equation of the line passing through A and B, i.e., $u = a_2 - b_2$, $v = b_1 - a_1$, $w = a_1b_2 - a_2b_1$. This provides us with an easy characterization of whether two points M, P lie on the same side of AB, i.e., they do iff $F(M, A, B) = F(P, A, B)$. For simplicity, we will always assume that no three points are ever collinear (relaxing this requirement involves adding unessential tedious details to the algorithms, so it is legitimate to allow such simplifications). The point M will travel across the array, along with the current value of T, examining each new point encountered for a possible update of T. Testing T against a new point C leads to the operations described in Fig. 4. As soon as a point C is found yielding a reflex angle with respect to A or B, a flag (traveling alongside M) can be set to indicate that M has already been determined to lie inside the convex hull. If the flag is never set during the entire traversal of M, the point is declared lying outside the convex hull.

The handling of Operation 3 should be clear by now, so we can proceed with Operation 2. One solution would be, in a first stage, to output copies of all the points, then in a second stage, to reinput them one after the other, executing Operation 3 on each of them. To achieve the same result *in place* and thus maximize the throughput, we must trigger the motion of each point across the entire array so as to be able to compare their relative position against all the others. The problem is to ensure a regular flow that prevents moving points to overtake or run into each other. To overcome this difficulty, we should regard the systolic array as a strip of paper. The idea is to pick up the strip at the input cell end and fold it over, pulling the input cell over from left to right (Fig. 5). By doing so, each point will be indeed brought in contact with each cell in the array. While traveling, each point will carry along its current wedge T, as defined earlier.

At any time, there are two types of points to distinguish: on the one hand, the points not yet in motion, and whose storing cell is responsible for the updating of their associated wedge; on the other hand, the points already pulled over and traveling to the right. These points will travel along with their associated wedge, updating it at each visit of a new cell. The updating is of the same nature as in Operation 3. Note that the left end of the folded strip will move twice as slowly as the right end. For this reason, no operation on the systolic array should be initiated within N systolic cycles after the start of Operation 2. This will ensure that no query will ever propagate to a cell already engaged in a computation for a previous query. Note that this delay still ensures optimal

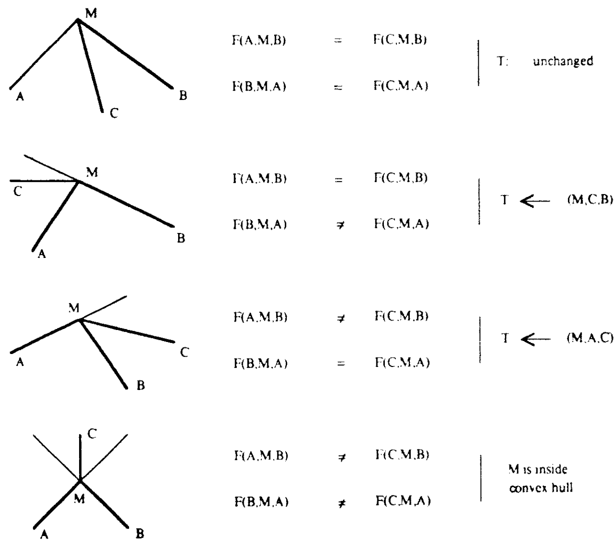


Fig. 4. Testing inclusion in the convex hull.

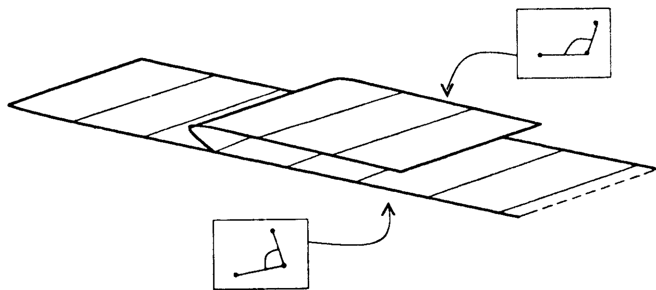


Fig. 5. The fold-over operation.

throughput as long as the number of points stored is on the same order of magnitude as the size of the array. To implement this *fold-over* operation, we need essentially two signals: one is the query itself, which follows the right end of the covering strip; the other follows the other end, i.e., the *front* of the folding strip. Keeping track of the front is necessary so as to signal the next cell to be started that at the cycle following the next, it will have to send a copy of itself to the right, thus becoming the current left front of the covering strip. Array **CH1** is quite simple and we do not feel the need to elaborate further on its description. Details of the cell implementation for **CH2** will be given in the next section, however, because of the more complex nature of the algorithm.

If the actual size of the convex hull p is significantly smaller than N, we notice some inefficiency in the scheme just proposed that might jeopardize the maximal throughput feature of the array. First of all, there is no need for a delay of N systolic cycles after the start of Operation 2. A reduced delay of roughly p cycles will indeed prevent any subsequent query to overtake the front of the folding strip, and will therefore be sufficient (provided that the array is free of large vacant gaps). Also, while the one-way scheme grants maximal throughput, a single query takes on the order of N cycles to be answered. We could possibly reduce this delay to a figure proportional to p by requiring each answer to reverse its motion as soon as it is computed, and thus be output through the "input" cell. These modifications are important at the implementation level, for they grant interesting trade-offs between performance and design simplicity. These are design and not conceptual issues, however, and are as such beyond the scope of this paper.

B. The Array CH2

This structure supports only insertions, but in return it provides an ordered description of the convex hull at any time. Also, since the array stores only vertices of the convex hull, it can support an arbitrary number of insertions, as long as this convex hull always keeps a number of vertices on the order of N. To begin with, let us give the geometric background behind the algorithm. Assume that $M_0, \cdots, M_{p-1}$ are the vertices of a convex p-gon P, given in clockwise order. Let M be an arbitrary point outside P. Considering the infinite line passing through an edge e of P, it is easy to see that adding M to the convex hull will cause the disappearance of e if and only if the line lies between M and P. The following result is simply a more formal statement of the remark above, and we leave out the proof (see illustration in Fig. 6). Once again we shall assume that no three points may be collinear. F is the function defined in the previous section. For clarity, we will use the notation $F(X, Y, Z) < 0$ (respectively, > 0) to denote $F(X, Y, Z) =$ "$-$" (respectively, $=$ "$+$").

Observation 2: Let $M_0, \cdots, M_{p-1}$ be the vertices of a convex p-gon P, in clockwise order. Let M be an arbitrary point and Q denote the convex hull of $P \cup \{M\}$.

1) M lies inside P iff $F(M, M_i, M_{i+1}) < 0$, for all i; $0 \le i \le p - 1 \pmod p$.

2) M_iM_{i+1} is an edge of Q iff $F(M, M_i, M_{i+1}) < 0$. Also, if M does not lie inside P, it is a vertex of Q and its adjacent vertices are, in clockwise order, M_u and M_v, defined uniquely by $F(M_{u-1}, M_u, M) < 0$, $F(M_{u+1}, M_u, M) < 0$, $F(M_{v-1}, M, M_v) < 0$, and $F(M_{v+1}, M, M_v) < 0$.

The array **CH2** has the same overall structure as **CH1** (Fig. 3). Instead of a point, each cell now stores an edge of the convex hull, however, and the left-to-right order in the array corresponds to a clockwise traversal of the boundary of the convex hull. Operation 1 (*inserting point M*) causes M to travel from the input cell to the output cell, computing the function F defined above in order to determine whether M lies inside the convex hull. If it lies outside, two edges have to be added to the structure, and in general, a bunch of consecutive edges (at least one, anyhow) must be removed. More precisely, assume that $M_iM_{i+1}, \cdots, M_{j-1}M_j$ are the consecutive edges of P to be removed. Upon encountering M_iM_{i+1}, M must cause the cell currently visited to substitute M_iM for M_iM_{i+1}. All the subsequent cells will delete their contents, until M encounters the first edge (M_jM_{j+1}) not to be affected by the insertion of M. At this point, the current cell must hand the cell M_jM_{j+1} to its right-hand side neighbor, and keep the edge MM_j in store. M has now ceased to cause changes in the array, and it can terminate its motion. However, there is now one cell in the array with two edges.

To repair this anomaly, we make sure that the cell keeps its additional edge but forward its former contents to its right neighbor. This only causes to shift the anomaly one cell to the right, but iterating on this process will eventually cause the

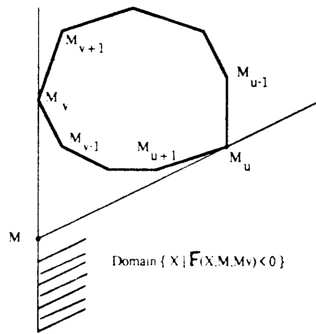


Fig. 6. Computing convex hulls in clockwise order.

last nonvacant cell to release an edge to its neighbor, which solves the problem. This phenomenon is known as *rippling*, as it mimics the propagation of a wave on water. We should observe that if the last nonvacant cell has no right neighbor, *overflow* must be reported. However, the insertion may have just caused the deletion of a number of edges, in which case reporting overflow would be undesirable. In general, we pose as a requirement that: "*No overflow should be reported if there is any vacant cell in the array*," no matter where. To comply with this rule, we must ensure that vacant cells which have edges on their right-hand side, i.e., *holes*, must be *filled* by edges from the right. To do so, it suffices to have each cell always check whether its left-hand side neighbor is vacant, in which case it must pass its contents to it. As a result, it appears that, in general, two opposite motions will take place within the array: one, to the right, corresponds to queries and insertions, while the other, leftwards, is meant to fill the holes just created.

Operation 2 can be implemented in the same manner as **CH1**, folding the array over itself, with the difference that no computation is required in this operation since the edges are already ordered around the convex hull. An alternative is to pump out all the edges of the array through the input cell, which will give the edges in clockwise order (note that the array will then no longer exactly conform to Fig. 3 since the left end-cell should now be labelled *input/output cell* instead of simply *input cell*). Operation 3 is a simple application of Observation 2, similar to Operation 1, yet without altering the state of the array. The query point M travels left-to-right, checking its location with respect to each edge in turn. If M is always found to lie on the same side of the edge as the interior of the polygon, inclusion must be reported, otherwise M lies outside the convex hull.

If the capacity of the array is significantly larger than p, the size of the convex hull, we face the same issues mentioned in the previous section but our previous comments still apply. Aside from this concern, it is clear that **CH2** confronts the designer with a whole set of problems that did not apply to **CH1**. The most delicate of them comes from our insistence on preventing spurious overflows by constant *hole checking*. For this reason, we must take a closer look at the implementation of **CH2**. In order to preserve the conceptual simplicity of this paper, these technical considerations appear in Appendix.

The last item which we must examine is the coupling of **CH1** and **CH2** mentioned at the beginning of this section. If we concatenate **CH2** to the right of **CH1**, we add the capability of ordered convex hull report to the present functionality of **CH1**. Indeed, a report signal will trigger a fold-over operation in **CH1**, which will in turn feed every point of this array into **CH2**. The latter array will then proceed to insert each of these points, thus forming the ordered list of convex hull edges. Finally, after a specified delay (see Appendix for a discussion on interquery delay), a report signal from **CH1** to **CH2** will complete the computation, with all the edges being output through either the right or left end-cell of **CH2**, depending on the scheme chosen. Note that the fold-over operation in **CH1** can be released from any convex hull computation, if desirable, since **CH2** must take up this task anyway. Synchronizing the entire process entails, in particular, specifying the number of cycles **CH1** must wait before receiving any further query. This delay depends on the relative speeds of **CH1** and **CH2**, but is easily seen to involve at most N cycles, up to within a constant factor.

IV. Inclusion, Intersection, and Closest-Point Problems

We next show that many of the most common geometric problems can be solved by means of a simple unifying scheme. The underlying idea, already used in arithmetic or pattern matching [12], [18], [20], exploits the inherent suitability of systolic designs to testing each input data against the contents of each cell, or more precisely their ability to perform pipelined convolution-like computations.

Let $S_1, \cdots, S_N$ be the data stored in the array, and let $a_1, \cdots, a_j$ denote a list of queries in the order with which they arrive at the input cell; the goal is to compute for each query a_k the value of $T_k^{(N)}$, defined by the recurrence relation

$$T_k^{(0)} = 0$$

$$T_k^{(i)} = F(T_k^{(i-1)}, S_i, a_k) .$$

Fig. 7 sketches a systolic solution for this class of problems. As we will see, it is possible, in most cases, to make the systolic scheme dynamic, that is, capable of handling updates in the array. If no order among the S_i's is required, a *delete(e)* operation simply results in marking the cell storing e vacant, while *insert(e)* causes the storing of e in the first cell vacant from the left. If on the other hand, some order is to be preserved among the S_i's, an *insert* operation will involve searching for the appropriate (nonnecessarily vacant) cell, and store the new element in it, thus possibly causing the remaining cells to ripple to the right. Symmetrically, deleting an element will incur the creation of a hole and the start of a leftward motion aimed at filling it, resulting in the propagation of the hole to the right end of the array. For a list of applications areas where the geometric problems addressed next arise in practice, consult Shamos' thesis [35].

A. Inclusion Problems

1) Point/Polygon: Does point M lie in polygon $P = M_1, \cdots, M_N$?

The polygon is taken to be simple, but no convexity as-

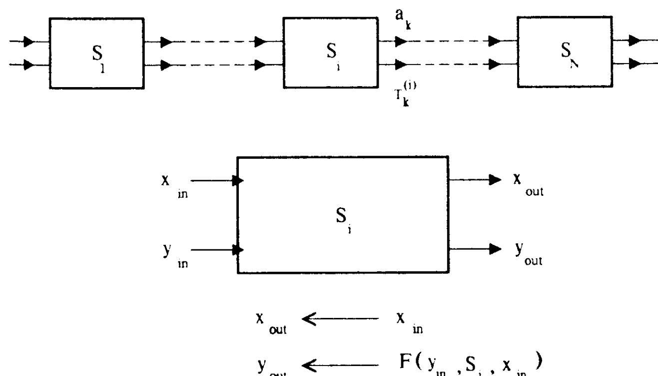


Fig. 7. A systolic scheme for iterative problems.

sumptions are made. It is possible to achieve unit period with the following systolic scheme. The register S_i holds the pair (M_i, M_{i+1}), where the list $M_1, \cdots, M_N$ corresponds to a clockwise traversal of the boundary of P. The variables **x** and **y** of Fig. 7 are, respectively, the point M and the pair $(uv, u'v')$, where uv and $u'v'$ are the edges of P (with u, v and u', v' giving the clockwise direction), such that their intersection with the vertical line L passing through M forms the smallest segment so far containing M (Fig. 8). Testing for the inclusion of point M involves pumping M throughout the array, from left to right, updating the pair of edges in **y** on the fly.

Eventually, the array can output an inclusion message if y_{out} falls into case b) of Fig. 8, or a noninclusion signal if it falls into case c). This is a direct application of the Jordan Curve Theorem, stating that a simple closed curve in the plane divides the plane into two parts: the *inside* and the *outside*. Note that the scheme used above is far from unique, and other tests for inclusion may lead to equally simple systolic structures. For example, simply counting the number of intersections with the line L above and below M is sufficient since these numbers are odd iff M lies inside the polygon. Note that the edges of the polygon do not even have to be stored in order in the array. Ensuring that each edge is directed, say, in clockwise order, is sufficient.

2) Planar Point Location: Given a planar graph with faces $f_1, \cdots, F_N$, and a point M, determine the face where M lies.

For this problem, several sequential algorithms with an optimal $O(\log N)$ query time exist [11], [22], [31], [35], but for the most part these methods require complicated preprocessing. Instead, we can design a very simple systolic array to solve this problem with unit period. To do so, we simply represent the graph by placing in the array, next to each other, clockwise descriptions of the faces. This consists of storing an edge per cell so that each face is described by a subarray of contiguous cells. Each cell will therefore store an edge, along with the name of the face associated with its subarray. Since in this way each edge is represented exactly twice, no more than a linear number of cells will be required. If we now regard the graph as a union of polygons, locating a query point M comes down to testing the point for inclusion with respect to each polygon in turn. We can use the previous scheme, finally concluding with a report of the name of the unique polygon (or edge, or vertex) that contains M.

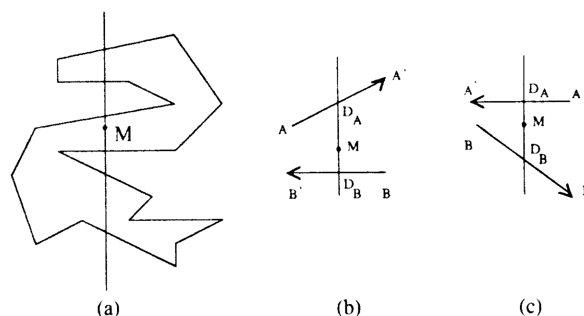


Fig. 8. Testing inclusion.

3) Range Search: Given a set S of N intervals and a query point M, compute the number of intervals of S that contain M [1], [4], [6], [10], [24], [25], [28].

The systolic array will store one interval per cell, so that the query point can scan the array left-to-right, checking for inclusion in the interval stored in each cell, and updating the partial count. Note that the problem can be extended to d-ranges and arbitrary polygons, as long as each cell is only made to store a constant amount of data. For example, polygons should be represented by subarrays of cells, each storing one edge. We omit the details of these straightforward extensions.

4) Intersection Search: Given a set of intervals (respectively, rectangles parallel to the axes) and a query segment (or a query rectangle), report the number of segments (respectively, rectangles) that intersect the query object [1], [4], [6], [10], [24], [25], [28].

Once again, testing pairwise intersection requires constant time, which ensures unit period. The algorithm is straightforward and needs no further development.

The last two problems arise constantly in graphics [27], and in design-rule checking for VLSI circuits [1], [4]. Often, however, instead of a mere number of intersections, an explicit report of all the intersecting pairs is desired. To give our systolic arrays this added capability, it is sufficient to add only a few instructions to the algorithms. One solution is to prescribe that upon encountering an intersection, a query first sends the intersecting pair forward to the next cell, then only proceeds in the same direction. Of course, this will cause a slowdown, therefore to prevent overtaking by subsequent queries, we require that before moving an object to the next cell, the algorithm first check the vacancy of that cell. To that end, each cell must keep sending *vacant* or *occupied* signals to its left-hand side neighbor. The scheme is somewhat similar to the traffic management described for **CH2**.

B. Intersection Problems

For sequential algorithms, see [1], [4], [6], [7], [23], [28], [35], [36].

5) Intersection of Polygons: Given two polygons P, Q, determine whether they intersect.

If we wish to determine only if the boundaries intersect, we may simply store the edges of P in the systolic array, and have those of Q travel left-to-right, testing each edge encountered for intersection. It is easy to extend the method and solve the

general problem by observing that P and Q intersect if and only if at least one of the following conditions is satisfied.

1) A vertex of P lies in Q.
2) A vertex of Q lies in P.
3) The boundaries of P and Q intersect.

Thus, it suffices to add to each cell two copies of the procedure described for Problem 1); one with respect to P, the other with respect to Q. Note that each cell must check whether the passing edge is the last edge of Q, in which case, it must tag a *yes* or *no* signal to the tail of Q to acknowledge if either endpoint of the edge stored in the cell lies inside Q or not. This is straightforward, and details are left to the attention of the reader.

6) Intersection of Half-Planes: Given N half-planes $H_1, \cdots, H_N$, compute their intersection.

This problem requires $\Omega(N \log N)$ time on a conventional machine [5], [35], [36]. As usual, we expect our systolic implementation to yield maximal throughput, and thus display an overall $O(N)$ time performance. Moreover, as we will see, it is easy to provide the array with the capability of handling queries and updates, without losing on the overall performance. This addition is very similar to the connection of **CH1** and **CH2** described earlier for the solution of dynamic convex hull problems. Actually, the similarity between the two problems is very deep, for it stems from the geometric duality which exists between convex hulls and intersections of half-spaces [5], [32].

Let I be the intersection of the N half-planes $H_1, \cdots, H_N$. If I is not empty, it is a convex polygon with possibly one open side, i.e., two edges that are half-lines meeting at infinity. Note, for the sake of completeness, that the intersection I may also be reduced to a single half-plane or an infinite parallel strip. It is possible to represent I either by a list of the lines supporting the edges of I, in arbitrary order, or if we wish more information, by a list L of edges (A, B), as they appear in a clockwise traversal of the boundary. In case of an open polygon I, we require that the vertex at infinity should appear at the ends of the list. For example, we may have two points I_1, I_k, in the list

$$L = \{(I_1, A_1), (A_1, A_2), \cdots, (A_k, I_k)\}$$

with the understanding that the edge I_1A_1 (respectively, A_kI_k) is the infinite ray starting at A_1 (respectively, A_k) and passing through I_1 (respectively, I_k).

Similar to **CH1** and **CH2**, we will design two systolic arrays **INT1** and **INT2** to support the following operations.

*Array **INT1**:*

1) Insert/delete half-plane H.
2) Report all lines on the boundary of I in arbitrary order.
3) Determine whether point M lies in I.

*Array **INT2**:*

1) Insert half-plane H.
2) Report all vertices of I in clockwise (or counterclockwise) order.
3) Determine whether point M lies in I.

Because of the similarity with **CH1** and **CH2**, we may only sketch the algorithms. Any standard representation of half-planes is adequate. For example, $(u, v, w, \geq)$ can be used to denote the half-plane $uX + vY + w \geq 0$. As before, each cell will store either a single line in the case of **INT1** or a single vertex (expressed as the intersection of two lines) in the case of **INT2**. The only point to investigate about **INT1** is, in Operation 2, the type of matching involved in the "fold-over" process. To begin with, it is easy to see that a half-plane H_i contributes an edge to I iff its supporting line L_i lies in the intersection of the $N - 1$ remaining half-planes $H_1, \cdots, H_{i-1}, H_{i+1}, \cdots, H_N$. Then since the intersection of L_i with the intersection of any subset of $H_1, \cdots, H_N$ is, if not empty, a segment, a half-line, or L_i itself, it can be expressed by means of at most two points, which can then be updated as L_i is matched against each H_j in turn. All of the other features of **INT1** are similar to those of **CH1**. As for **INT2**, we assume that, at all times, the array contains a clockwise description of I, with each edge stored in a separate cell. Once again, all the operations are handled as in **CH2**, including the hole-filling process, so we may omit the details.

C. Closest-Point Problems

7) Nearest Neighbor: Given N points, $M_1, \cdots, M_N$, and a query point M, determine the nearest neighbor of M (see [2], [3], [35]).

For this problem, we allow the dimension of the space to be arbitrary and the distance to be based on any of the $L_1, L_2, \cdots, L_\infty$ norms. Recall that the L_p norm of a vector $(x_1, \cdots, x_d)$ in d-space is $(|x_1|^p + \cdots + |x_d|^p)^{1/p}$. Whereas efficient solutions on a conventional machine involve the use of fancy data structures (e.g., *Voronoi diagrams, point location search trees, k–d trees,* etc.) entailing substantial implementation overhead, a simple dynamic systolic scheme can be devised as follows.

Once again, we store one point per cell. Queries travel left-to-right, determining their nearest neighbor on the fly. To do so, each query is accompanied by the closest point found so far. Updates in the structure are handled as in **CH1**, that is, inserting a point into the first available cell encountered, and deleting it by simply marking the corresponding cell vacant. If desired, a *report-all-nearest-neighbors* query can be added to the set of allowed operations. This instruction, which causes the nearest neighbor of each point in the array to be output, can be implemented by the *fold-over* procedure of **CH1**.

Applications areas where a device for reporting near neighbors would be of great interest are many. Air traffic control is one example; in this situation, typically, a few radars transmit streams of signals giving updates on the position of nearby airplanes, and minimum safety distances between planes must be constantly ensured. To speed up the signaling of anomalous positions, an *emergency* output port can be reserved on each cell, with direct link to the host. Although slightly unsystolic, this feature is quite feasible as long as emergency reports remain rare events.

8) Euclidean Minimum Spanning Tree: Given N points in the plane, construct a tree of minimum total length whose vertices are the given points [35].

In [33], Savage proposes a systolic structure for computing the connected components of a graph. This structure is a

one-dimensional systolic array, which can be connected to Leiserson's priority queue [21], so as to compute the minimum spanning tree in linear time. This result has appeared in the literature [33], so we refer the reader to this source for details. We only wish to mention that since the complete graph underlies the Euclidean minimum spanning tree problem, it might be desirable to deal with a sparse subgraph known to contain the MST (for example, the Delaunay triangulation [35]).

9) Triangulation: Partition the convex hull of N points $M_1, \cdots, M_N$ into triangles, so that the vertex set of the partition is exactly the set of N points.

This problem, which arises frequently in numerical analysis (*finite element method, numerical interpolations,* etc.), has an $\Omega(N \log N)$ lower bound on a sequential machine [35]. A one-dimensional systolic scheme can yet achieve linear time, while supporting the following features.

Array ***TRI****:*

1) Insert a point in the triangulation.

2) Determine in which face of the triangulation a query point lies.

3) Report all the faces of the triangulation, each face in clockwise order.

Operation 1 is to be understood as the insertion of a new point into the current triangulation, thus necessitating the introduction of new edges. The input arrives in the form of points to be inserted or in the form of query signals. The array **TRI** computes an arbitrary triangulation, without any consideration of "goodness." Since in many cases, however, it is crucial that certain quality criteria are met, e.g., minimizing a function of the edges, the array might be used more advantageously within the framework of a more complicated heuristic. Each occupied cell may serve one of two purposes: either it stores an edge of the convex hull (Register $R = (A, B)$) with A, B giving the clockwise orientation, or it stores the vertices of a triangle in clockwise order (Register $R = (A, B, C)$). We also require that, from left to right, the edges stored in the cells of the first kind should appear in clockwise order (Fig. 9). Finally, we assume the existence of a flag F to signal the first edge of either the upper or the lower chain of the convex hull; see Fig. 13 and discussion of **CH2** in the Appendix for the definition of these terms. With this arrangement, Operation 2 simply involves testing the query point against each triangle, carrying the containing triangle along with M, when detected (if ever), otherwise reporting an *outsideface* message, if no such triangle has been found. Yet simpler, Operation 3 involves pumping out the contents of each cell storing a triangle, one by one—see *report* operation in Section III-B. To handle Operation 1, two cases must be considered.

1) M lies inside a triangle (e.g., DFC in Fig. 9). We must replace R by, say, MCD, and insert the triangles MDF and MFC into the next two right neighbors of the current cell. This is done by rippling to the right (Fig. 10, case 1).

2) M lies outside the convex hull, and thus will become a vertex of the new convex hull. The algorithm is very similar to **CH2**. Instead of deleting non-convex-hull edges, however, we must now insert new triangles into the array. Referring to Fig. 12, with AB being the edge currently examined, and C, A, B occurring in clockwise order around the convex hull, we can give the new case analysis (see example in Fig. 10, case 2). 1) Delete AB, add AM and MBA; 2) no action; 3) delete AB, add MBA; 4) add MA.

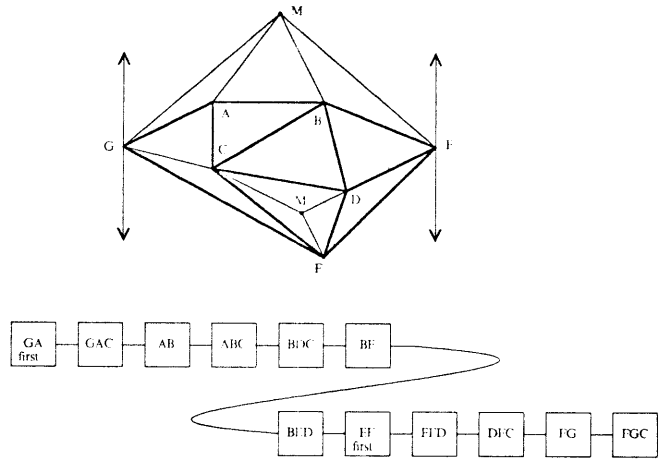


Fig. 9. Triangulating a set of points.

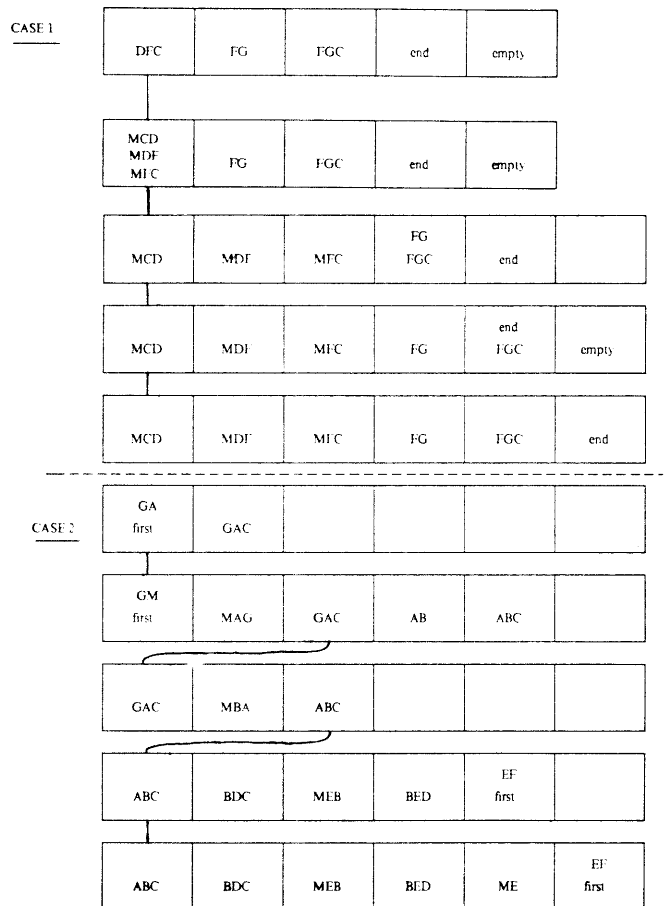


Fig. 10. The triangulation array **TRI** in action.

Fig. 10 illustrates the two basic cases on a running example. Some steps in the rippling process have been omitted for the sake of brevity. Also, since the figure only shows a subpart of the array, we have indicated the correspondence between successive subparts by curved lines. Before closing

this section, a few remarks concerning the throughput of the array are in order. As usual, a report signal will preempt all subsequent queries until completion. More interesting is the processing of inserts. In both of the cases examined above, at most a constant number of objects (edge or triangle) is added to the structure (via rippling) except when dealing with the repeated occurrence of case 2.3. This corresponds to the situation where M is outside the convex hull and (referring to Fig. 12) we are deleting AB and adding MBA. Since adding a triangle is in this case always accompanied by the deletion of a convex hull edge, we can use the same register to store the new object. This shows that only two more registers will be needed in the worst case when inserting a new point. These registers will store the two new convex hull edges. Recall that only one was needed in the case of **CH2** (see the Appendix). Following the reasoning found in the Appendix, we conclude that a delay of nine idle cycles between successive queries will ensure that queries cannot overtake each other. This margin of safety is actually overly conservative, and there is ample room for optimization.

V. Conclusions

The purpose of this work has been to present systolic designs for several geometric problems. Most of the algorithms described in this paper involve two distinct types of tasks. One is concerned with the actual computation of geometric functions, and is in general the easier to understand. The other involves initiating and granting requests, which entails moving data around, i.e., adding new items into the array or filling holes created by deletions. In general, the flow of data is irregular and not predetermined since it is contents dependent. With the exception of priority queues and similar structures [14], [21], this constitutes a major departure from most systolic arrays described in the literature, especially those for arithmetic computations [12], [18], [20]. Instead, most of the known systolic structures have a fixed predetermined data flow, usually highly regular. One major difficulty with *random* motion is the absence of adequate tools for proving the correctness of the algorithms, and in particular, describing the behavior of the data flow. There, certainly, lie promising avenues of research.

In practice, most of the algorithms given here should undergo substantial revision before being implemented, so as to take into account the opportunities for local optimization granted by the particular applications for which the device is intended. Also, the current state of VLSI technology certainly imposes definite constraints which are bound to influence the overall design. For example, the pin/bandwidth limitation of today's chips, rightly seen by many as the major bottleneck in VLSI, can be partly overcome by clustering several cells onto a single chip. Also, one highly desirable feature of a systolic array is that it should be compute bound and not I/O bound [19]. This amounts in practice to ensure that the cells do not spend most of the time idle, *waiting* for inputs to come. As things stand it is doubtful that this could be the case with the algorithms given here since executing the microcode, alone, is most likely to take longer than completing any I/O operation. At any rate, it is always possible to circumvent this difficulty by providing each cell with a small random access memory (perhaps $\sim$(1–2K) with present NMOS technology), and simulating a few dozens of cells sequentially with a single processor. This solution also has the advantage of making the handling of very large inputs possible, without requiring an excessive number of chips, hence partly overcoming the interchip communication bottleneck. Of course, this may seem like an overt denial of the systolic philosophy. However, the presence of many cells ($\sim$100) within the array will largely preserve the systolic nature of the overall structure as well as its benefits.

At the implementation level, we urge staying away from floating-point representations whenever possible because of the inevitable complications which they entail. Note that in all the algorithms given above, only fixed-point additions, subtractions, and multiplications are needed, with the exception of the intersection algorithms, which involve the solution of linear equations. In this case, division is needed, yet can be avoided if rational numbers are kept as pairs of fixed-point numbers, as is common practice in linear programming. We should also observe that the arithmetic computations involved in the algorithms are in general very simple and limited, most of them consisting of simple fixed-point inner products.

Future work in the area of systolic algorithms includes, of course, their actual implementation and evaluation. Also, trying to classify the problems that lend themselves to systolic implementations appears very worthwhile. Finally, we must once again emphasize the current need for an original description language for systolic systems, as well as new tools for studying the behavior and proving the correctness of the underlying algorithms.

Appendix
Implementing *CH2*

For reasons which will become apparent later on, the frequency of operations initiated on the input cell must follow the rules below.

1) After starting an operation on the input cell, wait for at least seven idle cycles before initiating another request.

2) No operation can be initiated before Operation 2 is completely finished. A special symbol *end* will acknowledge this fact.

These figures correspond to a general relation, which will be discussed later on. For the sake of simplicity, actually, our rules have been made overly conservative. Because of its generality, we may focus exclusively on the generic cell.

A. The Implementation

We assume that the report scheme chosen involves pumping out the edges through the left end-cell. As shown in Fig. 11, the generic cell can be described with six basic variables and three registers R, F, C, the former storing one edge (A, B) of the convex hull. The type of the variables x, y, z is, respectively, (data, signal, data/signal), where signal denotes flags or commands. Testing the inclusion of a point $x_{in} = M$ in the convex hull involves having y_{in} set to *thruinclusion* if noninclusion has already been determined, or

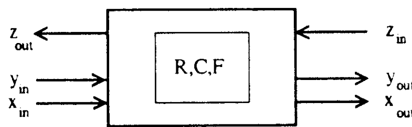


Fig. 11. The generic cell for **CH2**.

inclusion otherwise, in which case, computing $F(M, A, B)$ allows us to iterate on to the next cell. The variables z_{in}, z_{out} serve a double purpose. On the one hand, if z_{in} is a pair (A, B), the cell is vacant ($R = \epsilon$) and must be filled ($R \leftarrow z_{in}$). On the other hand, once a *report* (Operation 2) has been initiated on the input cell, the contents of each nonvacant cell will get to travel towards the input cell to be eventually output. To distinguish between these two kinds of leftward motion, one bit (*report*) is tagged to z_{in}, i.e., $z_{in} = (report, A, B)$, so that the cell knows that it must only pass this value along ($z_{out} \leftarrow (report, A, B)$). Of course, if $y_{in} = report$, z_{out} is set to $(report, R)$.

The last case to examine (Operation 1) is by far the most delicate. To decide the status of a point M in the convex hull, Observation 2 shows that four possible situations should be considered. With M traveling along the array from the input to the output cell, let $R = (A, B)$ be the edge stored at the cell currently visited, and let the variable u be set to *in* if the edge (C, A) of the previous (nonempty) cell satisfies $F(M, C, A) < 0$, or *out* otherwise. A variable v is defined similarly with respect to $F(M, A, B)$. Observation 2 shows that the following actions should be taken.

1) $u = in, v = out$ [Fig. 12(a)]. *Delete R to replace its contents by (A, M).*

2) $u = in, v = in$ [Fig. 12(b)]. *No action.*

3) $u = out, v = out$ [Fig. 12(c)]. *Delete R.*

4) $u = out, v = in$ [Fig. 12(d)]. *Insert (M, A) before R. Send R to next cell.*

Note that if all four cases should arise, they would occur with the order $2, \cdots, 2, 1, 3, \cdots, 3, 4, 2, \cdots, 2$ (up to circular permutation). Since we wish to pipeline the updates, it is very important that as an insert-M operation travels left-to-right, the insert signal, at any time, leaves behind the exact clockwise description of the boundary as it should be after inserting M. For this reason, we must ensure that if the four cases should arise, they do so in the following order:

$$2, \cdots, 2, 1, 3, \cdots, 3, 4, 2, \cdots, 2.$$

This problem comes from the fact that the variable u cannot be computed for the first cell since it involves knowledge of the last occupied cell in the array. To overcome this difficulty, we adopt a slightly different representation of a convex polygon, which involves partitioning the boundary into two chains of consecutive edges. One, the *upper chain*, consists of the upper edges of the polygon, i.e., edges with increasing x-coordinates in clockwise order; the other, the *lower chain*, consists of the lower edges, defined as the edges pointing to the left (Fig. 13).

We now require that from left to right, the array **CH2** should store first the edges of the upper chain, then the edges of the lower chain, both in clockwise order. Of course, we must assume the presence of a flag register F in each cell,

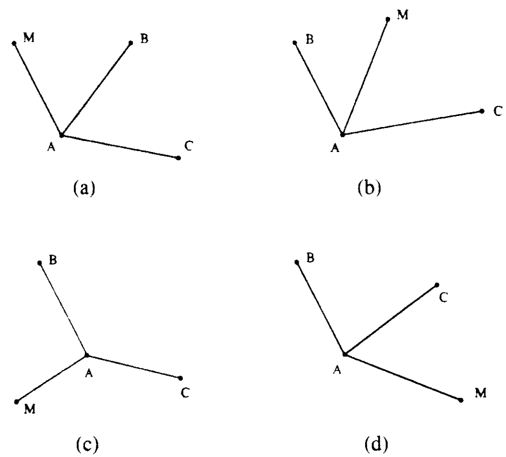


Fig. 12. Establishing the status of a new point.

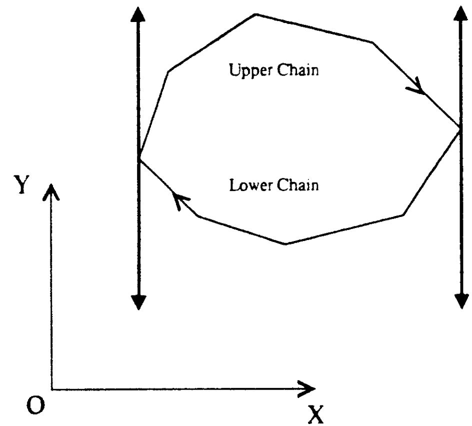


Fig. 13. The partition of a convex polygon.

which takes on the value *firstup* (respectively, *firstlow*), if the cell is currently storing the first edge of the upper (respectively, lower) chain. Otherwise, F is set to ϵ. The flag plays the role of u for the two edges in the array whose neighbors, in counterclockwise order, are conceptually two infinite vertical rays.

Situations 1 and 2 are straightforward to handle, unlike situation 3 which creates "holes" in the array, and situation 4 which adds one extra edge. In the latter case, the edge R and the flag F will bounce their contents on to the next cell, which will store them in its registers, and send the former contents of these registers to its neighbor. This process will iterate until the last cell ($R = end$) has been reached, thus adding one to the overall cell occupancy. While a cell is busy sending its contents to its neighbor, it must hold up the *insert* request to forward it at the next step. To do so, it uses the third register C. To handle situation 3, i.e., to fill holes, we require that at the end of the computation, each cell checks whether it is vacant ($R = \epsilon$), in which case it issues a *hole* signal to its right-hand neighbor ($y_{out} \leftarrow hole$), provided that y_{out} has not already been set to another value (e.g., a query/update signal). Upon receiving a *hole* signal ($y_{in} = hole$), the cell must empty its register T onto its left-hand neighbor ($z_{out} \leftarrow R$). One major difficulty is that with a naive implementation a right-moving query/update may miss some left-moving edges. To circumvent this pitfall, we re-

serve the odd systolic cycles for all leftward transfers, and the even cycles for the remaining computations.

Only a few words need be added concerning the input and output cells. Before computation starts, we assume the presence of $R = end$ in the input cell. Aside from a special treatment for the first three points entering the array, most of the behavior of this cell is identical to that of the generic cell. As for the output cell, its most notable feature is to detect and report possible overflows, as well as outputting an inclusion message if $y_{in} = inclusion$, and a noninclusion message if $y_{in} = thruinclusion$.

*B. Correctness of the Algorithm for **CH2***

To begin with, we should note that along with an insert-M request, two flags (u and w) should be tagged to M. The variable u is, as shown above, the status *in* or *out* of M with respect to the previous cell, and w is a flag set to *firstup* (respectively, *firstlow*) if the next edge created, MA, happens to be the first of the upper (respectively, lower) chain. This information is needed when the first edges are deleted by repeated occurrences of situation 3, and w is thus the only way to acknowledge the first new edge that it is indeed the first edge of a chain. It is important to realize that filling holes with a left motion of edges is meant only to improve the performance of the array, i.e., put the limitation on the size of the convex hull rather than on the number of operations which can be performed. For this reason, we may for the sake of the exposition ignore the hole-filling task momentarily. One crucial point to ensure in the algorithm is that during a single cycle y_{out} is never set more than once.

In order to do so, we may start with a few helpful observations. Let us call an even *stage* the conjunction of an even cycle followed by an odd cycle. The rules on operation rate specified at the outset of the Appendix impose a delay of at most four even stages between two consecutive operations. However, an *insert* operation may entail the loss of one stage, caused by the possible (unique) setting of C, thus reducing the above delay to 3. On the contrary, a cell may issue a hole signal ($y_{out} \leftarrow hole$) possibly at every even stage, and similarly a cell is in a position to respond to a hole message at every odd stage ($z_{out} \leftarrow (R, F)$). From these facts, we derive in particular that whenever $C \neq \epsilon$, we also have $y_{in} = \epsilon$, from which it is easy to see that there is never any conflict in setting y_{out}. Now including the hole-filling instructions, we only have to show that there is no conflict in setting the register R. More precisely, we must prove that whenever $z_{in} = (A, B, F) \neq \epsilon$, we have $R = \epsilon$. This comes from the fact that $z_{in} = (A, B, F)$ if and only if at the previous odd cycle y_{out} had the value *hole*. This, in turn, implies that at the end of the previous even cycle, we had $R = \epsilon$. Since, in addition, R can only be set to ϵ (if it is ever) at odd cycles, our proof is complete.

The last item to verify is what precisely motivated the distinction between odd and even cycles: the assurance that all right-moving queries or updates encounter all the edges of the array. If $z_{in} = (A, B, F) \neq \epsilon$, the first action taken by the cell at an even cycle is to set (R, F) to z_{in}, so that an *inclusion* or *insert* operation at that cycle will effectively deal with the just-left-moved edge. On the other hand, since the edge leaves the cell only under a $y_{in} = hole$ situation, the cell will not have to handle any query/update at the next even cycle, so its contents may leave the cell without missing any matching, which proves our claim. Our final investigation concerns the storage efficiency of the array. We claim that no overflow will ever occur as long as the number of vertices in the convex hull at any time does not exceed $N/2$.

The above assumption clearly implies that no more than $N/2$ cells are occupied ($R \neq \epsilon$) at any time since inserting a vertex involves first deleting old edges, then adding the new ones. Trouble may arise, however, if edges tend to cluster towards the output cell. To dispel that worry, we introduce the concept of *leading front*, defined as the rightmost cluster of occupied cells, i.e., the rightmost group of cells without $R = \epsilon$. A leading front can be characterized by the position H of the first cell, measured as its distance to the input cell, along with the length L of the cluster. To prove the absence of leading fronts near the output cell, hence the absence of overflow, it clearly suffices to establish the following result.

Observation 3: $H + 2L \leq N$.

Proof: To look at the evolution of a leading front, suppose that the front $(H, L - 1)$ just had one cell added to it as the result of an insertion, yielding a front (H, L). From the rules, it follows that during the next seven cycles, no more cell can be added to the front. However, a *hole* signal will necessarily be transmitted to the leftmost cell of the front during the first two even cycles; therefore, this cell will be detached from the front by the second odd cycle, at the latest. For the same reason, a *hole* signal will reach the new leftmost cell of the front by the fourth even cycle at the latest; therefore, this cell will also detach itself before the seven cycles are elapsed, thus leaving a front $(H + 2, L - 2)$ in the worst case. This completes the proof. ∎

It is easy to generalize the rules specified above, which may be useful for tuning the algorithms according to the average distribution of requests. For the sake of simplicity, we will give a very conservative estimate of a safe scheduling of requests. Let A be the number of cells in the systolic array, and let α be the ratio *speed of head/speed of tail*. If we wish to allow up to N convex hull vertices in the array at any time, we must have the relation $\alpha A \leq A - N$, hence $\alpha \leq 1 - N/A$, satisfied. On the one hand, if a (respectively, b) is the delay, measured in number of stages, imposed between consecutive *insert* (respectively, *inclusion*) operations, the right-end of the leading front cannot grow at a speed higher than $1/a$. On the other hand, over a period of time T, the number of stages during which the left-end of the leading front is unable to transfer the contents of its register to the left is at most $T/a + T/b$. This shows that the speed of the left-end is at least $1 - (1/a + 1/b)$; therefore, we have the relation $1/a \geq \alpha(1 - (1/a + 1/b))$. Consequently, setting the requirement that

$$1/a \leq (1 - N/A)(1 - (1/a + 1/b))$$

will prevent the leading front from stretching all the way to the right-end of the systolic array; it will therefore ensure a safe scheduling.

This completes our discussion of implementation issues related to **CH2**.

ACKNOWLEDGMENT

The author wishes to thank H. T. Kung and the Programmable Systolic Chip Group at Carnegie-Mellon University as well as the referees for their helpful comments.

REFERENCES

[1] J. L. Bentley and T. Ottmann, "Algorithms for reporting and counting geometric intersections," *IEEE Trans. Comput.*, vol. C-28, Sept. 1979.

[2] J. L. Bentley, D. F. Stanat, and E. H. Williams, Jr., "The complexity of near-neighbor searching," *Inform. Processing Lett.*, vol. 6, Dec. 1977.

[3] J. L. Bentley, B. W. Weide, and A. C. Yao, "Optimal expected-time algorithms for closest-point problems," in *Proc. 16th Allerton Conf. Commun., Control and Computing*, 1978.

[4] J. L. Bentley and D. Wood, "An optimal worst-case algorithm for reporting intersections of rectangles," *IEEE Trans. Comput.*, vol. C-29, pp. 571–577, July 1980.

[5] K. Q. Brown, "Geometric transforms for fast geometric algorithms," Ph.D. dissertation, Carnegie-Mellon Univ., Pittsburgh, PA, 1979.

[6] B. Chazelle, "Filtering search: A new approach to query-answering," in *Proc. 24th Annu. FOCS Symp.*, Tucson, AZ, 1983, pp. 122–132.

[7] ——, "Intersecting is easier than sorting," in *Proc. 16th Annu. STOC Symp.*, Washington, DC, 1984, pp. 125–134.

[8] J. Clark, "A VLSI geometry processor for graphics," *Lambda*, vol. 1, no. 2, 1980.

[9] Y. Dohi, A. Fisher, H. T. Kung, and L. Monier, personal communication, PSC Group, Carnegie-Mellon Univ., Pittsburgh, PA, 1982.

[10] H. Edelsbrunner, "A time- and space-optimal solution for the planar all-intersecting-rectangles problem," Tech. Univ. Graz, Graz, Austria, Tech. Rep., Apr. 1980.

[11] H. Edelsbrunner, L. Guibas, and J. Stolfi, "Optimal point location in a monotone subdivision," in preparation, 1983.

[12] M. J. Foster and H. T. Kung, "The design of special-purpose VLSI chips," *Computer*, vol. 13, Jan. 1980.

[13] M. R. Garey, D. S. Johnson, F. P. Preparata, and R. E. Tarjan, "Triangulating a simple polygon," *Inform. Processing Lett.*, vol. 7, June 1978.

[14] L. J. Guibas and S. M. Liang, "Systolic stacks, queues, and counters," in *Proc. Conf. Adv. Res. VLSI*, Massachusetts Inst. Technol., Cambridge, MA, Jan. 1982.

[15] L. J. Guibas, H. T. Kung, and C. D. Thompson, "Direct VLSI implementation of combinatorial algorithms," in *Proc. Caltech Conf. on VLSI*, Jan. 1979.

[16] R. A. Jarvis, "On the identification of the convex hull of a finite set of points in the plane," *Inform. Processing Lett.*, vol. 2, 1973.

[17] D. G. Kirkpatrick and R. Seidel, "The ultimate planar convex hull algorithm?," in *Proc. 20th Annu. Allerton Conf.*, Monticello, IL, Oct. 1982.

[18] H. T. Kung, "Let's design algorithms for VLSI systems," in *Proc. Caltech Conf. on VLSI*, Jan. 1979.

[19] ——, "Why systolic architectures?," Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-CS-81-148, Nov. 1981; see also *Computer*, Jan. 1982.

[20] H. T. Kung and C. E. Leiserson, "Systolic arrays for VLSI," in *Proc. Sparse Matrix 1978*, Soc. Indust. Appl. Math., 1978.

[21] C. E. Leiserson, "Systolic priority queues," in *Proc. Caltech Conf. on VLSI*, Jan. 1979.

[22] R. J. Lipton and R. E. Tarjan, "Applications of a planar separator theorem," *SIAM J. Comput.*, vol. 9, no. 3, [illegible]

[23] H. G. Mairson and J. Stolfi, "Reporting and counting line segment intersections," unpublished, 1984.

[24] E. M. McCreight, "Efficient algorithms for enumerating intersecting intervals and rectangles," Xerox Palo Alto Res. Center, Palo Alto, CA, Tech. Rep. CSL-80-9, June 1980.

[25] ——, "Priority search trees," Xerox Palo Alto Res. Center, Palo Alto, CA, Tech. Rep. CSL-81-5, 1981.

[26] C. Mead and L. Conway, *Introduction to VLSI Systems*. Reading, MA: Addison-Wesley, 1980.

[27] W. M. Newman and R. F. Sproull, *Principles of Interactive Computer Graphics*. New York: McGraw-Hill, 1973.

[28] J. Nievergelt and F. P. Preparata, "Plane-sweeping algorithms for intersecting geometric figures," *Commun. Ass. Comput. Mach.*, vol. 25, Oct. 1982.

[29] M. H. Overmars and J. Van Leeuwen, "Dynamically maintaining configurations in the plane," in *Proc. 12th Annu. SIGACT Symp.*, Los Angeles, CA, May 1980.

[30] F. P. Preparata, "An optimal real-time algorithm for planar convex hulls," *Commun. Ass. Comput. Mach.*, vol. 22, July 1979.

[31] ——, "A new approach to planar point location," *SIAM J. Comput.*, vol. 10, no. 3, Aug. 1981.

[32] F. P. Preparata and D. E. Muller, "Finding the intersection of a set of N half-spaces in time $O(N \log N)$," *Theoretic. Comput. Sci.*, vol. 8, no. 1, Feb. 1979.

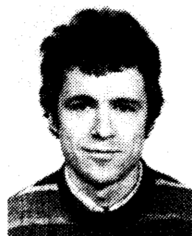
[33] C. Savage, "A systolic data structure chip for connectivity problems," in *Proc. Carnegie-Mellon Univ. Conf. on VLSI Syst. and Computat.*, Pittsburgh, PA, Oct. 1981.

[34] J. B. Saxe and J. L. Bentley, "Transforming static data structures to dynamic structures," in *Proc. 20th Annu. FOCS Symp.*, Puerto Rico, Oct. 1979, pp. 148–168.

[35] M. I. Shamos, "Computational geometry," Ph.D. dissertation, Yale Univ., New Haven, CT, 1978.

[36] M. I. Shamos and D. Hoey, "Geometric intersection problems," in *Proc. 17th Annu. FOCS Symp.*, Oct. 1976, pp. 208–215.

[37] J. Vuillemin, "A combinatorial limit to the computing power of VLSI circuits," in *Proc. 21st Annu. FOCS Symp.*, Syracuse, NY, 1980, pp. 294–300.



Bernard Chazelle received the Diplôme d'ingénieur from the Ecole Nationale Supérieure des Mines de Paris, Paris, France, in 1977, and the M.S. and Ph.D. degrees in computer science from Yale University, New Haven, CT, in 1978 and 1980, respectively.

From 1980 to 1982 he was a Research Associate in the Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA. In September 1982, he joined the Department of Computer Science, Brown University, Providence, RI, where he is currently an Assistant Professor. His research interests include analysis of algorithms, complexity theory, computational geometry, VLSI, and graphics.

Dr. Chazelle is a member of the Association for Computing Machinery.