1993 International Conference on Application-Specific Array Processors - October 25 - 27, 1993 - Venice, Italy.

L. Dadda and B. Wah (editors). IEEE Computer Society Press

A Real-Time Systolic Algorithm for On-the-fly Hidden Surface Removal

T. Risset*
LIP - IMAG
École Normale Supérieure de Lyon
46, Allée d'Italie
69364 Lyon Cédex 07 - France

S. W. Song†
University of São Paulo
Institute of Mathematics and Statistics
Department of Computer Science
C.P. 20570 - São Paulo, SP 01498- Brazil

Abstract

Hidden surface removal for real-time realistic display of complex scenes requires intensive computation and justifies usage of parallelism to provide the needed response time. We present a systolic algorithm that identifies visible segments on a scanline with the "real-time" characteristic: visible segments are output on-the-fly as soon as segments are input to the systolic array. The proposed systolic architecture consists of a linear array of simple cells. The size of the systolic array is equal to the maximum number of input segments that are crossed by a vertical line. The correctness of the systolic algorithm is proven by first establishing the correctness of an initial version, which is then successively transformed to more efficient versions, while preserving the properties for correctness.

1 Introduction

Given a set of 3D objects and a viewing specification, we wish to determine which lines or surfaces of the objects are visible, either from a center of projection (perspective projection) or along a direction of projection (parallel projection), so that we can display only the visible lines or surfaces. This process is known as visible-surface determination or hidden-surface elimination. Hidden-surface removal has been well-researched by the Computer Graphics and the Computational Geometry communities. As it is often used for real-time realistic display of complex scenes (flight simulator, medical images, etc.), there is a dramatic need of fast treatment to give short response time. The hidden surface removal requires intensive computation thus justifying usage of parallelism to provide efficient algorithms.

*Supported by the Project C3 of the French Council for Research CNRS, and by the ESPRIT Basic Research Action 3280 "NANA" of the European Economic Community.

†Supported by the Program BID/USP (Interamerican Development Bank/University of São Paulo) - Project 30.01, FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo) - Proc. No. 91/1646-0 and CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) - Proc. No. 306063/88-3 and PROTEM-CC.

There are several different techniques to solve the hidden surface removal problem. Algorithms have been first presented by Sutherland, Sproull and Schumacker [10]. A recent general survey can be found in [2]. For a parallel implementation survey see [3]. Scanline algorithms operate at image precision to create an image one scanline at a line [2, p. 680].

In recent years, systolic architectures [1, 4, 5, 8] have been used in various application domain, especially in Computer Graphics where specialized architecture seem to be useful [2, p. 920]. In this paper we present a linear systolic array that implements the determination of visible segments on a particular scanline. We will apply the treatment on each scanline of the image. Vaidya et al. [11] have presented systolic algorithms for this same problem, that use $2N$ cells and the visible segments are output after $3N$ steps (where N is the number of input segments). The size of the systolic array proposed is equal to the maximum number of input segments that are crossed by a vertical line and the execution time is less than $3N$. Furthermore, the proposed algorithm has the desirable "real-time" characteristic, namely visible segments are output on-the-fly as soon as segments are input to the systolic array. The advantage of our systolic architecture approach is that one can reach a real-time output to display the projection on a screen.

We will assume that a description of a three-dimensional scene is given in the form of polygons with some attribute, such as color, that may differ from polygon to polygon. The viewing position is also given. We will also assume that a preprocessing on the image send non-intersecting segments, ordered by their left end-points or extremities, into the systolic array. We will then obtain the parallel projection. This algorithm may also be used for perspective projection but, in this case, a preprocessing called the perspective transformation [2, p. 657] is performed. If we focus on our algorithm, we do not use any kind of coherence because each scanline is treated independently and, in a scanline, each segment in disjoint from the others. However, we can notice that if our 3D image is composed of non-intersecting polygons then object coherence can be used in preprocessing and apply the Sechrest and Greenberg object-space algorithm [2, p. 685] which may give good reduction on the number of scanlines computed.

Two systolic algorithms will be shown. The first version is merely for the purpose of correctness proof. We then proceed to adapt it to a second version, without violating the needed properties for correctness. The second version utilizes a smaller number of cells, equal to the maximum of segments crossed by a vertical line, and has the real-time characteristic mentioned earlier.

In Section 2 we present the problem formulation and discuss a general solution. In Section 3 we propose a preliminary version of the systolic architecture and cell algorithm. In section 4 we give a sketch of the correctness proof for this first version. In Section 5 we adapt the first version to a more efficient one while preserving the properties needed for correctness.

2 Problem formulation and discussion of a general solution

Consider n non-intersecting segments on a plane given by their end-point or extremity coordinates.

A segment point (x, y) is *visible* if there is no other segment point (x, y') with $y' < y$.

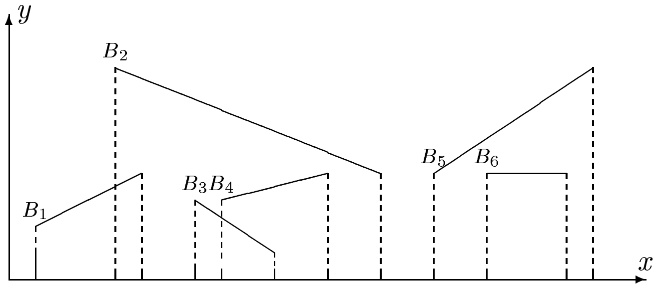

Figure 1: Consecutive intervals defined by the segment end-points

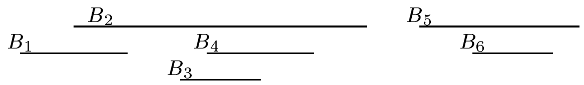

Figure 2: Segments with same visibility characteristics as the previous example

The problem is to identify all such visible segment points.

Consider the consecutive intervals defined by all the segment end-point coordinates x_i. It is easy to see that if a segment has a visible point in an interval, then all its points lying in the interval are visible. See Figure 1 for an example.

As noted in [11], a simple solution can be formulated from the above observation, that consists of the following. At each starting point x_i of an interval, compare the y coordinates of all segment points (x_i, y) and select the segment with the smallest y value. All the points of this segment are visible in this interval.

Given the segment end-point coordinates (x_1, y_1) and (x_2, y_2), it is easy to compute the y coordinate of a point (x, y) on the segment for any x, $x_1 \leq x \leq x_2$. With the consideration of the previous paragraph, the presentation of our method can be simplified by considering the set of segments of Figure 2, with equivalent visibility properties.

The problem tackled here is often merged with other processing kernels. Thus it is hard to find a precise complexity analysis. In [7], a very complete complexity analysis on problem related to union of isothetic rectangles is provided. We can notice that, if we consider segments of figure 2, the problem of finding the visible parts of these segments is equivalent to finding the contour of a union of intersecting rectangles (each segment would be the *bottom* of a rectangle). The complexity given in [7] for this problem is $O(nLog(n))$. The sequential complexity of our problem is therefore $O(nLog(n))$.

Some preprocessing are needed to decompose the problem. In particular, we have to sort the segment by their left end-points, which can also be done, in time $O(nLog(n))$. The assumption of non-intersecting segments is not always verified. If not, we will need to provide another preprocessing (described in [7]).

Consider a segment S with end-points (x_1, y_1) and (x_2, y_2), we use the notation left(S) and right(S) to denote x_1 and x_2, respectively. We denote by $B_1, B_2, \ldots$ the input segments to the systolic algorithm. Furthermore, the segments to be input to the systolic algorithm

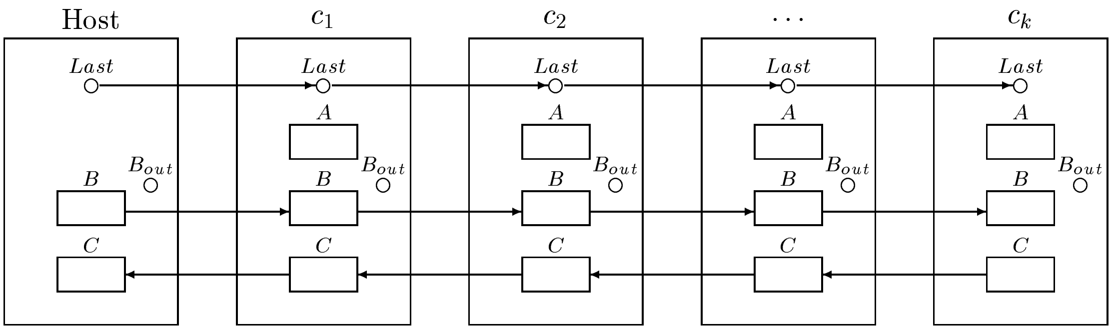

Figure 3: Array architecture - version 1

are already sorted in ascending order by their left(B_i). This preprocessing can be easily done by a simple systolic algorithm, for example given in [6], in linear time.

3 Systolic solution - version 1

The purpose of this preliminary version is merely for correctness proof. We present a systolic algorithm that is not efficient in terms of time and number of cells needed. We derive some properties to prove its correctness. These properties will be essential to other versions to be described later.

3.1 Array architecture

The systolic array architecture is shown in Figure 3. Each cell has three segment registers called A, B, and C, and two 1-bit flags called *Last* and B_{out}. Each one of the 3 registers can hold one segment. We use a *null* content to indicate that a particular register is empty. When not empty, the registers are used as follows. Registers A and B hold segments that are compared against each other, to be shown shortly. Register C contains a visible segment that will be output to the host. The *Last* flag, if equal to 1, indicates the last or rightmost cell that is currently being utilized. All the remaining cells have this flag equal to 0. The B_{out} flag is set to 1 when the segment in B can be sent to the right neighbor.

In the cell algorithms that follow, we use the notation

$$R \Leftarrow S$$

to denote *register transfer* which means that the content of register S is to be transferred to register R, thereby making S empty (with null content). Similarly, if R and S are flag registers, flag R gets the value of flag S which will contain zero after the transfer, independently of its original value.

The following assumptions will be made.

1. Each cell accesses its neighbors to get its input data. Thus, each cell can access the B register and the flags *Last* and B_{out} of its left neighbor, and the C register of its

right neighbor, to be denoted respectively by

$$B.\text{left.neighbor}, Last.\text{left.neighbor}, B_{out}.\text{left.neighbor}, \text{ and } C.\text{right.neighbor}.$$

We consider the host as the left neighbor of cell 1. The last cell has no right neighbor and any attempt to access a non-existant cell will result in null content.

2. At the beginning of execution, the host places the first input segment in its B register and sets its flags B_{out} and $Last$ to 1. Once the value of the $Last$ flag is transferred, this flag register in the host remains zero throughout the duration of the execution.

3. The host will supply its B register with the next input segment whenever it becomes empty, and set B_{out} to 1. Furthermore, after the last input segment is removed from its B register, the host will put in B a fictitious segment with left position $= +\infty$. The purpose of this is to flush out all remaining segments in the array.

4. At each step the host transfers the contents of the C register of cell 1 to its own C register. If not null, then it contains an output visible segment.

3.2 Cell algorithm 1

Each cell executes Algorithm 1 at each step (Figure 4). Be aware that Algorithm 1 represents what is usually called a *step* or a *clock cycle* in systolic description. Here, a *step* of a cell is divided into two *phases*: the *input phase* and the *comparison and output preparation phase*. In the input phase, each cell gets its needed input from neighbor cells, as noted in assumption 1 above. At the beginning the segment registers in all the cells contain the null value and the flag registers contain 0. By assumption 2, the host sets its flags B_{out} and $Last$ to 1 and places the first input segment in its B register.

During the comparison and output preparation phase, registers A and B are compared, and the various registers are redefined. For the sake of clarity, we use a pictorial description for this phase. At the beginning of this phase, except in case I, there is 2 segments: an "older" segment or resident segment in register A, and a newly input segment in register B. We compare these 2 segments and apply the method mentioned in Section 2 to a very particular case of only 2 segments. After the computation (that is, after the output preparation phase) the segments (or part of the original ones) in registers C, A, or B, if they are non-null, are all visible. When a B register is not empty after the computation, there is $B_{out} = 1$. This means it will be the input to the right neighbor during the next step. While the A register contains a resident segment, the B register contains a segment "in transit". Segments "in transit" will travel to the right and thus be compared to all the other resident segments to the right. Informally speaking, Algorithm 1 implements the method of Section 2 by transforming the general case of N segments to a number of simple cases of 2 segments each, which are treated in parallel, in a pipelined fashion.

4 Correctness proof

Now we will prove that the algorithm we have proposed is right. We must show that, each time a segment is output (in C), it is really visible, and that every visible part is output during the algorithm.

Algorithm 1

Input phase:

1. if (B empty or $B_{out} = 1$) & B_{out}.left.neighbor $= 1$ then
   $B \Leftarrow B$.left.neighbor and
   if $Last \neq 1$ then
   $Last \Leftarrow Last$.left.neighbor;
2. $C \Leftarrow C$.right.neighbor

Comparison and output preparation phase:

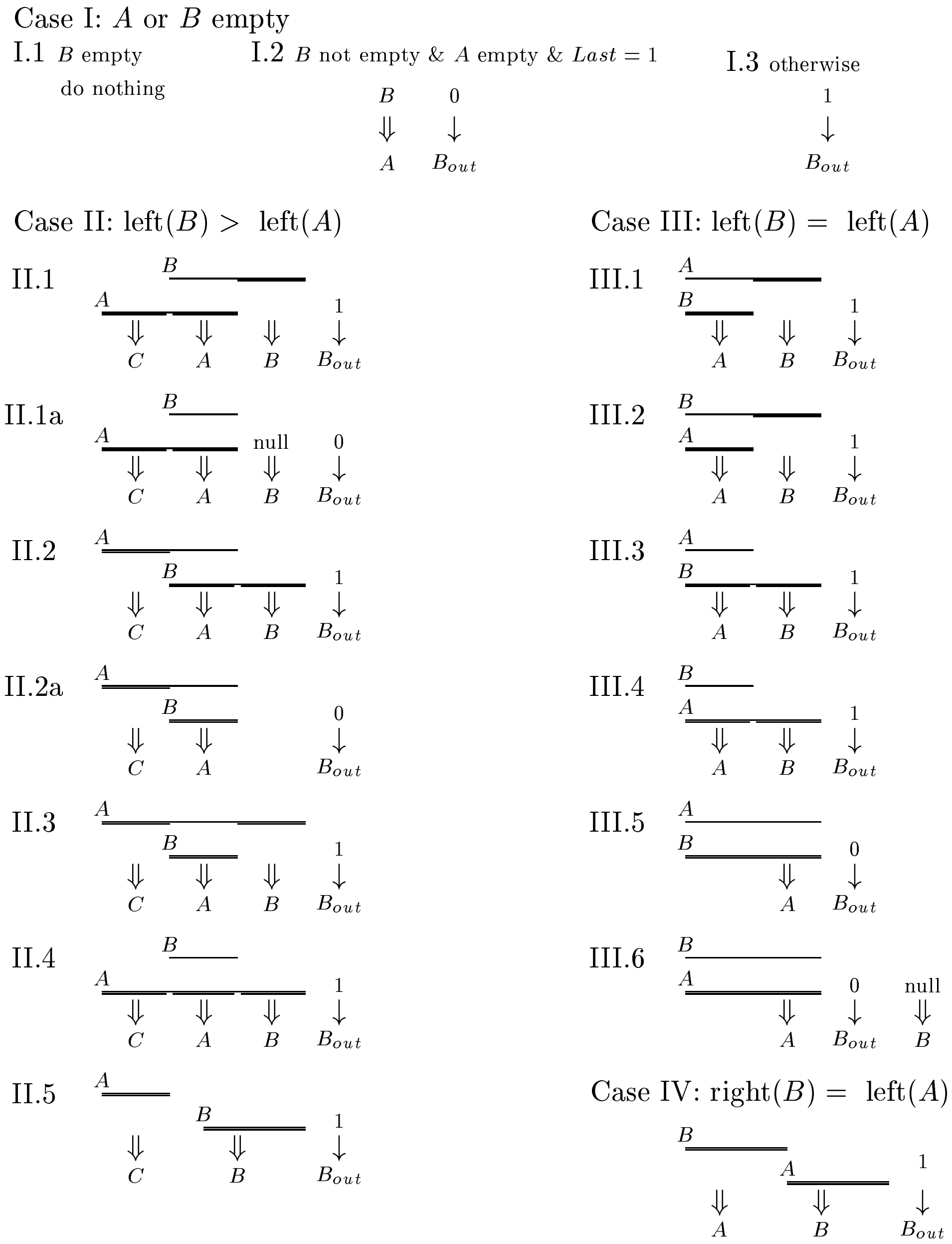

Figure 4: Cell algorithm: version 1

We assume that segments $B_1, B_2, \ldots$ enter the array by increasing left end-point. Therefore, once B_i is input to the array, for any segment $B_j (j > i)$ input after B_i, we have $\text{left}(B_i) \leq \text{left}(B_j)$.

As we can see, the architecture is bidirectional. During the proof, we will not care about the output channel C. We assume that once a cell has output a segment in C, there is no problem in bringing this segment to the host. When we will have proved that the algorithm provides the right visible segments, we will show that there is no problem to output these visible segments to the host. But during the proof, we only consider the B channel, thus we have a unidirectional array. In order to allow us to separate the input, we suppose that at each step, there is only one segment B moving in the array (if we wait a sufficient number of steps between two B-input). After that, the fact of sending one B at each step will only be a pipeline version of what we have proved. This transformation works because every B-segment goes at speed one (one cell per step) until it stops. Thus, the next segment could be input one, two or n steps later, it will meet the same segments in the same cells.

To show the proof, we will consider one more case, to be called case V, not taken into account by Algorithm 1 (Figure 4). Later on we will prove that this case never happens. Case V will resume all the cases that are not treated by cases I to IV, namely, when $\text{left}(B) < \text{left}(A)$ and $\text{right}(B) \neq \text{right}(A)$. In this case A does not change and we B is " destroyed" (meaning that no segment is send to the next cell, by making flag $B_{out} = 0$).

4.1 Theorem 1

We first show a result on the array configuration during execution of the algorithm. In the following proof, we call a cell *empty* if its A-register is empty, and *full* if its A-register contains a segment. The segment that is in the A-register of cell i will be called S_i. S_i^L will be the segment of cell i after B_L has gone through the cell and B_L^i is the segment that will be transmitted to cell $i+1$ after the computation with B_L in cell i.

Theorem 1 *When there is no B-segment moving in the array, the array is in a configuration which we will call the "no-hole" configuration with J and K that is:*

- *There exist $J (J \geq 0)$ and $K (K > 0)$ such that the first J cells of the array are empty, the following K cells are full (cell c_{J+1} with segment $S_{J+1}, \ldots$, cell c_{J+K} with segment S_{J+K}), with:*

$$\begin{aligned}
\textit{right}(S_{J+1}) &= \textit{left}(S_{J+2}) \\
\textit{right}(S_{J+2}) &= \textit{left}(S_{J+3}) \\
&\vdots \\
\textit{right}(S_{J+K-1}) &= \textit{left}(S_{J+K}).
\end{aligned}$$

Proof:

See [9]

¿From this theorem, we can easily deduce the other results:

- During the entire execution of the algorithm, situation V never occurs in any cell.

- In a particular cell c, when a visible part T of a segment is output, for all segments B entering the array after this output, we will have left$(B) \geq$ right(T).
- When we have input K segments, each full cell contains a segment which is visible with respect to segments $B_1, \ldots, B_K$.
- We never "hide" a segment which is visible.
- When we output a segment, it is visible.
- We output all the visible segments (the algorithm is right).

4.2 The output problem

As we have said at the beginning, during the proof, we did not care about eventual collisions on the C channel. In the proof of [9], we show that when a cell produces an output, this cell is the first full cell of the array. As the C-channel goes from right to left, no previous output or following output can collide with it.

This proves also that there is at most one output per step.

4.3 Complexity

It is not easy to analyze precisely the complexity of the algorithm because it depends a lot upon the configuration of the segments at the beginning. In the worst case, each segment force one more cell to be active. In such case, if N is the number of input segments, N cells are needed and the algorithm is performed in $3N$ steps: N steps to input all the segments, N steps until the last segment reaches the last cell and N steps until this last segment is ouptput. We will see how to improve the complexity of this algorithm in the following section.

5 Systolic solution - version 2

Theorem 1 shows us that we lose both space and time. When the array is in a configuration "no-hole" with J and K, the first J cells are empty and will never be active again. Furthermore, any segment input to the array will have to go through J empty cells to reach a full cell. In this section we try to get a better algorithm by "removing" dynamically these empty cells. As we cannot physically remove these cells, we will translate all the segments of the full cells towards the empty cells. We will manage to obtain an array that is always in a "no-hole" configuration with 0 and K. In Algorithm 2, when one cell c is emptied by a segment B, it will indicate this to its right neighbor $c{+}1$ and S_{c+1} will come into c. During the next step, cell $c+1$ will be empty and S_{c+2} will come into cell $c+1$. The "hole" will therefore propagate through the entire array until it reaches the last full cell. Segment B must wait one step in cell c (waiting for segment S_{c+1}), thus all the input of the array are retarded (otherwise, B would cross S_{c+1}). This can be done dynamically because of the array architecture: the next segment is sent to a cell only if the cell is ready to receive the segment.

In this algorithm, the work of the first cell is more complicated than the work of the other cells thus, the first cell is *included* in the host. This simplifies the cell architecture and algorithm, but puts a demand on the host which must be fast enough to match the speed

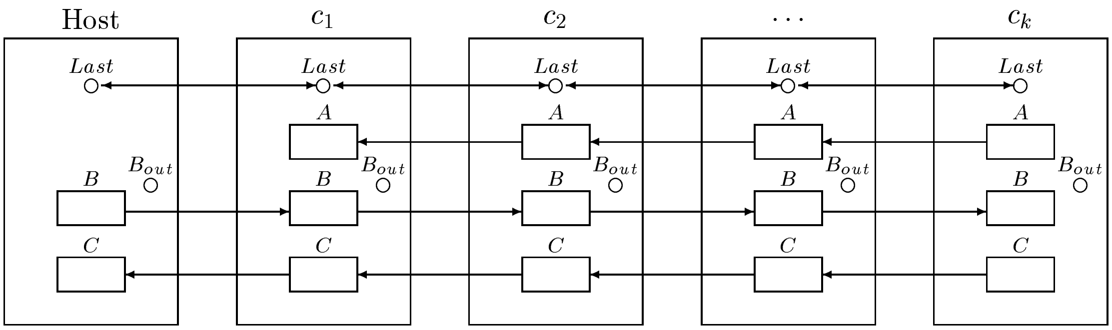

Figure 5: Array architecture - version 2

of the systolic array. Due to the advantage of a simplified cell, we may want to choose this version even at the penalty of slowing down the clock of the systolic array. This solution should be preferred if the resulting response time still satisfies the demand imposed by the applications.

5.1 Array architecture

The version 2 systolic array architecture is shown in Figure 5. To perform what we mentioned above we need a little more control than in the first algorithm, but the principle is basically the same. There will be one more channel for the A registers (from right to left). Its purpose is to propagate to the right any "hole" that may arise in cell 1, thus guaranteeing the "no-hole" configuration with 0 and K mentioned earlier. The channel of the *Last* signal will become bidirectional. When a segment A is transferred back to its left neighbor, the *Last* flag is also transferred together. Moreover, each cell is allowed to access the A and the *Last* register of its right neighbor.

5.2 Cell algorithm 2

The new cell algorithm 2 is given in Figure 6. Relative to Algorithm 1, there are only two differences. We have added one more step (step 2) in the input phase that brings in segment A, together with the *Last* flag, from its right neighbor, when register A is empty. In the comparison and output preparation phase, only case II.5 has changed: there is $B_{out} \leftarrow 0$ instead of $B_{out} \leftarrow 1$. This means that in case II.5, B will no longer be taken away by its right neighbor. With A transferred to C, a "hole" appears at this step. During the next steps it will be propagated to the right by step 2 of the input phase, mentioned earlier.

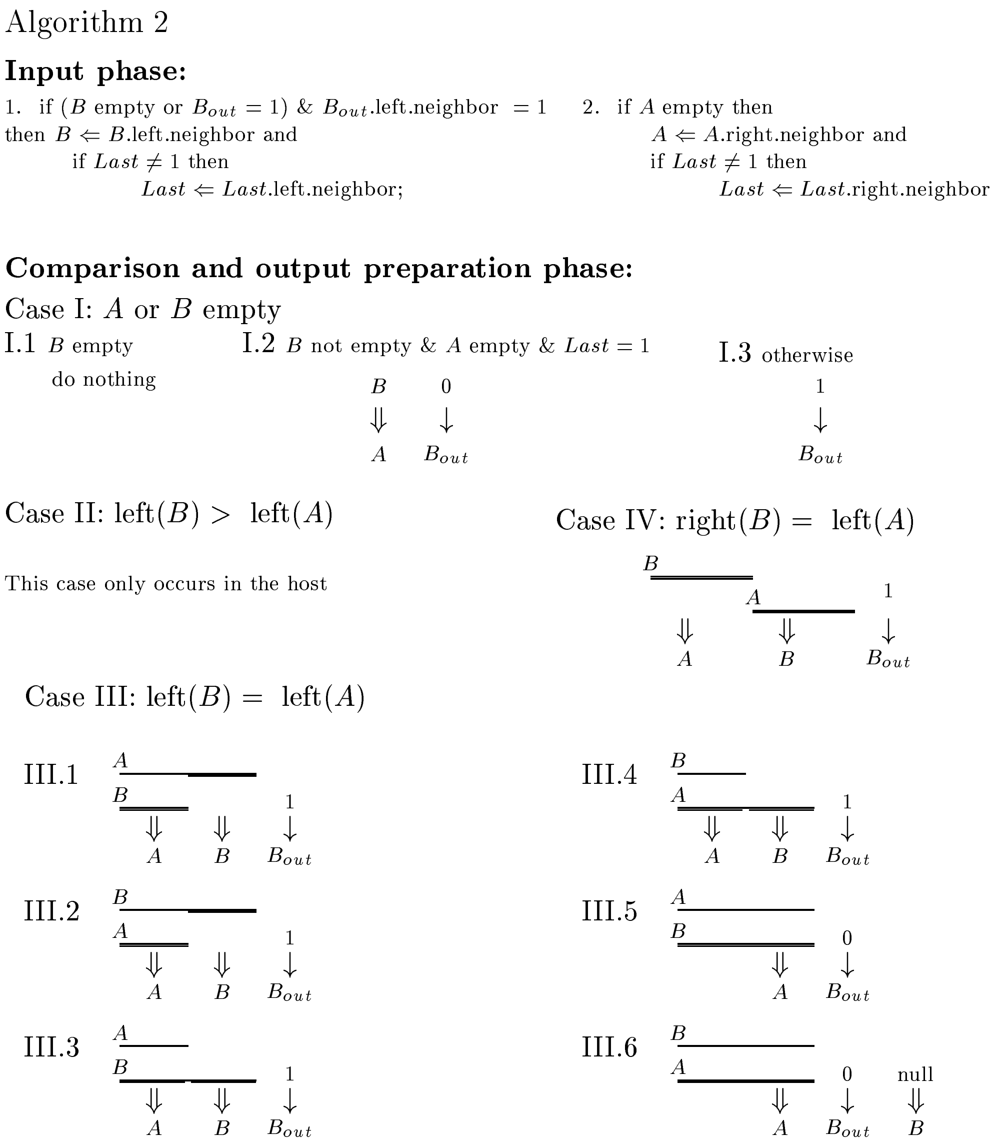

Figure 6: Cell algorithm: version 2

5.3 Correctness proof

We prove that this algorithm is equivalent to algorithm 1. Instead of giving a formal correctness proof, which is quite long, we will try to give the intuitive idea of how the transformation works. We still suppose the segments B enter one at a time into the array.

We can prove that if we input the same segments into two arrays, one with algorithm 1 (array 1) and one with algorithm 2 (array 2), then, between two segment input, the configuration of the two arrays are exactly the same except that there are no empty cells at the beginning of array 2.

¿From the complexity point of view we can also prove the following. When a segment empties j_1 cells (from J to $J + j_1$) in array 1, then when it is input in array 2, $2j_1$ steps later, the first cell is filled up by segment S_{J+j_1+1} (modified by B). We need $2j_1$ steps to propagate the "hole" and "bring back" the segment S_{J+j_1+1}.

In [9], we give an example where algorithm 2 is executed on a particular set of segments.

5.4 Complexity

In Algorithm 2, the space complexity obviously changes. It is exactly the maximum number of cells simultaneously full during the execution of Algorithm 2. (If we compare to Algorithm 1, the empty cells are no more counting in the space complexity.) It is easy to see that if the segments do not overlap each other vertically (right(S_i) < left(S_{i+1})), the algorithm uses only one cell and performs the output in $2N$ steps (because each segment waits one step before being input).

In general, we can say that the number of cells used by the algorithm is the maximum number of segments intersected by a "vertical" line. For example, in Figure 1, this maximum number is 3 (B_2, B_3, and B_4 can be intersected by a straight line parallel to the y-axis). Thus, on this set of segments, the algorithm will use 3 cells. This is quite easy to understand: it comes from the fact that, when B_3 is input, one can decide that B_1 is visible and B_1 is output. On the other hand, when B_5 is input, one can decide which parts of B_2, B_3 and B_4 are visible and can output these parts. Thus there are only 3 full cells at a time. As the algorithm is such that there is no empty cell at the beginning of the array, only the first 3 cells are used. We can give a bound for the time complexity. If we call $S(\text{Seg})$ the space complexity of the algorithm executed on the set of segments Seg, then $T(\text{Seg}) \leq 2N + S(\text{Seg})$. This complexity is better than the one in [11], in which $2N$ cells are used and at least $3N$ steps are needed (because $S(\text{Seg}) \leq N$).

6 Conclusions

We have presented systolic algorithms with the real-time characteristic of identifying visible segments on-the-fly, as input segments enter the systolic array. We have established a correctness proof for an initial version, which is then transformed to more efficient versions, while preserving the properties for correctness.

Versions 2 should be considered as the final version. By using version 2, we simplify the cell architecture and put a processing demand on the host, which should be able to match the speed of the systolic array.

The size of the systolic array $S(\text{Seg})$, where Seg is the input set of N segments, in both versions 2 and 3, is equal to the maximum number of segments crossed by a vertical line. $S(\text{Seg})$ depends of course on the particular scene. It can be as low as 1 and, in the worst case, it is N. Regarding the execution time, visible segments start to emerge as soon as input segments are fed into the systolic array. The total execution time is bounded by $2N+S(\text{Seg})$. Notice, however, that with the real-time characteristic, display of the already determined visible segments can be overlapped with the systolic algorithm execution, thereby shortening the overall execution time.

Acknowledgments

We would like to thank Yves Robert for suggesting this problem and many helpful discussions. The second author would also like to thank Michel Cosnard for the invitation to visit LIP-IMAG/ENS Lyon.

References

[1] M. Cosnard, P. Quinton, Y. Robert, & M. Tchuente (editors) *Parallel Algorithms and Architectures*, North Holland (1986).

[2] VanDam Foley, Hughes Feiner, *Computer Graphics Principles and Practice*, second edition, Addison-Wesley (1990).

[3] W.R. Franklin, M.S. Kanhalli "Parallel Object-Space Hidden Surface Removal", *Computer Graphics*, Vol. 24-4 (1990) pp. 87-94.

[4] H.T. Kung "The structure of parallel algorithms", *Advances in Computing*, 19 (1980) pp. 65-111.

[5] H.T. Kung "Why systolic architectures", *Computer*, 15 (1982) pp. 37-46.

[6] C. Leiserson "Systolic priority queues", Technical Report, Carnegie Mellon University, Department of Computer Science (1979).

[7] F.P. Preparata, M.I. Shamos "Computationnal Geometry, an introduction" Spinger-Verlag New-York (1985)

[8] P. Quinton, Y. Robert, *Algorithmes et architectures systoliques*, Masson, Paris (1989).

[9] T. Risset, S Song. A real time algorithm for on-the-fly hidden surface removal. Technical Report 91-13, LIP-IMAG, Ecole Normale Supérieure de Lyon, France, 1991.

[10] I.E. Sutherland, R.F. Sproull, and R.A. Schumacker "A characterization of ten hidden surface algorithms", *Computing Surveys* 6, 1 (1974) pp. 1-55.

[11] N.H. Vaidya, S.R. Das, P.C. Mathias, L.M. Patnaik "A systolic algorithm for scanline-based hidden surface removal", *Proceedings Third International Conference on Supercomputing* (1988) pp. 239-246.