Cellular Architectures and Algorithms for Image Synthesis

Michel Meriaux

Charge de recherches au CNRS

Laboratoire d'Informatique Fondamentale de Lille
UA 369 CNRS - Bat M3 - Universite des Sciences et Techniques de Lille Flandres Artois
59655 Villeneuve d'Ascq Cedex - France

1. Introduction

The aim of this paper is to provide some reflexions and partial results about cellular architectures for image synthesis and graphics. As some steps of image synthesis involve a long processing time, quite incompatible with interactivity, a natural solution consists in parallel processing. Though a lot of work has been done about cellular hardware, only a little exists about cellular graphic algorithms and hardware.

After a brief tutorial about cellular machines in section 2, we will first describe three architectures differing by their control and command schemes in section 3. Then in section 4 we are going to explain solutions for two basic algorithms matching those architectures. At last we are going to show in section 5, that existing components --such as Transputers-- or specifically designed ones enable us to conceive practicable cellular architectures.

2. Cellular machines

Though an interested reader will find in [COD68] and [VOL77] more detailed information, it is useful to remember here some principles of operation. Cellularity may be usually expressed in two ways: cellular automata and cellular logic arrays; their common feature is that they are made of identical elementary cells bound together.

2.1. Cellular logic arrays

Cellular logic arrays are made of identical logic cells, each one consisting of:

- internal or external (i.e. from outside the array) inputs
- outputs
- a logical function between the inputs and the outputs, usually combinatorial
- a memory function, if necessary

CLIP [DUF73] is a good example of such an architecture, here designed for image processing.

2.2. Cellular automata

This concept was first introduced in the 50's by Von Neumann. He described a universal cellular automata, with 29 configurations, able to calculate every computable function.

A cellular automata is based on a cellular space, i.e. a set of evenly distributed cells in an n-dimensionnal geometry. A configuration is the set of all cell states at a given generation. The neighbourhood of a cell is the finite set of directly connected cells. The transition function is the rules producing the state of a cell, assuming its state and the states of its neighbouring cells at the preceding generation. This function is the same for every cell; if it is univocal, the automata is said to be deterministic.

2.3. Cellular machines

It is necessary to extend the two previously described machines, so as not to be limited to synchronous behaviours. In our sense a cellular machine is a cellular automata but not tied to the concept of generation, i.e. synchronous parallel transitions. It includes asynchonous multiprocessor architectures.

3. Architectures for graphics

The theoretical cellular approach consists in processing one pixel per cell in a regular cell array. We will limit such an ideal approach to the case of a bidimensional square array with four neighbours, for two main reasons: first because it is the most simple approach, very compatible with existing VLSI; second because extensions to 3D or more connected arrays look easy, from a theoretical point of view.

We can divide this class of architectures according to some criteria of processing. The first one is the complexity of every cell, which may extend from simple logic circuits to complex microcomputers. There already exists some examples of cellular logic arrays, almost always dedicated to specific tasks such as $Ax+By+C$ calculations (Fuchs machine - [FUC81]) or area filling; the complexity of every cell is low, but that does not fit in with what we intend to do, i.e. a more versatile and flexible solution we will only manage to find in more complex logic circuits or programmable ones.

A more important criterion is control and command distribution inside the array. It defines three classes of architectures we are going to describe now. As far as image data are concerned, we suppose here that data are read to or written from video units through dual port memories in every cell.

3.1. SIMD architecture

At one end of the classification we find a completely synchronous and host-controlled array of slave cells, each cell executing the same instruction at the same time (figure 1). In that class we can also distinguish cooperating or non-cooperating cells. We will show later on that this way of working is not very compatible with a lot of graphic algorithms which are not strictly parallel in nature, but involve some communication of commands and data from cell to cell. We suppose that every cell receives, by a way to be defined later, the command to be executed. It executes the command, maybe with the help of some preexisting data coming from the connected neighbours. Then another command is sent by the host and executed.

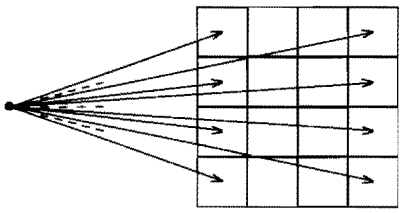

Figure 1: SIMD architecture.

We can separate here the problem of command broadcasting all over the array and the execution of it. It is not necessary to have a completely synchronous behaviour; for instance the architecture we have implemented follows such rules as these:

- commands are injected from the sides of the array and transmitted towards the centre by a very simple routing algorithm, i.e. every cell receiving a command sends it again to its neighbour towards the centre
- each cell can execute the command, as soon as the transmission has been completed

The processing time for one command is composed of two well defined parts:

- the first one deals with the propagation through the array, which can be assessed at $\frac{n}{2} * t$, where n is the size of the array and t the cycle time for one elementary transmission
- the second one deals with the execution time, which can generally be maximised for a given command.

Let us notice at this point that command pipelining may be used as long as execution time is much lower than propagation time.

3.2. Data driven architecture

It is often possible, in the field of computer graphics, to foresee some special points where data are obvious and to elaborate incremental algorithms based on those initial points.

The architecture based on those remarks is very simple to explain: you start with cells where you know the results and then propagate new commands throughout the network, depending on the data (figure 2). The most common example is line drawing, based on Bresenham's algorithm, which will be detailed later.

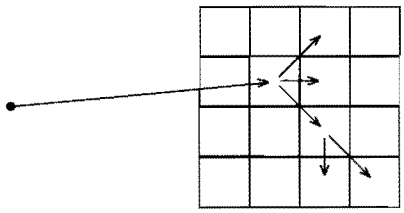

Figure 2: Data driven architecture.

Two very important problems occur with this approach:

- the first one deals with the strong hypothesis that we are able to reach every cell straight away. This is possible either with an adressing scheme over the array or a routing protocol. These two solutions are not very satisfactory and we would offer a third one depending on the technology: it would consist in having access to all the cells in parallel, via a third dimension, which could be optical, and then indicating the address, every cell being supposed to know its own address. (This also applies in 3.1)
- the second one deals with the fact that for some algorithms such as area filling, we are not able to foresee the execution time and consequently to schedule tasks; this problem cannot be solved easily.

3.3. Multi pipeline architecture

A Multi pipeline architecture consists of an array of identical cells (figure 3) only accessible by one side, for instance the left one. Each cell is asynchronous and can perform some basic operations we will talk about later.

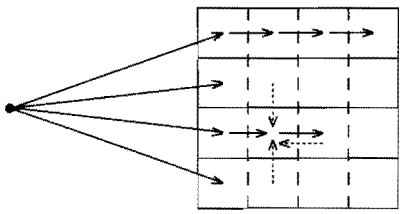

Figure 3: Multi pipeline architecture.

Each cell of the host column is filled with correct values and send a command and data to the first column of the array. A cell can read its four neighbours but write only to its right one in order to make it work, i.e. by sending it a command. The activity of a column progresses from the left to the right at the speed of the basic cell operation. Obviously different activities can be piped through every line of the array.

4. Algorithms

We present now different solutions to two basic problems, each solution dealing with one of the previously described architectures. It should be noticed here that we are studying numerous other algorithms, such as area shading or ray tracing, but we think that explanations should be given for basic characteristic ones.

4.1. The line drawing problem

4.1.1. SIMD solution

This is a trivial problem: we only have to solve the equation of the line in every cell lying inside the rectangle limited by the two points. Thus we are obviously able to obtain information very useful for antialiasing, because $E = Ax + By + C$ is nothing else than the distance between the pixel and the true line.

4.1.2. Data driven solution

This is also a simple problem: given the two starting points, they have to choose which of their neighbours will be elected for the next step (figure 4). The Bresenham's algorithm is to be used in such a case. We can notice that two processes are working in parallel and that the execution time depends of vertical or

horizontal distance between the two starting points.

As Bresenham's algorithm calculates the error between the ideal line and the chosen pixel, we are also able to handle antialiasing without extra cost.

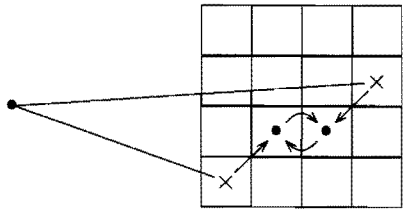

Figure 4: Line drawing in data driven architecture.

4.1.3. Multi pipeline solution

In case we draw a line from A to B, we have to access the host column between y_A and y_B and calculate the initial values of every host-cell with the formulas:

$$a = y_B - y_A$$

$$v = y*(x_B - x_{A)} + x_A*y_B - x_B*y_A$$

Then we program every cell of every line to execute:

receive v,a
$v := v + a$
if $[v] \leqslant ½$ then OK
transmit v,a

We will not talk about special cases of drawing here: the most important is that we can draw distinct segments simultaneously and any segments in pipe-line.

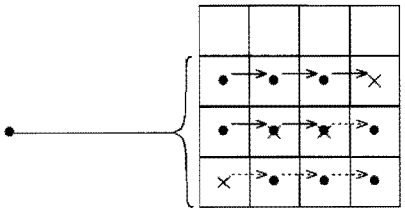

Figure 5: Line drawing in multi pipeline architecture.

4.2. The area filling problem

4.2.1. SIMD solution

There exists a good solution for convex polygons, involving computations of the position of every cell against every border: if a cell is on the right side of every border then it belongs to the inside (figure 6). This is not very satisfactory indeed, because it needs a lot of time.

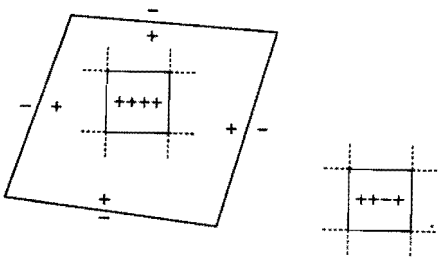

Figure 6: Area filling: "inside" test; one cell is inside and one cell is outside the convex polygon.

4.2.2. Data driven solution

A better solution is given by a Life solution, derived from Conway's algorithm: given the border, previously written in the array, and one or several inside points, we only have to propagate this "inside" information in the four directions until we reach the border, while storing in the encountered cells a bit indicating it has been visited (figure 7). As long as the border is topologically closed, the algorithm works well, whatever its shape.

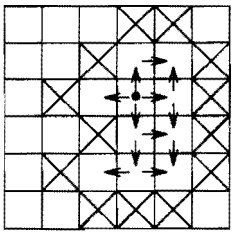

Figure 7: Area filling: propagation of "inside" information in four directions.

4.2.3. Multi pipe-line solution

The line drawing algorithm extends easily to the area filling problem: instead of only marking with "OK" the edge of a segment, we "shadow" it to the right or to the left according to a previously determined direction and use a simple boolean operator to obtain the inside of a polygon (figure 8).

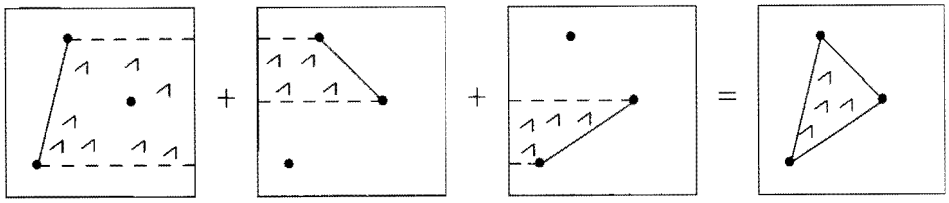

Figure 8: Area filling by casting "shadows" followed by a boolean operation.

A great number of basic algorithms can be implemented very easily this way. Obviously such an architecture is not designed to handle high level algorithms, such as matrix products for 3D transforms.

5. Implementation and performance evaluation

Most of our simulations have been made using a network of Transputers, driven by a Development System based on a PC. This network consists of an array of 1*1 to 4*4 Transputers, connected to a master one (figure 9). The language we use is obviously OCCAM, which enables us to handle parallelism and synchronisation very easily.

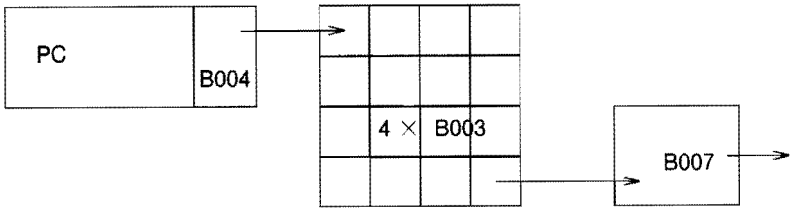

Figure 9: Network of transputers as used for most of the simulations.

5.1. SIMD

The main problem here is that the cells have to be very complex to handle very simple problems such as circle drawing or area filling. They involve multiplications and divisions, maybe for floating-point numbers. It sounds incompatible with a realistic VLSI implementation. On the other hand it looks obvious that a lot of cells are working for no useful result.

However a simulation has been led on our array of transputers, the results of which are really bad: one bottleneck appears quite clearly, due to the command propagation throughout the array, which involves a lot of communications.

5.2. Data driven

Based on incremental or local propagation laws, the cells may be very simple. A software simulation has been conducted, which shows however that access and termination problems are not at all academic.

If we have to route the initial commands through the array (and we cannot evaluate the end of a process) then we loose a lot of time doing nothing. Besides the resulting parallelism is not high at all.

5.3. Multipipeline

As we were not very fond of specific Silicium, we first tried to implement and evaluate a prototype on our (reduced) set of Transputers with the following algorithms: line drawing, area filling, circle drawing, smoothing. The last one involves more complex communications between the cells because each cell has to know about more than 4 neighbours to evaluate its own data. So our model has been somewhat extended to handle more complex situations than the easiest one which is the left-to-right mono-cell.

We also developed our model to take advantage of the Transputer, especially from the communication point of view: our model privileges the left to right communications, thus underusing the others. We implemented very easily, because of the parallel nature of the Transputer, four identical processes in the four directions of the array, thus optimizing and balancing the load of communications all over the array. The only disadvantage is that the jobs must be carefully scheduled between the four ways so as not to interfere.

5.4. Comparisons

It is difficult to compare our simulations, because we must take different things into account. What we can say at the moment is that SIMD is easy to program, involves complex cells, and some algorithmic problems are not solved. As far as Data Driven is concerned, it is obvious that the termination problem is the most important, but it seems that only area filling is concerned. Multipipeline is the architecture we are developing now, and the first results we obtained on Transputer look very interesting. We are now extending our simulations, which are in fact real

implementation on a reduced array, in order to see whether it can support real data (i.e. a great number of lines or polygons to display).

6. Conclusion

We think that nowadays such a huge array of cells is no more unthinkable: a dedicated cell would consist of no more than thousands of gates. So we could integrate about 16*16 cells on one chip and a complete 1K*1K array would consist of 4K identical components. If we choose to implement such an array with a more complex but existing component --such as the Transputer-- it is necessary to make every cell manage and process a sub-array of pixels because of their cost. The choice of the rules associating a physical processor to a set of pixels must be carefully studied, in order not to create either communication or task scheduling bottlenecks. An array of 32*32 Transputers would have a theoretical power of 1000 Mips very compatible with our problem.

At last, let us add that a lot of problems have to be discussed, such as:

- precise evaluation of the host processor
- data segmentation to build independant groups of simultaneously processed jobs
- fault tolerance and reliability
- other relevant architectures, such as pyramidal or oct-tree ones.
- more connected architectures, diminishing the need of routing and simplifying the algorithms.

7. Acknowledgments

We thank all the members of the Graphic Research Group of the Computer Science Laboratory of Lille and the Referees for the good remarks they made about this paper, which I hope I have taken into account.

8. Bibliography

[COD68] Codd E.F., "Cellular Automata", Academic Press, New York (1968)

[DUF73] Duff M.J.B., "A Cellular Logic array for Image Processing", *Pattern Recognition* **5** (1973)

[FIS86] Fisher A.J., "A multi processor implementation of OCCAM", *Software Practice and Experience* **16** 10 (1986)

[FUC81] Fuchs H., Poulton J., "Pixel-Planes, a VLSI-Oriented Design for a raster graphic engine", *VLSI Design* 3rd Q (1981)

[HEM86] Hemery F., "Une architecture cellulaire multi pipe-line a base de transputers." *Memoire de DEA* Lille (1986)

[MER84] Meriaux M., "Contributions a l'imagerie informatique, aspects algorithmiques et architecturaux." *These de Doctorat d'Etat* Lille (1984)

[PEL85] Pelerin M., "Synthesed'imageset parallelisme: algorithmes et architectures" *These de doctorat* Universite de Lille

[SEI85] Seitz C.L., "The Cosmic Cube" *Communications of the ACM* **18** 1 (1985)

[VOL77] Vollmar R., "Cellular Spaces and Parallel Algorithms - an introductory survey." *Parallel computers, parallel Mathematics* IMACS77