DisArray : A Graphics-Oriented Fifth Generation Workstation

Published in Proceedings of NicoGraph83,
Nippon Computer Graphics Association Conference,
Tokyo, Dec. 1983.

Ian Page,
Computer Systems Laboratory,
Queen Mary College,
University of London

Abstract

This paper describes a hardware solution to the problem of updating bitmap displays very rapidly. Starting from requirements of the man-machine interface of a powerful personal computing system, it is seen that it is essential to have hardware support for the 'RasterOp' graphics algorithm which operates on two-dimensional (rectangular) bitmaps. The DisArray hardware employs an array of 256 1-bit processor/memory pairs in a 16×16 configuration, thus directly supporting the manipulation of two-dimensional bitmaps.

A separate, conventional, bit-slice processor provides the array instruction sequence to execute RasterOp and other graphics algorithms (such as polygon filling). Video data is asynchronously extracted from array memory using a double buffering scheme.

The system can be used in a 'display list' mode in which DisArray continuously interprets an applications-oriented data structure which describes the screen layout in terms of windows, text strings and pictures. This operation is fast enough to support animation of a complex screen layout with no overhead whatever on the applications software which only operates on the high-level intermediate data structure.

In future enhancements to the current system we hope to increase the processing speed still further and to add real-time video data input into array processor memory. This would allow us to perform some image processing and pattern recognition tasks and assess the machine's relevance for these tasks also. We also wish to investigate the use of the array processor as a host for functional programming environments.

1 Introduction

The user interfaces of complex information processing systems such as computer-aided design, computer-based training and automated office systems are rightly receiving an increasing amount of attention from computer scientists. However, many of the new techniques being employed in user interfaces such as dynamic windows and menus, dragging and animation all require enormous computational power, often from the host processor. The software structures needed to run such a user interface can easily become extremely complex in an attempt to cater for the entirely natural desire of the user to apply any operation that he knows about, in any context that he happens to be in. Indeed, two of the most important aspects of a 'good' user interface are that it should have a small number of 'orthogonal' commands and that there should be as few 'modes' as possible.

Many advantages can be gained from using a *display list* as an interface between the applications programs and the screen image. The display list is a high-level data structure stored in the address space of the host computer which represents the screen image in terms of windows, groups of windows, strings of text, bitmap pictures etc. The separate (often concurrent) applications programs can then interact with the data structure very easily. Moving a window, for example,

becomes a simple matter of changing the x and y co-ordinates of the window in the display list and changing the z-order of the windows might only involve re-ordering a window list.

This approach *considerably* simplifies the task of constructing complex interactive systems but it also increases still further the amount of processing power needed to support the screen image. We now need to continuously refresh the screen image from the data structure at a rate of up to 25 times per second and this clearly needs special-purpose hardware support.

A short investigation showed that virtually all of the computation time is spent in a software procedure often known as 'RasterOp' [Reference 1]. This operates on rectangular sub-areas of bitmaps and can move and logically combine such bitmaps. Despite the fact that it is conceptually very simple, RasterOp is an extra-ordinarily powerful procedure and can even perform many operations which are not intuitively obvious (such as rotating pictures and doing fast scalar arithmetic! [references 5, 7]). We decided to support this procedure and also the display list interpretation with special purpose hardware. The hardware system that has resulted from this we fondly know as 'DisArray' (short for **Dis**play **Array**).

DisArray uses a two-dimensional interconnected array of very simple computational elements each one paired with its own local memory. The DisArray hardware has the ability to manipulate two-dimensional areas of a bitmap (up to 16 $\times$16 bits square) in a single memory/processor cycle. DisArray can use this ability to form new screen images in a very short time by rapidly assembling the components of the picture (individual characters, line segments, half-tone shaded areas, whole windows etc.) into a screen-sized bitmap which can then be displayed on a conventional C.R.T. monitor.

This approach has the advantages of high speed since the memory and computational elements are very closely coupled, and of high throughput because of the high degree of parallelism and the two-dimensional interconnections within the array.

2 A Quick Introduction To Rasterop

As noted above RasterOp operates on rectangular arrays of Pixels (Picture Elements). An over-simplified Pascal version of the RasterOp algorithm is in Fig. 2. This version ignores the problems caused by having overlapping source and destination rectangles and assumes that pixels are only ever Black or White (false or true). The data type 'raster' is not defined, but is essentially a two-dimensional array of Boolean variables corresponding to a picture. The rasters are only manipulated through the access functions 'GetPixel' and 'SetPixel' which are not defined here but have the obvious meanings. The 'operation' parameter has the effect on the destination rectangle given by Fig. 2.

This would be a hopelessly inefficient implementation of RasterOp, but it does effectively show the nature of the algorithm and the sophisticated versions still retain these same essential features. Also, this version has only some of the 'operations' that a real implementation might offer. Fig. 3 gives a simple pictorial example of this version of RasterOp in which an 'OR' operation would add a house and another tree to the one already in the destination rectangle. Further details of the algorithm can be found in reference 1.

3 Display List Approach

The display list is an abstraction of the image to be presented on the screen to the user. It acts as an interface between the application programs and the refresh process which has to present a view of the display structure on the screen. The design of the display list is therefore a crucial

1. : all black
2. : all white
3. : same as source rectangle
4. : inverse of source rectangle
5. : logical AND of source and destination rectangles
6. : logical OR of source and destination rectangles
7. : logical EXOR of source and destination rectangles

Figure 1: The 'operation' parameter for RasterOp.

```
procedure RasterOp (operation : integer;
   var Destination : raster;  xd, yd, width, height : integer;
   var Source      : raster;  xs, ys : integer);
const White = false;  Black = true;
var x, y : integer;
begin
  for y:=1 to height do
    for x:=1 to width  do
      case  operation  of
        1 : SetPixel (Destination, x, y, Black);
        2 : SetPixel (Destination, x, y, White);
        3 : SetPixel (Destination, x, y, GetPixel (Source, x, y));
        4 : SetPixel (Destination, x, y, Not GetPixel (Source, x, y));
        5 : SetPixel (Destination, x, y,
               GetPixel (Source, x, y)  And  GetPixel (Destination, x, y));
        6 : SetPixel (Destination, x, y,
               GetPixel (Source, x, y)  Or   GetPixel (Destination, x, y));
        7 : SetPixel (Destination, x, y,
               GetPixel (Source, x, y)  <>   GetPixel (Destination, x, y));
      end
end;
```

Figure 2: The RasterOp algorithm (simplified).

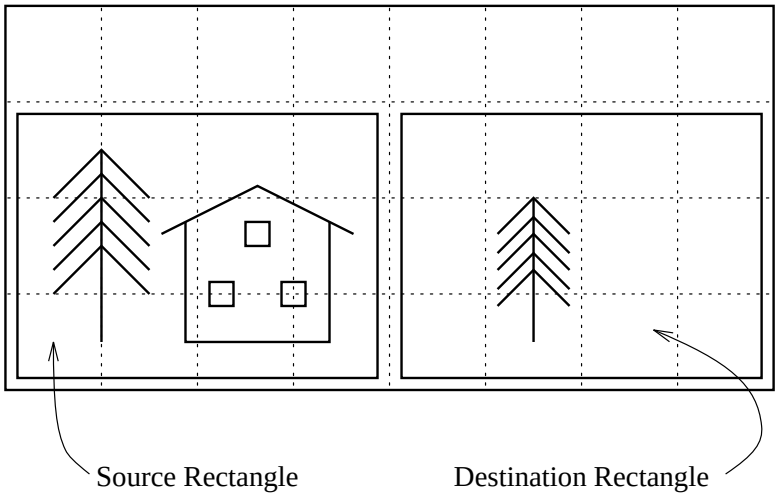

Figure 3: A simple example of the RasterOp primitive.

issue, since it is at the heart of all the display operations and these need to be performed quickly. Also its structure needs to be appropriate both for the different applications programs and the display manager. This always means that trade-offs have to be made since the requirements of applications programs and the refresh process are significantly different from each other.

A simple example of a display list in which the major structural elements correspond to 'windows' (or 'pieces of paper') on the screen is shown in Fig. 4. The whole structure is a list of window descriptors which each store the attributes and the contents of one window. The order of the windows in the display list is the same as the z-ordering of the windows when shown on the screen. Thus the screen image can be generated using the painter's algorithm (ie. back-to-front fill) by traversing the list from beginning to end, filling in the contributions to the screen from each window in turn. Fig. 5 shows the screen image which corresponds to the display list in Fig. 4.

In this simple example, the windows contain only text and so the data structure for each window includes a set of text strings which comprise the text to be put in the window. An application program operating on one of the windows can now be given a pointer to 'its' window descriptor and can move it on the screen by changing the x, y offset values in the descriptor. It can similarly change the background colour of the window or its size or the fount of the text by altering the descriptor appropriately. The text in the window can be scrolled by simply cycling the pointers in the array of text string pointers.

The display processor needs to continuously scan this display list and generate a screen image from it. It can be appreciated that this process involves little more than scanning the display list in order, performing some simple clipping operations (to the screen boundary for instance) and then performing the RasterOp operation on whole windows (to clear them to their background colour) and then to use RasterOp again for each of the characters in the text strings to copy the bitmap images of the characters from a fount table into the window area. Thus we need a relatively fast machine to traverse the display list and a super-fast machine to implement the RasterOp commands generated from the display list interpretation.

As a historical note, around 1977 at QMC we built a hardware system known as the *QMC Text Terminal* [reference 8] which used exactly this style of display list and generated 50 completely

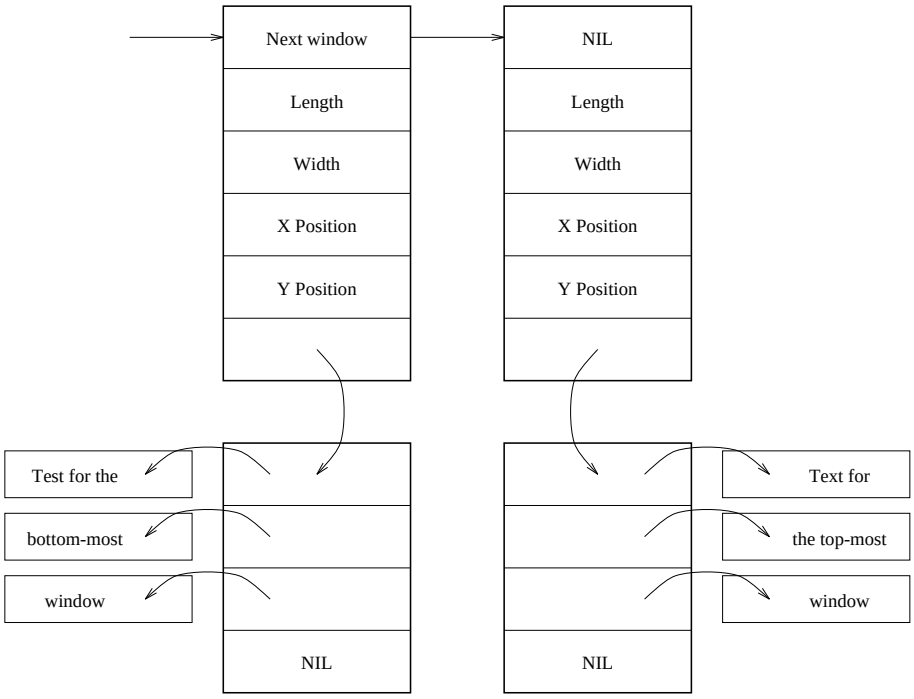

Figure 4: An example of a simple Display List.

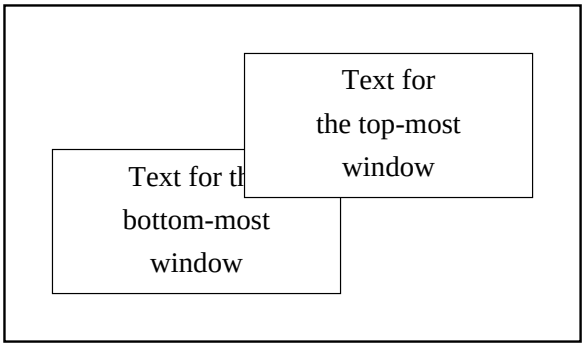

Figure 5: Screen appearance of the Display List of Fig. 4.

new frames per second using a purpose-designed bit-slice processor. It produced a *fully-animated, colour display* of a desk-top with 'pieces of paper' but was restricted to showing only textual and block graphics information. A large part of the motivation for the DisArray project stemmed from the success of the Text Terminal experiment and our desire to have a system which would perform similarly with arbitrary raster-graphics images.

4 Disarray And Rasterop

Despite its simplicity, RasterOp is an extraordinarily powerful primitive and, when it is well-supported in the hardware in such machines as the Alto [Reference 2] and the Perq [Reference 3], it gives those machines a very good interactive graphics capability. However, we wish to improve significantly on that performance by employing parallelism in the hardware. We would also like to find a form of parallelism which can offer even greater throughput by simply increasing the amount of parallelism in the hardware.

In its simplest form on a conventional machine, the inner loop of RasterOp would take a *word-aligned* word from store (part of the source) and move it to a *bit-aligned* word in store (part of the destination). This bit-aligned word will usually fall across the boundary of two store words and thus require two read/modify/write cycles to update it. This effect slows down the algorithm but it can be ameliorated by pipelining the data over a number of such inner loop cycles.

To improve their graphics speed, machines such as the Perq have a micro-programmed RasterOp. Also, they make special arrangements to increase memory bandwidth (by employing wide data highways), and have a barrel shifter to do the alignment to bit boundaries. This is probably about the limit of support that a conventional processor can offer to RasterOp, but there are (always) good reasons for wanting to increase its speed still further. It should also be noted that such a conventional machine would always be very much better at doing short, fat RasterOps than tall, thin ones. Such an implementation will perform most poorly (in terms of pixels/second updated) when drawing a pixel-wide vertical line, when only one bit is being usefully updated on each memory cycle. Paradoxically, in this particular case, this bandwidth degradation only gets worse as the machine data paths are made wider in an attempt to improve the general throughput of RasterOp.

In fact, in the graphics world, there is no reason why RasterOp rectangles should be of any particular shape and the theoretically most efficient shape for the basic *word* is thus a square, which is equally optimised for both worst cases of the tall, thin and the short, fat rectangle.

DisArray uses such a square word, known as a *plane,* to support RasterOp. The analogous inner loop of a DisArray RasterOp, takes a (plane-aligned) plane from memory and moves it to a bit-aligned plane, which will usually fall across *four* actual memory planes. A series of diagrams Fig. 6 show one possible sequence of DisArray operations to perform a single step of the inner loop of a RasterOp operation.

5 A Simple Example

In the example shown in Figs. 6-11, the letter 'A' is to be copied from its current position in a fount-table to a position within the portion of DisArray memory which is (currently) being refreshed onto the screen. The user would thus see the word 'DISARRAY' being completed on the screen Fig. 6). DisArray has a 16 $\times$16 register, known as the Q-Register. Fig. 7 shows the contents of the Q-Register after a single memory cycle which loads the letter 'A' from the appropriate source plane. During this same cycle, the Row and Column masks shown in fig

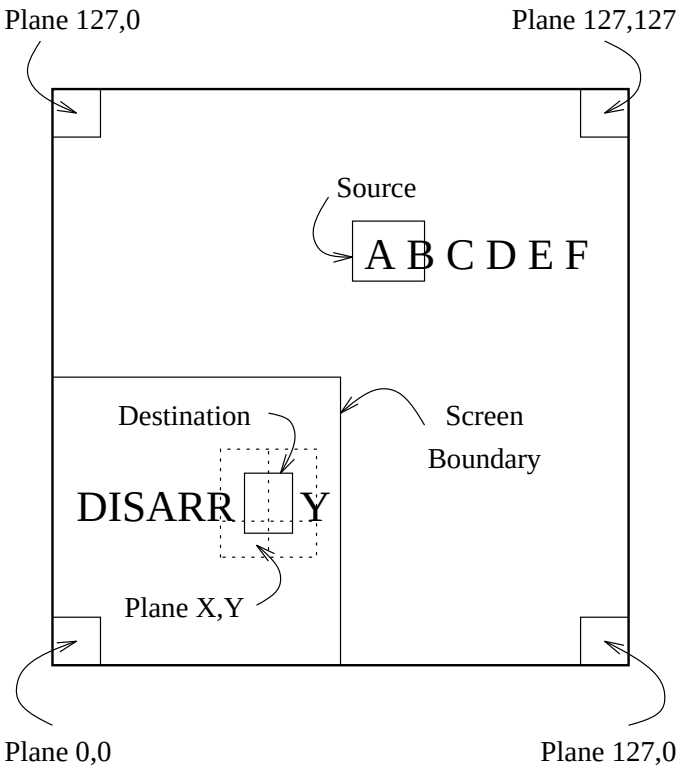

Figure 6: The Whole DisArray address space regarded as a single bitmap. The word 'DISARRAY' is being completed using a particular bitmap-defined fount.

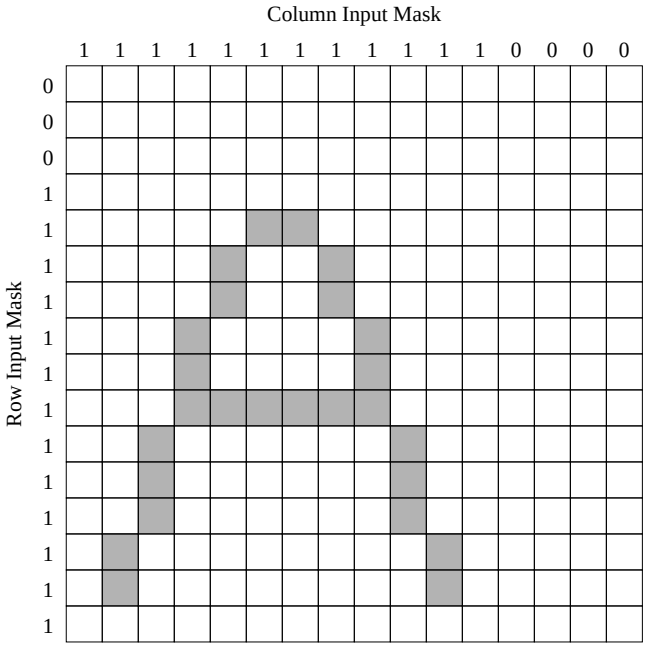

Figure 7: The Q-register after loading from store under the mask shown. The mask has suppressed part of the 'B' from the Source plane

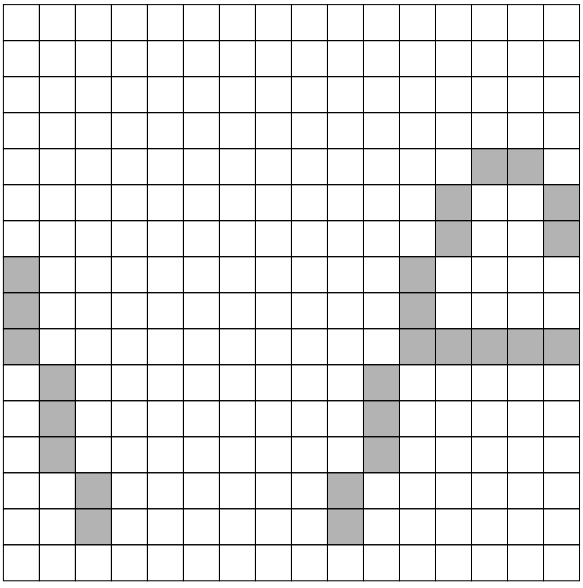

Figure 8: The Q-register after circulation in the East-West direction.

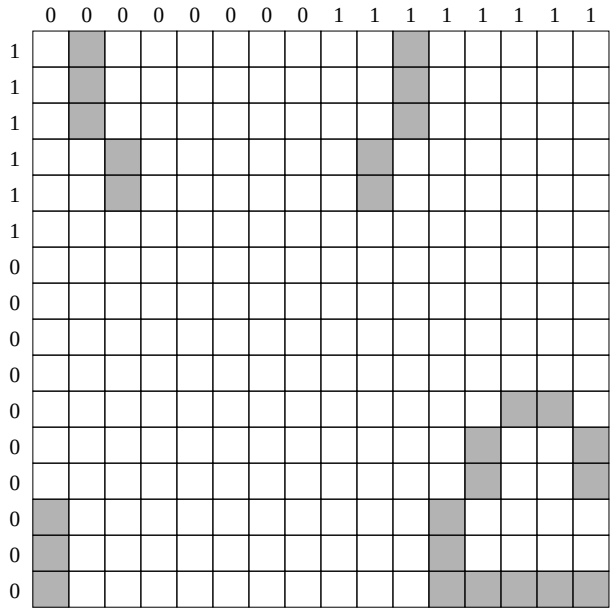

Figure 9: The Q-register after circulation in the North-South direction. Shown with a quadrant mask.

1b are ANDed bit-wise in each processor to obtain a 16 ×16 bit mask. This mask defines the limit of validity of the source region within this source plane. A single 'read' operation will read the source plane from memory, AND it with the 16 ×16 mask and then store the result in the Q-register.

Using the nearest-neighbour connections between the processors, the Q-Register is shifted by appropriate amounts, first in the east-west direction (Fig. 8) and then in the north-south direction (Fig. 9). Fig. 9 also shows the state of the Row and Column masks which will be needed for the subsequent operations. These operations are on the four separate 'quadrants' of the letter 'A' which lie in the four corners of the Q-Register. Notice that the logical AND of these two masks selects the bottom left-hand corner of the 'A' (now in the top right-hand corner of the Q-Register). The other quadrants are selected by appropriately inverting the row and/or column masks.

Fig. 10 shows the state of the four destination planes after a single read/modify/write cycle involving the lower left of these four planes. The shifting performed above has resulted in the bottom left-hand corner of the 'A' being correctly aligned with the top right-hand corner of the destination plane where it eventually needs to go. A single machine cycle reads the previous state of the destination plane and writes it back unchanged, except in the area designated by the quadrant mask, where the contents of the destination plane are replaced by the logical OR (in this case) of the corresponding part of the Q-Register (the source) and the previous contents of the destination plane.

Fig. 11 shows the state of the destination planes after a further 3 read/modify/write cycles and the desired operation has been completed. With slight variations, this sequence can be repeated many times to deal with source rectangles consisting of many planes and this example is thus representative of the inner loop of a generalised RasterOp.

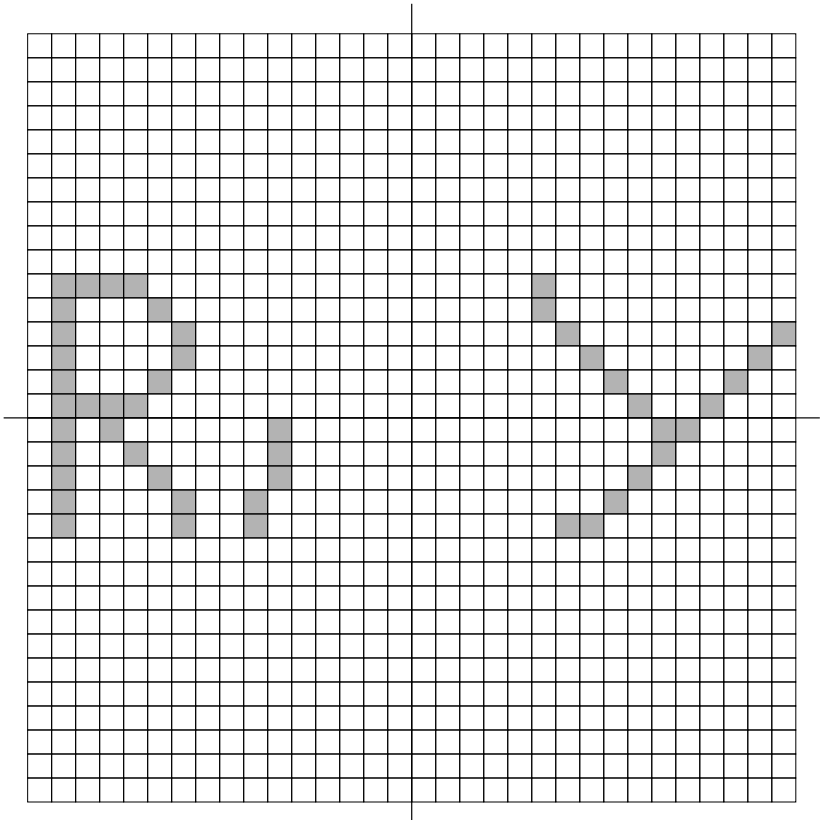

Figure 10: The four destination planes after a 'read', 'OR with Q-register', 'write' cycle (under the quadrant mask) to plane m.

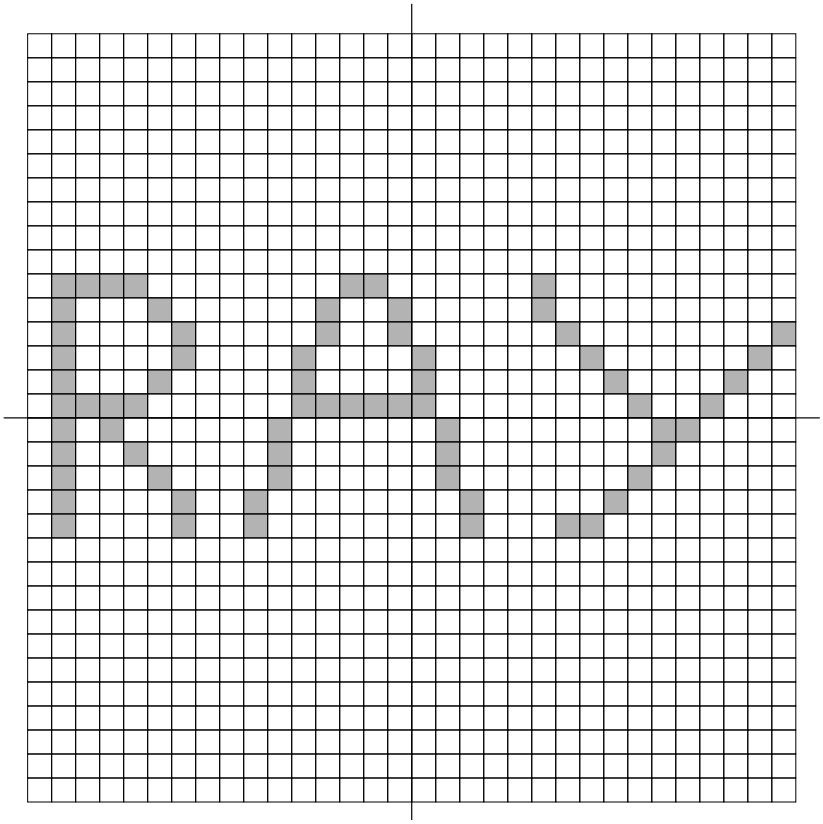

Figure 11: The four destination planes after a three more similar cycles with appropriately altered addresses and inverted row/column input lines.

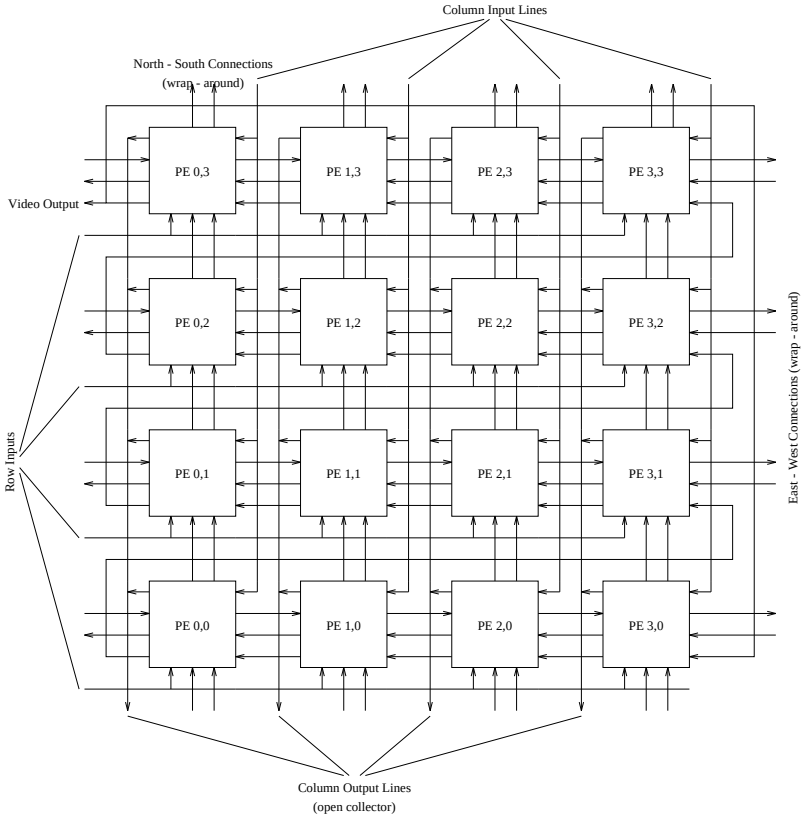

Figure 12: A 4×4 DisArray.

6 The Array Processor

The DisArray hardware has an array of 16 ×16 Processing Elements, which are each simple 1-bit processors, each having a 16K×1-bit local store. The array as a whole therefore deals with 256-bit square words, known as *planes.* All Processing Elements execute the same instruction simultaneously on their local data, making it an *S.I.M.D.* (**S**ingle **I**nstruction stream, **M**ultiple **D**ata stream) machine. The Processing Elements each have connections to their four nearest neighbours in the array so that planes can be shifted bodily in the four orthogonal directions, one bit position at a time. The edge connections are toroidal so that the shifting is in fact circular in the two dimensions. The architecture is similar to the I.C.L. Distributed Array Processor [Reference 4], which partly inspired this project.

The Array Control Unit turns a set of RasterOp parameters into an appropriate sequence of array operations to implement the 'call' of RasterOp. An outline diagram of the array system configuration is given in Fig. 12 and Fig. 13, but with a much reduced size of array for reasons of clarity.

A host processor runs the application programs which produce the calls to the RasterOp procedure. We are using a PDP11 running Unix as the host system, but we intend to build ourselves a faster and more appropriate host in the near future.

The resulting RasterOp parameters can be directly interpreted by the array control unit or, in another model of screen updating, can be stored in an application-accessible display list which

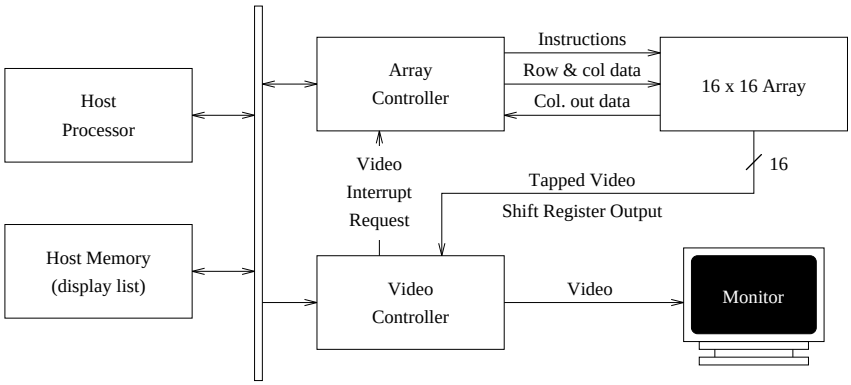

Figure 13: DisArray : overall structure.

represents the current state of the screen.

Using this *display list model* the Array Control Unit would autonomously and continuously interpret this display list to produce a sequence of (double-buffered) screen images. If this operation can be supported at a sufficiently high rate, then there will be a high degree of decoupling between application programs and the screen and also between different application programs which need to share access to the screen.

6.1 The Processing Elements

A block diagram of a single Processing Element is shown in Fig. 14. The processing is done by the ALU, which is an 8:1 multiplexor. This, in effect generates an arbitrary function of the Q-Register output and the memory output. However, one of two such arbitrary functions is selected on the basis of the logical AND of a row-derived and a column-derived input line. Together, these are normally used to select an arbitrary sub-rectangle of the array based on one or other of the corners of the array. Such sub-rectangles are known as *quadrants.*

Two 16-bit strings, each consisting of a single group of consecutive 0's and a single group of consecutive 1's, applied to the row and column input lines are sufficient to define a single quadrant (ie. that area where both Row and Column inputs are 1). The other three similarly-aligned quadrants can be selected by inverting the bits in either one or both of these bit strings.

This arrangement allows some arbitrary function to be performed in the region of the selected quadrant. Often, however, the rest of the processors outside of the quadrant will have to perform some simple (non-useful) operation such as copying whilst this is going on.

There is a single 1-bit register (the Q-register) which holds the result of computations and is also the holding register for array shifting operations. The design has optimised the nearest neighbour shifting by putting only the Q-register and a 4:1 neighbour selection multiplexor into the shift data path. We hope to enhance the system at some future date with one or more additional Q registers.

The local store is a 16 Kbit dynamic RAM whose control inputs (including the address lines) are derived from the Array Control Unit and whose data input is fed from the output of the ALU. A 16-bit output bus from the array is generated from 16 sets of 16 open-collector outputs in each column of the array being wired together. By selecting only one row of the array, using the row and column input lines, a single 16-bit word can be output from the memory to this bus. This

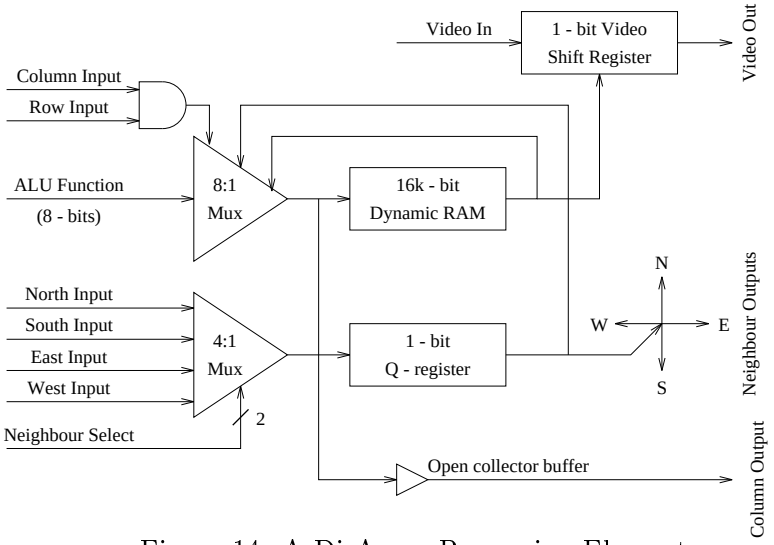

Figure 14: A DisArray Processing Element.

is used to map the DisArray memory into the host processor's address space. Similarly, 16-bit data from the host can be written into array memory by putting the data on the column input lines and selecting just one row using the row input lines.

As a bonus, the open-collector bus can be used to support some simple host-accessible *content-addressing* of memory. For example, arranging 16-bit words column-wise in store and broadcasting a 16-pattern along the row input lines the host can look for a match in any one of the 16 columns simultaneously, the corresponding column output line saying whether a match was found.

The control signals for the Processing Elements are provided by the Array Control Unit and are copied identically [1] to each processor. This set of control signals thus makes up a single *array instruction.* This instruction consists of :

1. The memory (plane) address to be used.
2. Memory control signals (RAS, CAS etc).
2. The two Boolean functions for the ALU.
3. The nearest neighbour selection.
4. Q-Register latch signal.
5. Multiplexor enable signals.
6. Miscellaneous edge control signals.

6.2 The Basic Array Cycle

The most general type of array instruction is one involving a read/modify/write operation on a single plane in memory. Such an instruction performs the following functions:

- Reads the contents of a 256-bit plane from memory.
- Examines the state of the row and column input lines. These usually contain masking information and the logical AND of these inputs is generated to select an quadrant.

[1] In fact, the memory address to each processor is in fact systematically altered under both the address-staggering and the quadrant-addressing schemes outlined later.

- Applies an arbitrary Boolean operation between the contents of the plane and the Q-register in the region of the selected quadrant and applies another arbitrary Boolean operation outside that quadrant.
- The result of the computation cycle c) is optionally written back to the same location selected in a), and/or latched into the register and/or sent to the column output lines. This read/modify/write cycle is terminated early whenever possible.

6.3 The Array Control Unit

The Control Unit has the task of generating the instruction stream which controls the array. This instruction stream is generated either from interpreting the display file or by directly executing RasterOp procedure calls from a process running in the host processor (usually a screen manager process). Since the control unit is micro-coded, both of these models of operation and many others are possible, simply by re-loading the micro-code store.

The control unit consists of an AMD29116 16-bit datapath chip, a 2910-based sequencer and a 4k $\times$32-bit writable micro-code store. It can execute independently of the array processor but is the sole means of initiating an array processor cycle. It has DMA access to the memory of the host processor and this is used to give the controller read-only access to the display list. All of the array processor registers can be directly written by the controller in order to set up array instructions and edge data. Additionally, there is a 4k $\times$16 local cache store which can hold working variables and copies of parts of the display list needed during interpretation.

6.4 The Refresh Controller

The refresh controller autonomously takes a bitmap from store by stealing array memory cycles. This produces 256-bits of data which is then broadside-loaded into a spirally-arranged video shift register which runs through the whole array. This data is then asynchronously clocked out directly to the monitor at video speed.

There is will always be a fundamental mismatch between the requirements of RasterOp and those of video refresh; the former needs to operate on square words and the latter needs to operate on very long thin words (scan lines). DisArray has an *address staggering* scheme which overcomes this mismatch without any need to resort to buffering of the video output data.

DisArray currently supports a 512$\times$512 monochrome bitmap display, but this is relatively arbitrary and can be easily changed. There is currently over 16 times this video bandwidth available if necessary, to support higher-resolution displays or colour, or both.

7 Address Staggering Scheme

To achieve direct video refresh from the array without the use of output buffering needs some re-organisation of storage. Clearly, considering scan line 0, *all* of the bits contributing to this line are in row 0 of the array.

To get 16 consecutive 16-bit segments of scan line 0 out of the array simultaneously (which is what the t.v. monitor requires), these segments must necessarily then be in different rows of the array. This can be accommodated at no extra software or hardware cost by arranging that horizontally consecutive planes are stored such that they are circularly shifted respectively southwards by one row. Now, it is only necessary to arrange that each row of processors gets the appropriate refresh address on any video refresh cycle. These addresses can be simply formed

from a single video refresh plane address by adding [2] the row number in the array to the refresh address, but not allowing the carry to propagate past the bottom four bits.

The only other remaining problem is that the 'origin' of the 256-bit scan-line segment that gets loaded into the video shift register is also shifted. However, this is easily overcome, since the shift register is spiral and all we need to do is to arrange for 16 *tapping-points* on the shift register at the end of each row and to select one of these tapping points with a multiplexor to get the correct video bit stream.

8 Quadrant Addressing Scheme

As discussed above, the inner loop of RasterOp entails adjusting a plane to bit boundaries in both directions and then performing a read/modify/write cycle on each of the four quadrants involved. On each of these cycles the array is not performing useful work in the other three quadrants. The four memory cycles can be compressed into a single cycle if the processors in the four quadrants could each receive a different address.

There are a number of possible ways to achieve this, three of which are outlined here :

- Broadcast all four addresses in turn along the common memory address lines and use strobes generated on a per-quadrant or per-processor basis to cycle the memory chips. This is a simple extension of the method of time-multiplexing of the address lines already in use because of the nature of dynamic ram addressing.

- Build memory chips [3] with some intelligence in the address paths. What is needed is the ability to store a small number of addresses on the memory chip and to optionally combine one of these with the incoming address in a simple adder with outside control of the carry-in signal. In this way, with a current memory address m, and a stored address s, the chip could access the following locations under the control of two extra control signals :

  m, m+1, m+s, m+s+1

  This is precisely the pattern of addresses required by RasterOp.

- Partition the memory address into two parts; an x-plane-address and a y-plane-address. This institutionalises what the software often does anyway, which is to regard the array memory storage as a single, large two-dimensional bitmap.

  This means that it is possible to generate the two x-parts of the addresses and broadcast the appropriate one of them down each of 16 sets of column-wise address busses. If the same is done with the y-parts broadcast row-wise, then the address required by each processor is then formed by concatenation of the x-part and y-part that passes through that processor. This scheme has the considerable benefit of requiring no extra hardware in each processor.

In fact, none of these quadrant addressing schemes have yet been implemented in the hardware. We may however implement the third method in the not-too-distant future although we would really prefer option 2 if we could get access to the technology required to produce the

[2] In fact, with the memory mapping arrangement in DisArray the appropriate address to be sent to each row of processors is *RefreshAddress - RowNumber - 1* This is easily provided from a 4-bit alu slice on the low-order 4 bits of the memory address bus on each row of processors.

[3] In fact, if we had the capability of building a reasonable size of memory chip, then we would put somewhat more intelligence than this into the address lines and also put a processing element onto the same chip. This results in a smart memory chip with many other useful applications.

necessary 'smart' memory chips. We have already successfully designed and fabricated our first chip [4] but the fabrication facilities we have available are not nearly good enough for us to create a useful size of intelligent RAM.

9 Disarray Performance

The array can shift the Q-Register in any of the four orthogonal directions at 30MHz, giving a total bit shifting capability of over 7 Gbits/second. This operation is the basic mechanism for aligning a source plane with the (four) destination planes and it is the two-dimensional analogue of the use of a barrel shifter in the data paths of a one-dimensional RasterOp processor.

The array has a rather leisurely 600ns read/modify/write memory cycle time. This is slower than it need be because of the low speed [5] of the dynamic rams chips that we used. This gives the array a basic computational rate (Memory := Memory Op Register) of just over 400 Mbits/second (plane-aligned). This could easily be tripled using current memory chips to 1.2 Gbits/second.

Higher level performance information on the prototype is not yet available since the control unit is only now being commissioned. However, we expect a to achieve speeds something like the following :

	Current Hardware Addressing	With Quadrant Store	With 200nS	With Both
Character-sized RasterOps/sec	200k	300k	300k	500k
Large scale RasterOp > 10 kbits. Rate in Mbits/sec	80	160	200	260

These figures ignore the video refresh overhead, which is variable [6] depending on the screen size. These figures will improve when we move away from the PDP11 to a faster and more suitable host processor. We are currently considering a second user-microprogrammable 29116-based processor for this task.

10 DisArray2 - The Next Generation

The intention is that the DisArray concept will lead to the design and construction of a *fifth generation workstation* in which a Powerful Personal Computer is completely integrated with a DisArray processor. DisArray2 is currently an outline design for such a system, based on newly-available, very powerful LSI components, which could improve on the performance of the current DisArray by a factor of about 5 and also provide (at peak) about *300 Mips of vector processing* on 16-bit quantities. We will introduce the reader to this design in a number of stages.

[4] A 32-bit in, 16-bit out barrel shifter with two-level pipeline input registers for aiding RasterOp on conventional 16-bit micros.

[5] The ram chips were purchased over four years ago, before a rather long break in the project when no manpower was available for construction.

[6] For example, with a 1024×768 picture the overhead is about 200k memory cycles/sec at either 600 or 200 nS/cycle

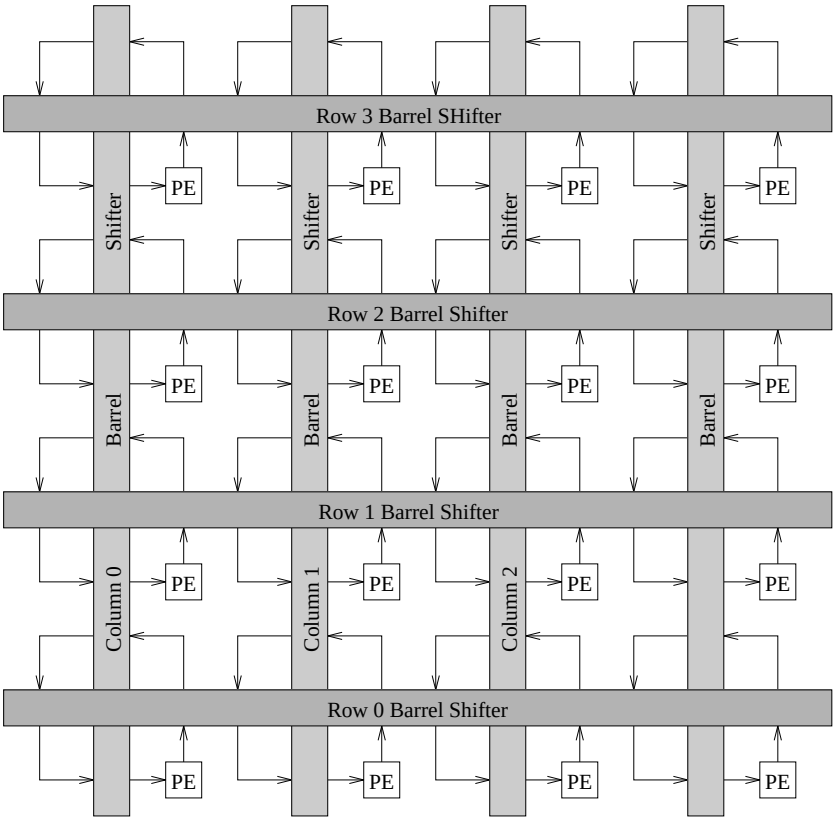

Figure 15: A 4×4 'Surface Shifter' implemented by using 8×4-bit barrel shifters connected to a 4×4 array of Processing Elements.

10.1 Surface Shifting

One of the areas in which DisArray can be speeded up is by speeding up the serial shifting that is required to align planes to bit-boundaries. In a conventional machine the solution would be to replace the serial shifter with a barrel shifter.

Happily, a similar solution is possible in the two-dimensional case. We simply replace all of the horizontal neighbour connections in a single row of the array with a barrel shifter. This then gives us a fully-connected arrangement where any processor has a direct connection to *any* other processor in the same row. We repeat this for every row in the array using a total of 16, 16-bit barrel shifters. We also repeat the pattern along the 16 columns of the array giving each processor immediate access to any other processor in the same column. In fact, there may be no need to have access to the intermediate result between a horizontal shift and a vertical shift. In this case, at each processor location the processor can send data to the appropriate input port of a horizontal barrel shifter. The corresponding output port of that horizontal barrel shifter is directly connected to the appropriate input port of a vertical barrel shifter, whose corresponding output is then fed back to the same processor.

This composite shifter, comprising 32, 16-bit barrel shifters, is a two-dimensional analogue of the one dimensional barrel shifter, Fig. 15. Since I am not aware that this structure has been reported before, I have called it a *surface shifter*.

This arrangement can obviously speed up the shifting [7] but at some cost. In fact one of the nice features of the simple nearest neighbour-connected array is that it can be extended indefinitely. The surface shifter can probably only reasonably be implemented with at least each barrel shifter being totally implemented on one chip. This means that the maximum size of the array would be limited by the number of pins on a package. Currently, we do not have access to the production facilities that would be needed if we made a 32-bit barrel-shifter/processor chip.

Another advantage emerges if it is possible to set a separate shift constant for each row and column shifter simultaneously. Consider the case in which the row number of each row is used as the shift amount for a circular shift in that row. Considering only the horizontal component of the shift, what we have achieved is a *skewed* shift of the data, with wrap-around. Row 0 is unchanged, Row 1 is circulated by 1 bit etc. I have called this a *skew-circular surface shift.* In this case it is a horizontally-based shift, but it obviously has a vertically-based counterpart. By applying the following sequence of shifts to a plane :

1. Horizontal skew-circular surface shift,
2. Vertical skew-circular surface shift,
3. Horizontal skew-circular surface shift,

the effect is to completely rotate the contents of the plane by 90 degrees in only three machine cycles [References 5, 7].

10.2 Processor Element Implementation

Having decided to use the surface shifter concept one needs to implement both the barrel shifters and the Processing Elements. Fortuitously, there are two newly-available fast LSI chips which combine a 16-bit ALU with a number of registers and a barrel shifter; these being the AMD 29116 and the TMS 3020. These chips are reasonably good for this application since each one bit section of the ALU/Register/Data Path can function as a one-bit Processing Element as well as providing the parallel shifting capability. Thus we get 16 of our 1-bit processors processors and a row (or column) barrel shifter on one chip. Without access to the appropriate VLSI fabrication facilities needed for a 32 $\times$32 DisArray2, we could use one of these two components for a 16 $\times$16 version. However, this is not as attractive to us since we have already built one 16 $\times$16 machine.

The AMD 29116 is a 100ns component, and using this has the added benefit that the 16-bit ALU can also carry out 16-bit arithmetic in 100nS. This means that with 32 of these processors in the 16-vertical, 16-horizontal arrangement, the array can achieve all that the DisArray currently can, together with faster shifting and having a capability of over 300 MIPs arithmetic processing performance on arrays of 16-bit quantities.

There are a number of important graphics algorithms which would have a greatly improved performance on this type of hardware. Some of these are:

- Line drawing with a DDA algorithm. The 16 horizontal (or vertical) 16-bit processors can work together to plot up to 16 points simultaneously from an arbitrary line across a 16 $\times$16 plane.
- 16 of the processors can co-operate in polygon filling applications.
- Co-ordinate transformation of multiple data points can be achieved in parallel with well-thought out arrangements of data.

[7] However, using currently available components this speed-up is not quite as good as one might at first suppose since serial shifting can be made quite fast (DisArray currently runs at 30 MHz).

- If a *bit-reversal* path could be included in each of the processors (which entails designing the processor chip from start), it is possible to mirror bitmaps and transpose matrices. Amongst other cases, this can be used to rapidly calculate transitive closure relationships which are very useful in advanced window-based interactive systems [Reference 6]

10.3 Other DisArray2 Features

In addition to the above, DisArray2 has a number of other features which can only be mentioned briefly here :

- The Address staggering scheme of DisArray for video refresh.
- A second similarly-arranged video shift register for the real-time *input* of video data.
- The quadrant addressing scheme to speed up operations on neighbouring planes.
- There will be *two* fast RAMs at each intersection point in the array, one being associated with the horizontal processor and one with the vertical processor. With this arrangement and a suitable mapping of array storage onto the address space of the host processor, it will be possible to fully utilise the processing capability of both sets of processors for well-organised vector arithmetic; for co-ordinate transformations, say. This would ensure that the 300 MIPs rating would not be degraded by both sets of processors competing for the same memory (in the cases where memory allocation could be so organised).

11 Conclusions

One solution to the problem of increasing the speed of the *RasterOp* graphics primitive has been presented, which uses a regular two-dimensional array of simple processor/memory pairs. A working version of this architecture has been built and performs its task well. The control unit which will allow the array processor to run at its full speed is now being commissioned.

Further work may increase the speed still further in a later version of the hardware. Hopefully, this later version will be in the form of a powerful personal computer fully integrated with a more general-purpose array processor than the current one. This combination would provide a very useful amount of computing power for the execution of graphics and other applications.

This hardware was only designed to solve just *one* of the problems of high-performance graphics; namely that of executing RasterOp at high speed. However, it has come to be appreciated that this hardware architecture, suitable enhanced, will also be capable of performing a very significant amount of the number-crunching required for a more general-purpose graphics environment. We have strong hopes that the architecture will be of some benefit in the parallel execution of functional languages. Also, because of its great regularity, the architecture is eminently suitable for VLSI implementation. The possibility of an 'intelligent' RAM chip for this and other applications is very attractive.

The author gratefully acknowledges the support of the United Kingdom Science and Engineering Research Council and International Computers Ltd. in this work.

12 References

1. **Principles of Interactive Computer Graphics**, Newman and Sproull, 2nd Edition, McGraw-Hill, 1979.

2. **Alto : A Personal Computer**, C.P. Thacker et al., Xerox Palo Alto Research Center, July 1979.

3. **Perq Software Reference Manual**, Three Rivers Computer Corporation, 720 Gross St., Pittsburgh, PA.

4. **DAP - A Distributed Array Processor**, S.F. Reddaway, 1st. annual Symposium on Computer Architecture, Gainesville, Florida 1973.

5. **A Language for BitMap Manipulation**, L. Guibas, J. Stolfi, ACM Transactions on Graphics, Vol. 1, No. 3.

6. **Playing Cards**, S. Cook, QMC Internal Report (to be published).

7. **Smalltalk-80, The Language and its Implementation**, Goldberg and Robson, Addison-Wesley 1983.

8. **The Q.M.C. Text Terminal**, Page and Walsby, Electronic Displays '78, Conference Proceedings, London, Sept. 1978.