Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

P.J.W. ten Hagen, I. Herman, J.R.G. de Vries

A dataflow graphics workstation

Computer Science/Department of Interactive Systems Report CS-R8910 April

---

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Research (N.W.O.).

A Dataflow Graphics Workstation

P.J.W. ten Hagen, I. Herman
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

J.R.G. de Vries
Dataflow Technology Nederland bv.
Maanweg 156, Gebouw BN 086, 2516 AB Den Haag, The Netherlands

A new, revolutionary architecture for a superworkstation for graphical purposes is presented. The architecture is based on the use of advanced graphics components and, mainly, on the heavy use of dataflow processing technology, a still unexplored field of parallel computing as far as graphics is concerned. The resulting initial configuration is able to produce 200.000 to 250.000 Gouraud shaded and Z-buffered 3D triangles in a second with a colour palette of 24 bits per pixels.

1983 CR Categories: B.1.5, C.0, I.3.1, I.3.2.
Keywords & Phrases: computer graphics hardware, parallellism in computer graphics, dataflow computing.
Note: This research has been made within the frame of the common development contract between CWI and Dataflow Technology bv. The present text has also been offered for publication for the journal Computers & Graphics.

1. Introduction

The architecture for a graphics workstation presented in this paper is a revolutionary one because it is the first in a new and promising line of architectures. The revolution stems from the fact that with the dataflow processors used, arbitrary combinations of highly dedicated processing can be dynamically created. For the case we will describe here all processing is dedicated to graphics but the application is by no means restricted to this field. In fact, such processors have been, until now, primarily used in image processing applications. Once the case is made for the graphics area, it is obvious that the next step will be to provide a system which combines graphics and image processing. There are many applications in the field waiting for a system which integrates both possibilities. The purpose of this paper is however, to illustrate how well dataflow architectures can support three dimensional interactive graphics.

The system described here is not an experimental system. It is much more. It uses existing hardware components only, albeit they are all the most advanced hardware components currently available. In addition, the entire architecture has been designed and is being implemented. Testing of the integrated system will start in June 1989. Production for the market might start before the end of 1989. The performance figures mentioned in the paper are based on testing the working individual components and/or simulation outputs.

The discussion about the results and possible improvements is based on a particular initial configuration chosen around 32 dataflow processors in a hypercube arrangement. The flexibility of the dataflow hardware and the overall design guarantee that simple extensions to larger configurations containing 64 or more processors are possible. However, in this paper all figures and possible improvements are only concerned with this initial configuration. In this way the reader can make a proper judgement about the possibilities opened up by this line of systems without getting confused by the great variety of extensions and further performance boosts possible.

Report CS-R8910
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

2. The Hardware

2.1. General Overview

The hardware of DFC (Data Flow Computer) is centered around three basic building blocks: the *Host*, the *Graphics Engine* and the *Dataflow Engine* (see Figure 1). The Host may be any UNIX BSD machine provided that a VME access is available (we use currently a SUN 3 Workstation). Both the Graphics Engine and the Dataflow Engine are used as co-processors of this Host †.

The building blocks are connected via two busses: a standard VME bus and a so called *Distributed Memory Bus*. The former is primarily used to connect the Host with the two other blocks. The Host may control the processing parts of the two other blocks via register I/O and DMA; all these possibilities are quite standard on such configurations.

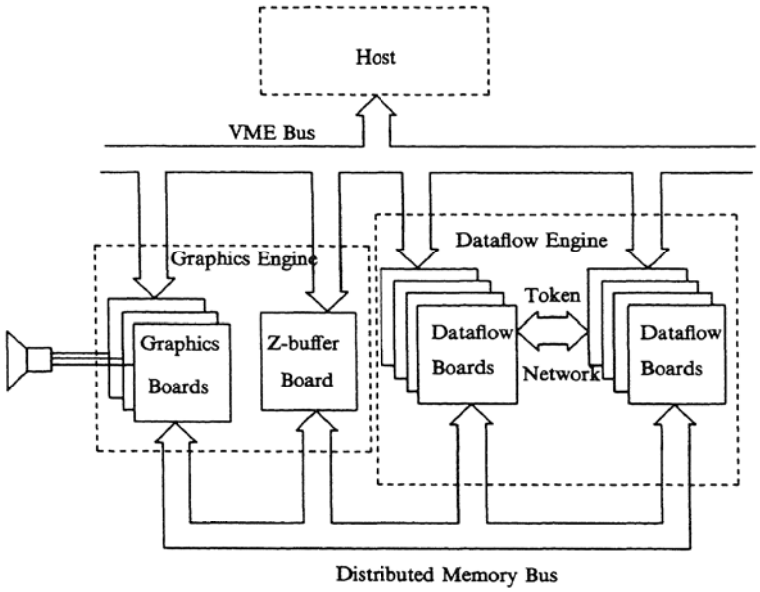

Figure 1: Overall view of the DFC

The Distributed Memory Bus is used to provide data transfer between the Graphics Engine and the Dataflow Engine as well as among the different internal components of these. This bus is much faster than the VME bus: the current transfer speed is 36Mbytes/sec (16 million data transfers per seconds). All memories on all boards are connected via this bus, including the video memories and the 3D Z-buffer used by the graphics processors.

A third means of data transfer is provided by the dataflow *Token Network* with a capacity of 320Mbytes/sec, which connects the components of the Dataflow Engine. The role of this network will become clear in the following paragraphs.

† It is theoretically possible to put for example two independent Dataflow Engines on the same VME bus, getting therefore two independent co-processors of this type. However, we will not deal with this configuration in the present paper.

2.2. The Graphics Engine

The Graphics Engine contains three identical components, namely the so called *Advanced Graphics Boards*, and, additionally to them a *Z-Buffer Board*. The conception of the Advanced Graphics Boards is such, that they are also usable as stand-alone graphics boards in a VME environment; however, details of this version are not covered in the present paper.

Each *Advanced Graphics Board* contains a 4Mbytes VRAM for pixel memory with RGB input and output. Using the three boards together we get therefore a frame buffer of 24 bit/pixel assuming a resolution of max. 2048×2048. Each board controls 8 bits out of the 24; the first board is responsible for the generation of the red pixels, the second one for the green pixels and, finally, the third one for the blue ones. Additional cursor hardware is available to mix a cursor on video output without overwriting the frame buffer. The board has also extensive facilities to generate, synchronise and input video signals in virtually all video formats; this feature might be very important for future image processing applications.

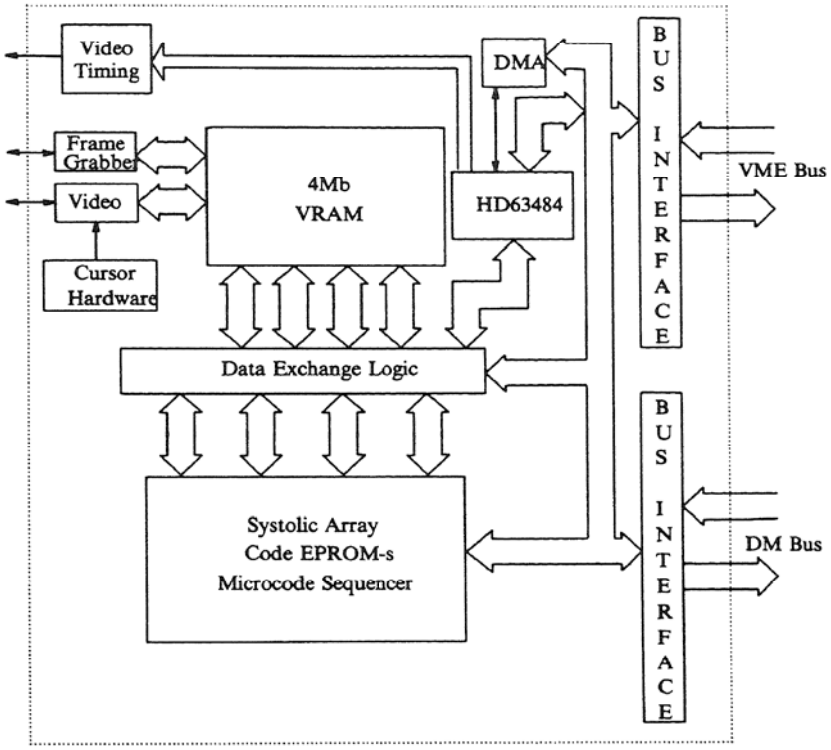

Figure 2: The Advanced Graphics Board

Each board contains a Hitachi HD63484 graphics processor, which may be used for high-speed two-dimensional graphics. However, as we will see in the following, the main use of the whole DFC is to provide three dimensional graphics; in other words, the HD63484 processor is used first of all for video timing generation and to help video input.

The main "processing unit" on the Graphics Board is a Systolic Array which contains four 16 MIPS bit slice processors (16 bits wide). Each one is accompanied by a small multiplier and a function table. This Systolic Array is microprogrammable, and with an adequate set of microcoded

instructions it turns the Advanced Graphics Board into a powerful graphics co-processor. Because of the uniform nature of the three Graphics Boards, the very same microcode runs in parallel on the boards controlling, as we have already mentioned, the red, green and blue pixels respectively. The available instruction set contains the usual BitBlt operations (Block Copy, Block Fill, etc.), vector drawing and some image processing instructions. The speed of pixel generation is considerable: in case of Block Copy, for example, 20 million pixels/sec. may be copied; the peak vector generation speed (for horizontal lines) is of 30 million pixels/sec.

For the purpose of a 3D Workstation, however, the most important instruction is the possibility of generating 3D Gouraud shaded triangles. The Systolic Array gets the colour values (red, green and blue values respectively) for each vertex; during scan-conversion these values are linearly interpolated to set the right colour value for each pixel. This instruction, however, has to cooperate with the fourth component of the Graphics Engine: the Z-buffer Board.

The *Z-Buffer Board* contains a similar Systolic Array to the Graphics Boards but with a different microcode. Additionally, the board contains a 16Mbytes Z-buffer memory. This memory is able to store the Z value of the incoming primitives, with a depth of 16 bits.

The microcode of the Z-Buffer Board determines which portion of a 3D triangle is effectively visible by performing a scan conversion in the third dimension and comparing the value of the generated depths with the content of the Z-buffer. As a result, the Z-Buffer Board dispatches commands to the Graphics Boards so that these latter ones will generate the visible x and y pixels by interpolating the colour values. In other words, via four cooperating boards and microcodes the Graphics Engine has the possibility to generate 3D Gouraud shaded triangles with Hidden Surface effects automatically calculated via the Z-buffer. This instruction will serve as a basis for the 3D graphics capabilities of the DFC; in a way, the whole software realised in the machine aims at the fast creation of appropriate triangles. At present, the generation speed is on average 180.000 to 200.000 triangles/sec (the real speed depends on the size of the triangles, of course).

Although the possibility of triangle generation is by far the most important feature of the Graphics Engine, it has also some additional instructions. An interesting example is the possibility of *shielding*.

A *shield* is a polygon which is set into the Z-buffer to any "location", that is, conceptually, onto a plane which is parallel to the $x-y$ plane. The effect is that all pixels whose coordinates belong the $x-y$ area covered by the shield and which are "farther" as the depth position of the shield will be clipped automatically. Such shields may be "or"-ed in the Z buffer, combining therefore simple shields to produce more complicated ones.

This facility offers a natural tool for traditional clipping. Furthermore, if some of the shields in use are the set-theoretical negation of a rectangle in the frame buffer with respect to the full 2048×2048 frame buffer, by the combination of these and "classical" clipping rectangles the clipping operations required by a window manager environment are authomatically covered. Indeed, if such a "negated" shield is set first for a given window and, subsequently, all rectangles belonging to overlapping windows are set, the resulting shield in the Z buffer will correspond exactly to the clipping area required by the windowing environment. Clearly, this facility has a great practical importance.

2.3. The Dataflow Engine

The Dataflow Engine is without doubt the most original and most interesting part of the whole Workstation, which gives it a very individual flavour. The Engine is a very fast computing co-processor of the Host; its primary task is to provide transformation of three dimensional points as well as offering a higher-level software interface to the Host than the plain 3D triangles. To be able to use the speed of the Graphics Engine (which is quite high), this co-processor should also be very fast. To achieve the required speed (that is $\simeq$200.000 triangles/sec) the technique of *dataflow computing* is used. As this technique is not yet widely known, we have to make a small detour to introduce the basics of it; for a more detailed introduction the reader is referred to Veen[1] or Herath et al.[2].

2.3.1. Dataflow Computing. Dataflow computing is an approach to a hardware and software organisation which aims at offering a highly parallelised structure as an alternative to the traditional von Neumann computing architecture. Its basic notion is *dataflow nets* (also called *dataflow graphs*), which may be defined as follows.

A *dataflow net* is a directed graph; the vertices of the graph are called the *nodes* in the dataflow terminology, while the edges are the *links* of the net. Each node represents a tiny processing element; the actual processing performed by the node is defined when defining the graph itself. A node may perform very elementary operations only; its possible instruction set may be compared to the assembly instructions of a traditional computer.

Data are encapsulated in *tokens*. These tokens are transferred from one node to another by the links, following the direction of the link. The nodes themselves operate on the data carried by the input token(s) of the node and the operation itself may either delete the token (if no output link is defined) or may produce one or more token(s) again, each of them being output on one of the output links defined in the net.

A node will perform its defined operation when all input links contain a token. This rule may remind the reader of the theory of Petri nets, although a dataflow net is *not* exactly a Petri net (in the latter case, all processing nodes are of analogous type, which is by far not the case in a dataflow net). No assumption may be used as far as the relative timing of the nodes: if all tokens are present, a node may process *at any time, regardless of the remaining nodes* (the exact timing depends on the hardware realisation of the dataflow net). This means that a dataflow net represents a very high level of parallelism; that is why it is also referred to as *data-driven fine-grain* parallelism.

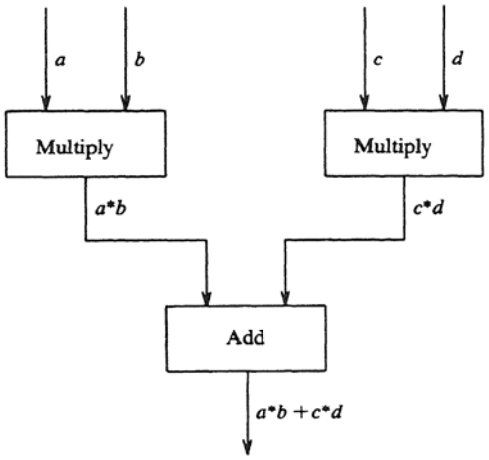

Figure 3: A simple dataflow net

Figure 3 shows a very simple example of a dataflow net; this net performs the calculations for $a*b+c*d$. The data for *a*, *b*, *c* and *d* are sent to the net on the corresponding links (see Figure 3); they may arrive in a random order. Once *a and b* are present (respectively *c and d*), the node which performs the multiplication may process and will produce $a*b$ (resp. $c*d$). The multiplier nodes will operate in parallel; no assumption may be made as far as their relative timing is concerned. On the other hand, once *both* $a*b$ and $c*d$ are produced on their respective links, the node responsible for the addition may proceed and will produce the final result. Figure 9 at the end of the paper gives an example for a much more complicated dataflow net.

A *dataflow computer* is a computer which is *programmable* using the dataflow model. In other words, such a computer should be fed somehow with the description of the net, including the topology of the net as well as the instruction(s) assigned to each node. Such a network is then *executed* by feeding all input links with the necessary tokens so that the nodes of the network could start proceeding following the rule cited above.

2.3.2. The Dataflow Boards. The Dataflow Engine consists of eight identical and interconnected *Dataflow Boards*. Each Dataflow Board is based, in turn, on NEC's μPD72181 chip, the so called Image Pipelined Processor[3], which we will call the DFP chip hereafter.

A DFP is a tiny dataflow computer. The appropriate description of a dataflow net may be loaded onto the chip and by getting the necessary tokens the chip is able to execute the net. It is tiny, because the dataflow net which may be loaded is relatively simple: it may contain at most 64 nodes and 128 links. In contrast to a von Neumann microprocessor, the coded instruction cannot be stored on an external memory; all instructions should be stored within the chip (which makes it, of course, much faster!). Some part of the instruction set, which may be used when defining the nodes, corresponds to an early day microprocessor: 16 bits integer addition, multiplication, comparison (no division!), bit setting/resetting instructions as well as shift operations. Additionally to these ones there are a number of operations to "manage" the net (token deletion, token duplication, dispatching over links etc.) as well as instructions specially dedicated for image processing purposes (e.g. shift and bit count instructions).

However, the DFP *is* a revolutionary machine: it is the first commercially available single-chip dataflow computer, which makes it extremely interesting to explore its capabilities for example for graphics. It may operate at 20MHz and, for example, a 16 bits by 16 bits multiplication is performed at 100 ns. The extensive use of internal parallelism makes it very fast indeed.

The DFP needs additional instructions to communicate with its environment. There are basically two ways of communications: either a token is sent to another DFP or some external medium is to be accessed for reading and/or writing. To manage this latter task, NEC offers also an additional chip, the μPD9305. Via the μPD9305 chip, a DFP has the possibility to access a 24 bits address range for external I/O.

Each DFC Dataflow Board contains 4 DFP-s, one μPD9305, a so called Image Memory, which is a 1 Mwords DRAM (one word equals 18 bits) and, finally, a DMA controller for fast memory I/O. As we have already mentioned, the board's registers may be accessed from the VME as well as from the Distributed Memory Bus. The DFP-s may access the Image Memory by making use of the facilities offered by the μPD9305 (see Figure 4).

The whole Dataflow Engine consists of 8 Dataflow Boards; in other words, it contains 32 DFP-s and 8 Mwords of Image Memory, which represents altogether quite an impressive processing power. Each DFP may uniformly access *the whole* Image Memory (not only the part of it which resides on the same board). As we have already mentioned, this memory is also accessible directly from the Host. Token communication among the DFP chips is routed by the Token Network; that is, token communication of the DFP-s has a private data path for itself.

How is this Engine programmed? Each DFP has to be loaded with its own dataflow net. The whole program of the Dataflow Engine is *not* one huge net; instead, it consists of 32 independent dataflow nets which may communicate with one another via special nodes and links. The Host may also send tokens to any of the DFP-s and the DFP-s are also able to send data to the Host in the form of tokens. By using these facilities the synchronisation of the dataflow net as well as the synchronization of the whole Dataflow Engine with the Host and/or the Graphics Engine may be achieved. How this program effectively works will be the topics of the forthcoming sections.

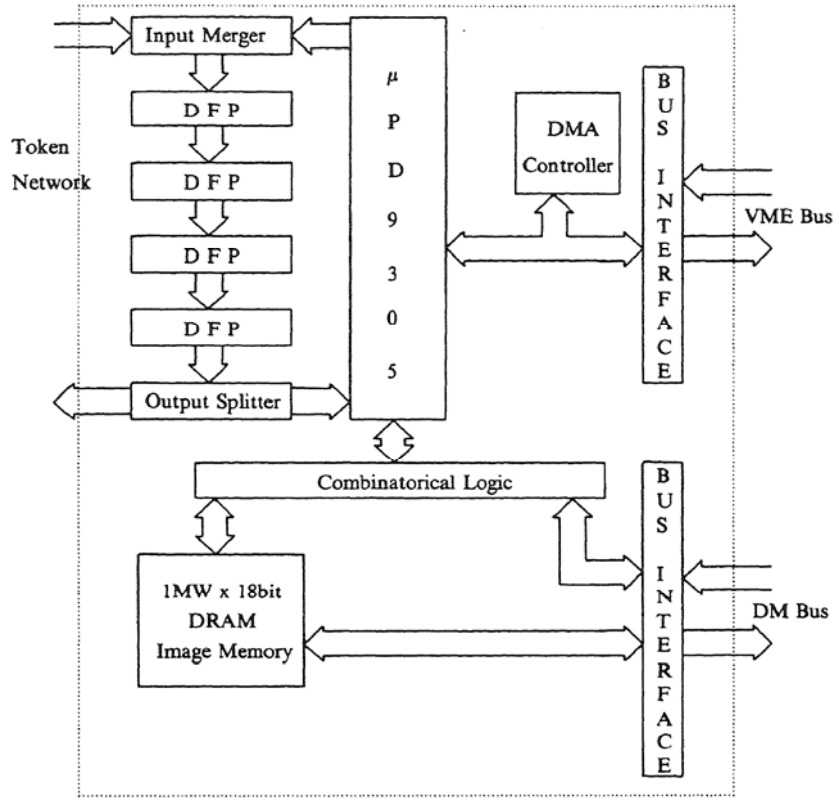

Figure 4: The Dataflow Board

3. The Software

3.1. Some Generalities

There are different forms of parallelisms which may be used in a graphics system. The choice where and in which form a parallel architecture is introduced, may lead to very different hardware/firmware/software configurations.

The image generation on a raster device may be cut into two distinct steps. On the one hand, high level geometric structures should be manipulated (clippings, transformations etc.) and, on the other hand, the objects should be rendered on scan converting level. The different attempts of parallelisation in graphics systems follow roughly the same division; there are approaches which try to introduce parallelism on the structure level while others try to exploit the possibilities of scan converting level parallelism. We do not want to give an exhaustive survey of all different approaches here. As examples for structure level parallelism we might cite all different pipeline architectures, which are used in a number of commercially available graphics workstations (e.g. Silicon Graphics), the PHIGS machine of Abi-Ezzi et al.[4], or the MAGIC II machine of Finch et al[5]. The different "smart" pixel memories like the Pixel Machine[6], the DisArray machine[7,8], Scan Line Access Memories[9], or others are good examples for scan converting level parallelism. We should also remark that this classification is, as all such classifications, not really precise; there are "hybrid" approaches as well

(see e.g. ten Hagen et al.[10], where the distinction created by the presence of a frame buffer disappears).

In case of the DFC, both forms of parallelism are present, although to a different extent. As we have seen when presenting the hardware, the Graphics Engine does include a certain level of parallelism; indeed, the pixel generation for red, green and blue pixels as well as Z-buffer handling are done in parallel by different pieces of hardware. Once "inside" the boards the fact that there are four processors within the systolic arrays is a source of parallelism again. Finally, the microporgramming technique itself is also a non-negligible source of parallelism. In other words, the Graphics Engine does contain elements of scan converting level parallelism; however, to exploit the possibilities of the Dataflow Engine, we had to concentrate first of all on the possibilities of structure level parallelism.

As far as structure level parallelism is concerned, there are again different possible approaches. Some of them are as follows.

(1)A scene to be visualised consists of a number of more or less independent objects (as far as geometry is concerned). In case of a full regeneration of a picture, the individual objects may be transformed and generated independently, provided that appropriate Z-buffer hardware takes care of hidden surface removal (such a hardware is available in the DFC). In other words, a set of independent output pipelines, including transformations and some basic clips may run in parallel.

(2)In case of interactive use, the whole scene is not necessarily to be regenerated starting on the top level of representation. If e.g. only one object has been moved, it is superfluous to retransform all objects, except the one whose position or orientation have been changed. On the other hand, to achieve correct output, more (sometimes all) objects must be re-drawn. This is another source of parallelism: while transforming one object, the others may be subject to a redrawing process, provided that the intermediate (i.e. transformed but not yet scan-converted) storage of the object is also available.

(3)The output pipeline itself may also be highly parallel. The same kind of operations (matrix-vector multiplication, shading calculations, projective division etc.) have to be performed on a set of points belonging to one objects.

(4)Finally, the mathematical formulas in use (e.g. vector-vector multiplication) include possible sources of parallelism.

In a "conventional" architecture it is fairly complicated to take into account all these possibilities of parallelism (and there may be even more!). Usually, one has to concentrate on one or two such aspects, to have a managable task at hand. This is why the data flow approach seems to be fruitful: by its very nature it is fairly straightforward to model all kinds of parallelism, by providing the appropriate software; in fact, all classical approaches to parallelism (SIMD, pipelines, MIMD etc.) may be "simulated" easily in a data flow environment. In the software we have managed to provide for the DFC, all aspects of parallelisms cited above could be realised in a natural way.

Figure 5 gives an overall view of the software running on the DFC Workstation. All of these software blocks will not be explained in detail; this would go far beyond the scope of the present paper. In the following, we will concentrate first of all on the most original part of this software, namely on the programs running on the Dataflow Engine.

3.2. The Dataflow Software

3.2.1. Overall Structure. The dataflow programs realise a number of logical functions operating on well defined data. These functions are resident in the DFP processors and are activated by sending some specified tokens to them and are invoked by the Host application program(s). One may view these as being co-routines of some concurrent programming languages, where sending one or more starting tokens corresponds to the activation of the co-routine. The hardware and the software organisation makes it certain that there is no hidden sequentiality; e.g. logically different functions do not share DFP-s. Each function sends back a token to the Host once its task has been finished and, therefore, the corresponding DFP is ready to process again.

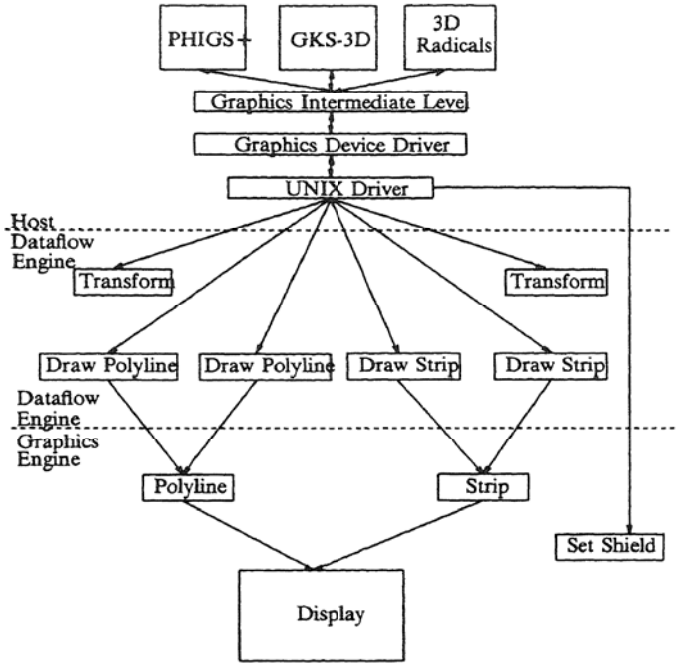

Figure 5 : Overall view of the DFC Software

There is, however, one significant difference: there might be more *instances* of the same function. This is, as we will see later, the case of e.g. the transformation function: the Host has *two* transformation functions at its disposal, which are identical as far as their functional specification is concerned but may run absolutely in parallel.

The functions operate on two types of entities: *point lists* and *colour values*.

Point lists are arrays of four dimensional (homogeneous) points stored in Image Memory in form of integers The points may be logically located in two coordinate systems. The first one is (to use the usual terminology) a *World Coordinate System* whose limits are not specified. The second coordinate system is the *Device Coordinate System* where the x,y coordinates are expected to be in the range of the display resolution and z within the limits of the Z-buffer. In the latter case the w coordinates have no real geometrical meaning but they are used for internal purposes.

Point lists are transformed by a dataflow transformation function which performs a matrix-vector multiplication and the projective division: in other words, it turns point lists stored in World Coordinates into point lists stored in Device Coordinates. The starting points (that is the ones described in World Coordinates) are put into the Image Memory by the Host. As we have already mentioned, there are two instances of this function; in other words, the Host has the possibility to start a transformation function two times *in parallel*; this is a significant source of speeding up the overall speed of the Workstation. The choice of having two such transformation function instances (and not more) is the outcome of a balance between DFP chip numbers and required overall speed; clearly, if the hardware were different e.g. by having newer versions of the dataflow processors, this number could change as well.

How the point lists are interpreted depends on various drawing functions. In the actual version

there are basically two of them: the *"draw polyline"* and the *"draw strip"*. Both of them read the points from the point list and read the associated colour (R,G,B) from the colour list. Here again, there might be several instances of the same function; at present we plan to have two instances of both the "draw polyline" and "draw strip" functions.

While the role of the "polyline" function is clear, the "draw strip" function needs some explanations. This function uses a list of points and a list of colour values to generate a series of adjacent triangles (see Figure 6). Such strips may be generated by the Host by decomposing polygons and they are also the natural outcome of a number of surface approximation methods. Using the two available entities the "draw strip" dataflow function can create a set of 3D triangles to be sent to the Graphics Engine which, on its turn, will render the Gouraud shaded triangles on the screen. (We have to remark that this primitive appears directly in the functional specification of PHIGS+[11]).

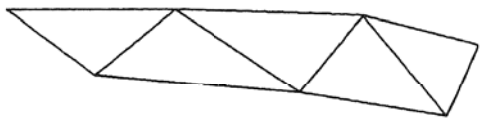

Figure 6: A strip

Figure 7 gives an overall picture of these basic dataflow functions. Not all functions are shown in the figure only the ones cited above. There are additional functions as well like "reset DFC", "set transformation matrix" and there is also a function aimed at speeding up pick input.

Clearly, the main advantage of this kind of software architecture resides in the fact that the different functions may be activated separately and they may run fully in parallel; in other words the structure level parallelism of the kind (1) and (2) listed in the previous paragraph are fully covered. Using this possibility of parallelism is of course the task of the Host; this is the unit which should have a full control over all objects on the screen, which should know which objects are to be retransformed, which ones have to be redisplayed only etc. In other words a rather complicated piece of software had to be realised to drive the whole DFC.

It is not possible to present here all details of all functions implemented in DFC. As an example, we will just give some details of the implementation of the transformation function; this will be enough, we hope, to give a flavour of the techniques used in programming the Dataflow Engine of DFC.

3.2.2. Details of a Dataflow Function: The Transformation Function. The primary task of the function is to transform four dimensional homogeneous coordinates. Mathematically, this transformation consists of two steps: the multiplication of a four dimensional vector with a 4×4 matrix and secondly, the projective division, that is the division of the x,y and z coordinates with the fourth w coordinate.

The particular difficulty in programming this function resides in the fact that the DFP processor has no division instruction. In other words, a separate dataflow program had to be written to implement this operation. Additionally, all arithmetic instructions are based on 16 bits arithmetic; unfortunately, however, 32 bits should be used for internal calculation, otherwise the resulting errors in the calculations would become excessively high.

However, all operations to be performed have a common feature, namely the fact that they can be performed on individual points of the point list absolutely independently from one another. In other words, there is a possibility of parallelism here which it is worthwile to exploit.

Figure 8 gives the overall view of how the transformation function is implemented. The function is spread over 7 DFP processors, each processor performing a specific task. All processors work in parallel and they form, as we will see, a kind of a transformation pipeline.

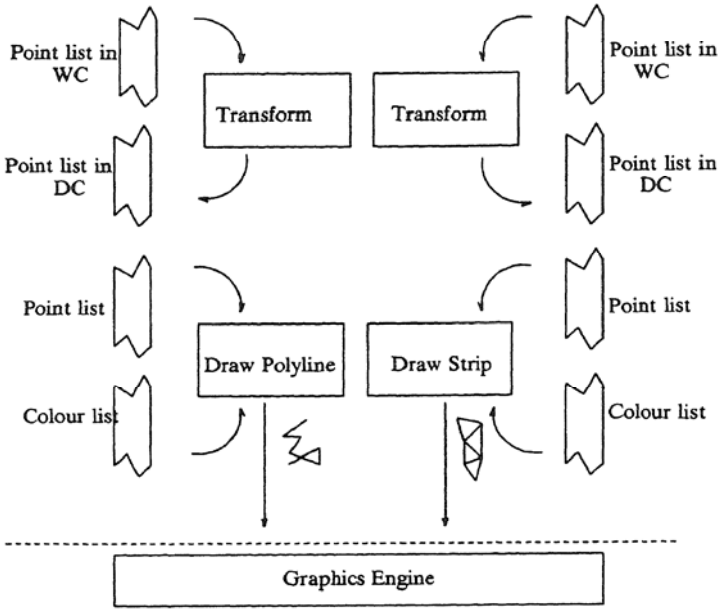

Figure 7: Overall structure of the DFC dataflow software

As it can be seen in the figure, some of the DFP processors perform functionally the same task; they realise several instances of the same function. The repartition of the tasks is as follows.

The "top" function, called the supervisor, actually reads the points from Image Memory. As we have already remarked, reading from memory is a time consuming task, it is therefore better to have a separate function for that purpose. The points are then dispatched to the transformation modules via the token network.

The transformation modules perform the matrix-vector multiplication, operating on 16 bits data but calculating, as a result, 32 bits data. The time needed by the DFP processor to perform this task is almost twice as much as reading the data from Image Memory; this is the reason why two instances of the same function are in use. The two corresponding DFP-s get the next point alternatively from the supervisor; using this approach the two instances *together* still keep at the rate of reading produced by the supervisor.

At the lowest level we find four instances of a divider function; this latter has the task of performing the projective division on a point and to store the result in Image Memory again. The reason for using four instances altogether is the same as before: the time required by this function is roughly twice as much as for the transformer; by getting the point alternatively again, there is no loss of speed.

As a result, the speed of the whole transformation function is determined by the speed of the supervisor; actually, this latter one can produce ca. 150.000 points/sec. As we have seen before, the whole DFC contains, in the present version, *two* of these functions (that is all functions in Figure 8 are duplicated); this results in a theoretical possibility of transforming 300.000 points/sec.

Finally, each DFP of Figure 8 contains a dataflow program in a "classical" sense. A simplified portion of the matrix-vector multiplication may be seen in Figure 9; in fact, the matrix-vector multiplication may be considered as a typical example of a calculation which may be optimally structured for

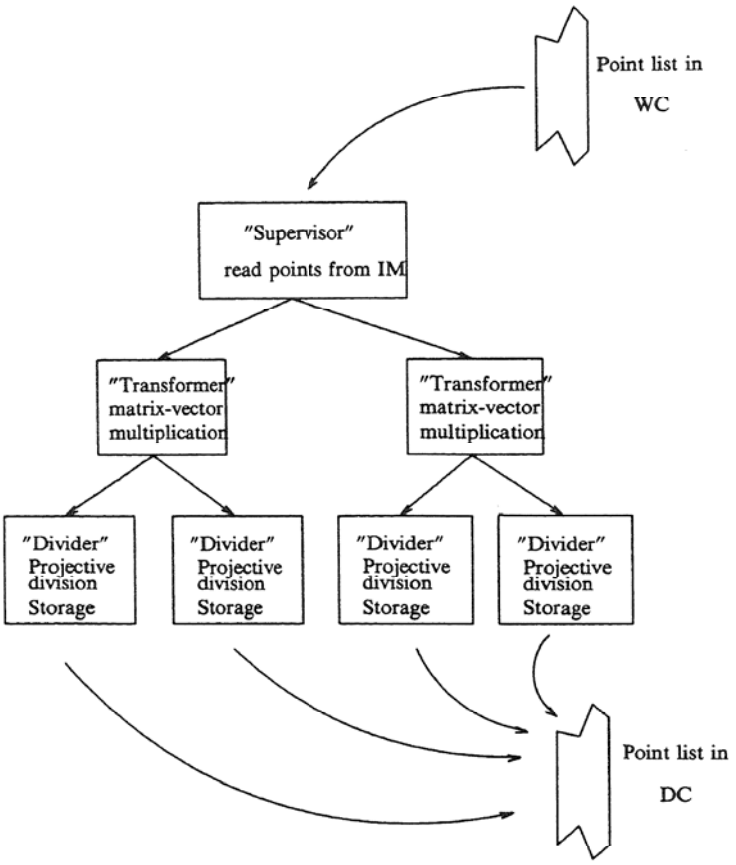

Figure 8 : The transformation function

dataflow purposes. The reason why this dataflow net is relatively complicated is that the DFP instruction set is centered around 16 bits arithmetic whereas 32 bits is needed for our purposes. Here is a short description of the net.

The x,y,z and w coordinates are duplicated 4 times in the first row to use them for the final x',y',z' and w' coordinate values. The multiplication instruction has the additional feature (not shown in the figure) that it may read values from internal memory "cyclically", in other words it will read first the element belonging to the first row of the matrix, then the second one, the third one and finally the fourth one. This ensures that after the multiplication the flow of tokens represent the right multiplicative factors.

The "multiply" as well as the "add" node may (on request) output two tokens: one for the high 16 bits value and one for the low 16 bits. These values must be successively accumulated to get the final high and low value of the result. This is what is done in the net of "add" nodes. Note that if only one output token is requested for an "add" or "multiply" node, this means the output of the low value only.

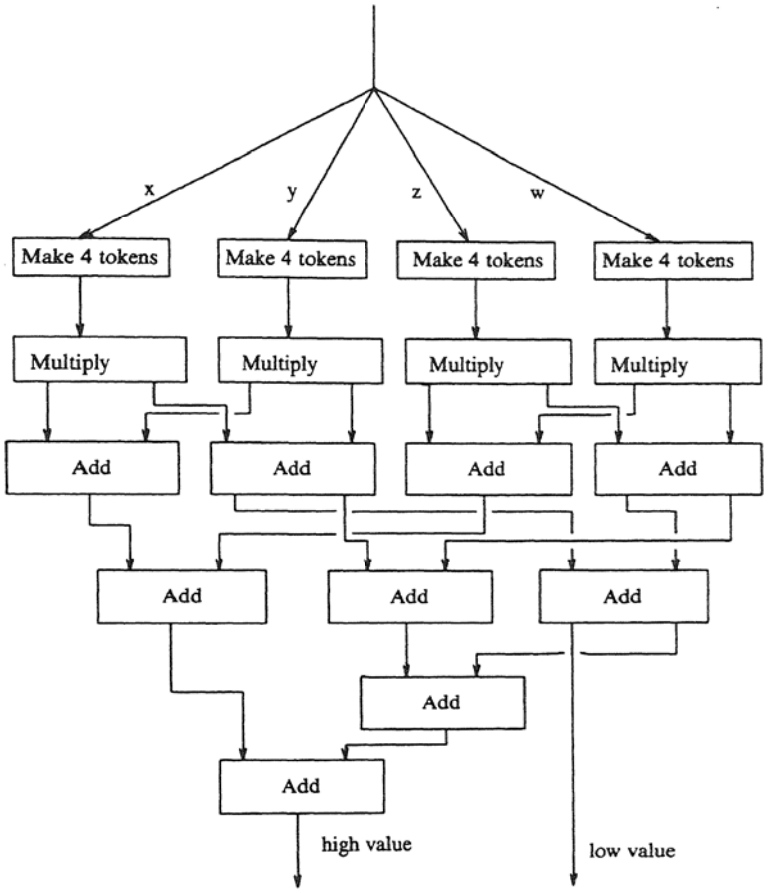

Figure 9 : Matrix-vector multiplication in dataflow

3.3. The Host software

As we have already mentioned, a rather complicated device driver is necessary on the host side to drive DFC. This device driver will serve as a basis for more elaborate software systems which are aimed at running on the Host. For this purpose, we plan to adapt GKS-3D[12], PHIGS+[11], and a three dimensional extension of the so-called radical system[13] on the top of this driver. As a later step, PEX[14] should also be ported, as soon as a public domain software becomes available for that purpose. The shielding facility of the graphics engine gives a particular help for the adaptation of window-oriented environments like PEX. Details of these adaptations go far beyond the scope of the present paper (see also Figure 5).

4. Conclusions

It is fairly difficult to give an impartial evaluation on the performance of graphics workstations; there is a lack of reliable bench-mark software for this purpose. What we expect in our case is that the overall graphics performance of the DFC will be around 150.000 to 200.000 Gouraud shaded triangles/sec, with peaks raising up to 230.000 triangles/sec. This speed is comparable to the highest end of the graphics super-workstations available on the market today.

The fact that the expected speed is not higher than the speed of some commercially available workstations should not be interpreted as a failure. All these workstations use well established technology, whereas DFC is the first exemple of a graphics workstation exploring the possibilities of dataflow technology as adapted for graphics. What we have to realise is that the basic processing element, that is the NEC μPD72181 itself is also the first commercially available dataflow processor on chip; in other words some improved versions are to be expected soon. Here are some aspects of such improvements we would welcome:

- The instruction set should be richer. In particular, division as well as floating point operations should be included.
- All internal memories should be significantly larger.
- A better internal structure should ensure higher on-chip parallelism.
- The overhead of accessing the Image Memory via the μPD9305 should be reduced (we should remark that a new and faster version of this chip is already in production, although still in prototype; the present hardware is prepared to use this chip instead of the older one).
- The way of connecting several processors should be improved. It should be possible to create large dataflow-nets where the fact whether a token is sent from one processor to the other or that it remains on-chip would be immaterial as far as software is concerned. This would allow a much easier way of programming the processor.

In addition to the improvements on the chip level itself, there might be improvements on the software level as well. Our personal experience in dataflow programming was practically non existent when starting the project and we have gained some experience only "run-time". It might well be that if we began reprogramming the whole DFC today, we could find new ways to improve the overall result just by using better programming techniques.

It is by itself an enormous advantage of this architecture that such reprogramming may be easily realized in contrast to an architecture which is based on custom design VLSI chips which is difficult and expensive to change. Furthermore, an eventual adaptation of the hardware-software environment for application-specific purposes is also easily realisable by reprogramming the dataflow processors. Merits of such flexible architectures have already been presented by a number of authors (see eg. England[15]).

By taking all these factors into consideration, an improved version of the DFP as well as a more experienced programming could raise the speed significantly; a fairly pessimistic estimation would still indicate a speed factor of 5 to 10, in comparison to the actual data. On the other hand, in addition to the improved speed of the already existing functions, new functions could also be added to the Dataflow Engine. e.g. some parts of the Host device driver, more elaborate shading calculations etc. It is almost impossible to measure exactly all impacts of such improvements.

5. Acknowledgements

We are grateful to all participants of the project who have contributed to the development of the ideas presented in the paper and who have also taken an active part in the actual development. We should cite the names of Frans van der Markt (DTN), Fons Kuijk, Bert Rouwhorst, Bèhr de Ruiter, Robert van Liere, Markus van Dijk (CWI) and Arthur Veen (Parallel Computing). Without their active role, this project would have no chance of success.

References

1. A.H. Veen (1986). Dataflow Machine Architecture, *ACM Computing Surveys*, 18.

2. J. HERATH, Y. YAMAGUCHI, N. SAITO, AND T. YUBA (1988). Dataflow Computing Models, Languages, and Machines for Intelligence Computations, *IEEE Transactions on Software Engineering*, 14.
3. NEC (1986). *μPD7281 Image Pipelined Processor, Product Description*, NEC Electronics.
4. S.S. ABI-EZZI AND M.A. MILICIA (1986). , in *Data Structures for Raster Graphics*, ed. L.R.A. Kessener, F.J. Peters and M.L.P. van Lierop, EurographicSeminar Series, Springer Verlag.
5. H.R. FINCH, M. AGATE, A.A. GAREL, P.F. LISTER, AND R.L. GRIMSDALE (1988). A Multiple Application Graphics Integrated Circuit MAGIC II, in *Advances in Computer Graphics Hardware II*, ed. A.A.M. Kuijk & W. Strasser, EurographicSeminar Series, Springer Verlag.
6. J. EYLES, J. AUSTIN, H. FUCHS, T. GREER, AND J. POULTON (1988). Pixel-Planes 4: A Summary, in *Advances in Computer Graphics Hardware II*, ed. A.A.M. Kuijk & W. Strasser, EurographicSeminar Series, Springer Verlag.
7. I. PAGE (1983). DisArray: A 16×16 RasterOp Processor, in *Eurographics'83 Conference Proceedings*, ed. P.J.W. ten Hagen, North-Holland.
8. TH. THEOHARIS AND I. PAGE (1988). Incremental Polygon Rendering on a SIMD Processor Array, *Computer Graphics Forum*, 7.
9. S. DEMETRESCU (1985). High Speed Image Rasterization Using Scan Line Access Memories, in *Proceedings of the 1985 Chapel Hill Conference on VLSI*, ed. H. Fuchs, Computer Science Press.
10. P.J.W. TEN HAGEN, A.A.M. KUIJK, AND C.G. TRIENEKENS (1987). Display Architecture for VLSI-based Graphics Workstations, in *Advances in Computer Graphics Hardware I*, ed. W. Strasser, EurographicSeminar Series, Springer Verlag.
11. (1988). PHIGS+ Functional Description, Revision 3.0, *Computer Graphics*, 22.
12. ISO (1988). *Information processing systems - Computer graphics - Graphical Kernel System for Three Dimensions (GKS-3D) functional description,* ISO 8805 .
13. P.J.W. TEN HAGEN AND H.J. SCHOUTEN (1987). Parallel Graphical Output from Dialogue Cells, in *Eurographics'87 Conference Proceedings*, ed. G. Maréchal, North-Holland.
14. W. CLIFFORD, J.I. MCCONNELL, AND J. SALTZ (1988). The Development of PEX, in *Eurographics'88 Conference Proceedings*, ed. D.A. Duce & P. Jancène, North-Holland.
15. N. ENGLAND (1986). A Graphics System Architecture for Interactive Application-Specific Display Functions, *IEEE Computer Graphics and Applications*, 6.