MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ARTIFICIAL INTELLIGENCE LABORATORY

Working Paper No. 163 May 1978

APPLICATION OF DATA FLOW COMPUTATION TO THE SHADED IMAGE PROBLEM

Thomas M. Strat

Abstract. This paper presents a method of producing shaded images of terrain at an extremely fast rate by exploiting parallelism. The architecture of the Data Flow Computer is explained along with an appropriate "program" to compute the images. It is shown how shaded images of terrain can be computed in less than one-tenth of a second using a moderate-sized Data Flow Computer.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

The system presented in this paper is capable of producing shaded, orthographic projections of terrain using a Data Flow Computer. Its purpose is to provide video input to a flight simulator. This requires that pictures be produced in real time (i.e. less than 0.5 seconds per frame, anyway). Current implementations in LISP on a conventional computer require at least one minute per frame. Thus there is a genuine need to overcome this time barrier.

The paper first describes the structure of the algorithm. This is followed by a brief description of the structure of the Data Flow Computer on which the program is to run. Finally, the actual Data Flow Program is presented along with an analysis of its performance.

II. DETAILS OF THE PROBLEM

INPUT

The input to the algorithm is a two-dimensional array of elevations of terrain (called a Digital Terrain Model, abbreviated DTM). These are obtained a priori by interpolation of an ordinary contour map. Each DTM contains about 256 x 256 grid points. The conventional algorithm uses this data in the form of a two-dimensional LISP array. For a

Data Flow implementaion, the DTM would be stored as a list of lists where each list is converted to a stream corresponding to a single column in the DTM. This representation will permit us to exploit some of the inherent parallelism as will be seen later.

OUTPUT

In the conventional algorithm the output is a two-dimensional LISP array of intensity values at each point (pixel) in the image. These values can be displayed by a CRT or other means to generate the picture. The Data Flow Machine would generate a set of streams where each stream is the sequence of intensity values for each pixel in a given column of the image array. The streams are then converted to array form for display.

THE ALGORITHM

For purposes of simplicity in explanation, we will restrict attention to viewpoints with only one degree of freedom--namely, fixing the rotation angle and varying the angle of elevation. If the coordinates of the DTM are x and y, the view direction will be perpendicular to the x axis and in the direction of y.

The mapping from input to output is accomplished by two independent calculations:

1) Each surface point (x, y, z) is mapped into a unique image point (i, j) using

the appropriate transformation and projection. Details are omitted here in order to focus attention on data flow issues [Strat, 1978].

2) The intensity that should be displayed at that point is calculated. It can be shown that the intensity is a function of the view direction, the sun's direction, and the surface normal at the point.

COMPLICATIONS

The algorithm is complicated by the age-old Hidden Line Problem. That is, some terrain points do not affect any image point since they are obscured by mountains or hills closer to the viewer. *Hidden-line elimination* is accomplished essentially for free by a sneaky trick. We simply generate the picture from the foreground toward the horizon. Whenever we encounter a surface point that wants to be displayed at an image point that has already been displayed, we conclude (correctly) that it is a point on the surface which is blocked from view by a hill closer to the viewer and ignore it.

Another complication is the fact that projecting all points in the DTM into points in the image does not guarantee that all image points will be found. Thus we will have a picture with "holes". The remedy is to *interpolate* intensity values for the missed points. The interpolation proposed for the data flow implementation is simple but by no means optimal (nor correct). The holes are simply filled by repetition of the intensity value directly above each hole. In practice, the results are close enough to the correct values so that the resulting picture appears correct.

III. HIGH-LEVEL IMPLEMENTATION

PARALLELISM

Exactly what parallelism is there?

1) Processing of each column in the DTM can be performed independently.

2) Calculation of the projection of the surface point onto the image plane is independent of the calculation of the intensity to be displayed there.

3) Calculation of the projection and intensity of each individual point is independent of every other projection and intensity calculation.

Type 3 parallelism is not exploited in the data flow implementation presented here because the speedup afforded by 1 and 2 alone is sufficient for the problem at hand (See Section VI - Performance Analysis). Adding type 3 parallelism would put storage requirements out of reach.

THE COMPUTATION

Let us focus attention on the processing of each column. Thus we have the xth column of the DTM which, when transformed, will yield the ith column of the image. Note that the lengths of these two columns are not equal in general. We will represent the xth column of the DTM as a stream called TERRAIN. Then the projection is accomplished by a functional mapping of the elements of TERRAIN into a stream called J such that the kth element of J is the value of j at which the kth element of TERRAIN is to be displayed.

Simultaneously, the intensity can be calculated for each element in TERRAIN. To simplify this calculation we will assume the surface normal to have been precomputed for every point in the DTM. The surface normal at a point can be represented as an ordered pair (p, q) where p is the partial derivative of z with respect to x and q is the partial derivative of z with respect to y. Then the surface normals of the points in a column of the DTM can be represented by two streams P and Q. The intensities can be represented as a stream called INTENSITY calculated as a function of P, Q, and the sun and eye locations. To further simplify things, we will assume the surface to have a special reflectance property known as lambertian. Lambertian surfaces reflect light in a way that is independent of the viewing direction. Thus we can eliminate the eye location from our calculation of the intensity to be displayed.

THE PROGRAM

This section presents the algorithm in an undocumented language that was developed for data flow systems. The main procedure is called SHADE. XHI and YHI are the dimensions of the DTM. SINEYE and COSEYE define the view angle, such that if θ is the angle of elevation, SINEYE=sin(θ) and COSEYE=cos(θ).

```
SHADE: procedure (DTM, P, Q:array[array[real]], XHI, YHI:int,
                  XSUN, YSUN, ZSUN, SINEYE, COSEYE:real
                  returns array[array[int]]);
    return forall X:int in (1, XHI)
                  construct IMAGE:array[array[int]]
                  where IMAGE[X] := COMPOSE(
                                    PROJECT(DTM[X], YHI, SINEYE COSEYE),
                                    REFLECT(P[X], Q[X], YHI, XSUN, YSUN, ZSUN));
end SHADE;
```

COMPOSE is designed to handle hidden line (point) elimination and interpolation. The array, INTENSITY, supplies the candidate values to be placed in the image. COMPOSE eliminates hidden points by skipping values from INTENSITY and interpolates (fills the holes) by repeating INTENSITY values according to the information in J.

```
COMPOSE: procedure (J:array[int], INTENSITY:array[int] returns array[int]);
        for Y:int := 1, T:int := 0, IMAGE:array[int]:=empty          %T is threshold
                if J[Y] > T then iter Y, T+1, append(IMAGE, INTENSITY[Y])
                else iter Y+1, T, IMAGE;                             %Skip hidden point
        return IMAGE;
end COMPOSE;
```

The procedures PROJECT and REFLECT accomplish what their names imply. PROJECT provides the pixel in the image to which each point in the DTM maps. REFLECT gives the intensity value to be displayed at that pixel based on the lambertian reflectance function.

```
PROJECT: procedure (TERRAIN: array [real], YHI:int, SINEYE, COSEYE:real
                    returns array[int]);
        return forall Y:int in (1, YHI)
                    construct J:array[int]
                    where J[Y] := fix(COSEYE * TERRAIN[Y] + SINEYE * float(Y));
end PROJECT;

REFLECT: procedure (P, Q:array[real], YHI:int, XSUN, YSUN, ZSUN: real
                    returns array[int]);
        return forall Y:int in (1, YHI)
                    construct INTENSITY:array[int]
                    where INTENSITY[Y] := fix(256.0 * DOT(XSUN, YSUN, ZSUN
                                                NORMALIZE(P[Y], Q[Y], 1.0)));
end REFLECT;
```

XSUN, YSUN, and ZSUN define the direction to the sun and are assumed to be normalized. DOT and NORMALIZE are functions which compute the dot product and normalization of ordered triples.

```
DOT: procedure (A, B, C, D, E, F:real returns real);
                return (A*D + B*E + C*F);
end DOT;
```

```
NORMALIZE: procedure (A, B:real returns real,real,real);
                D:real := sqrt(A*A + B*B + 1.0);
                return  (A/D), (B/D), (1.0/D);
end NORMALIZE;
```

Figures 1 and 2 contain a schematic of the organization of the Data Flow algorithm. They illustrate the concurrency that is being used in the proposed implementation.

MODULAR ORGANIZATION OF THE ALGORITHM

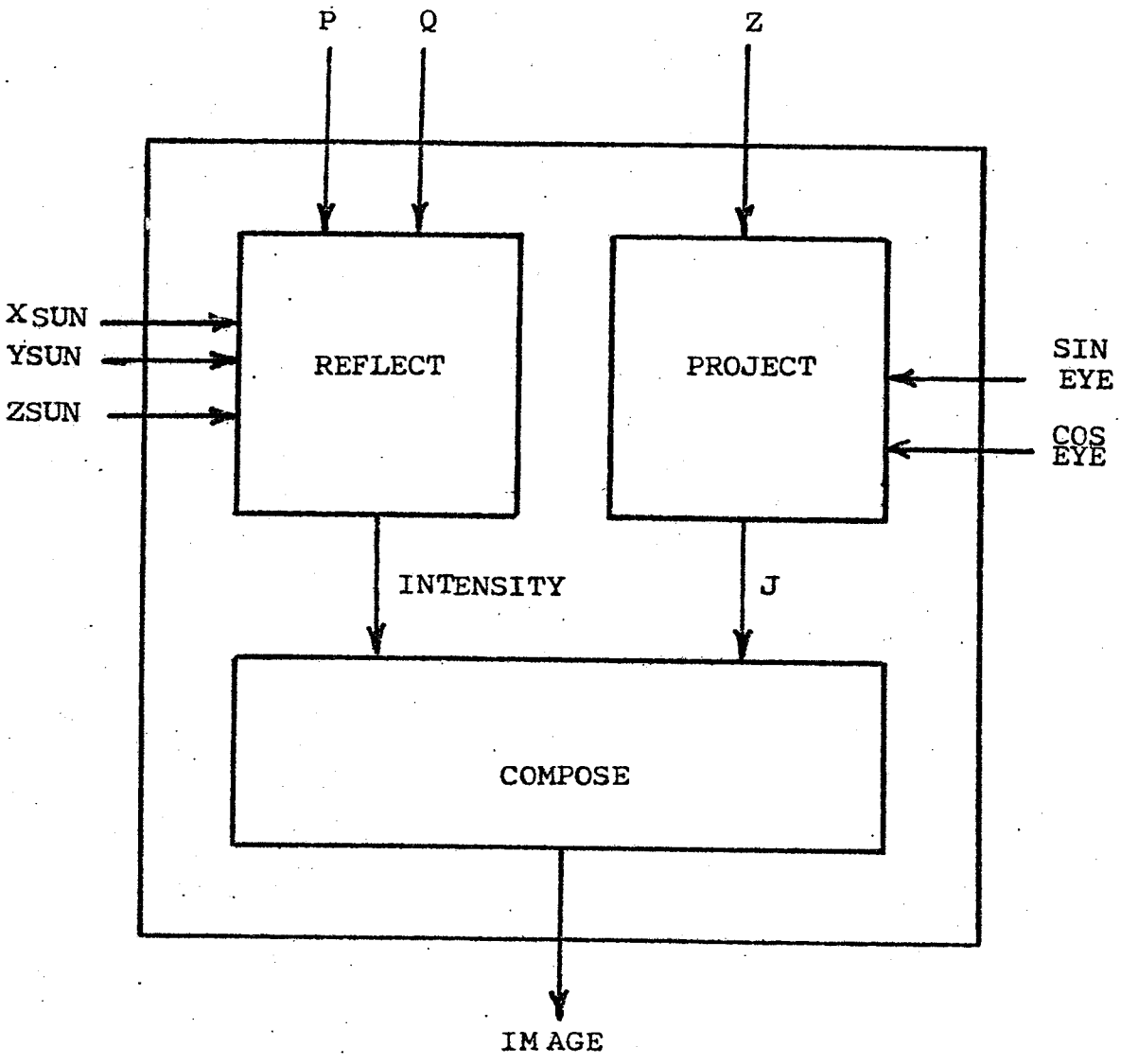


Figure 1

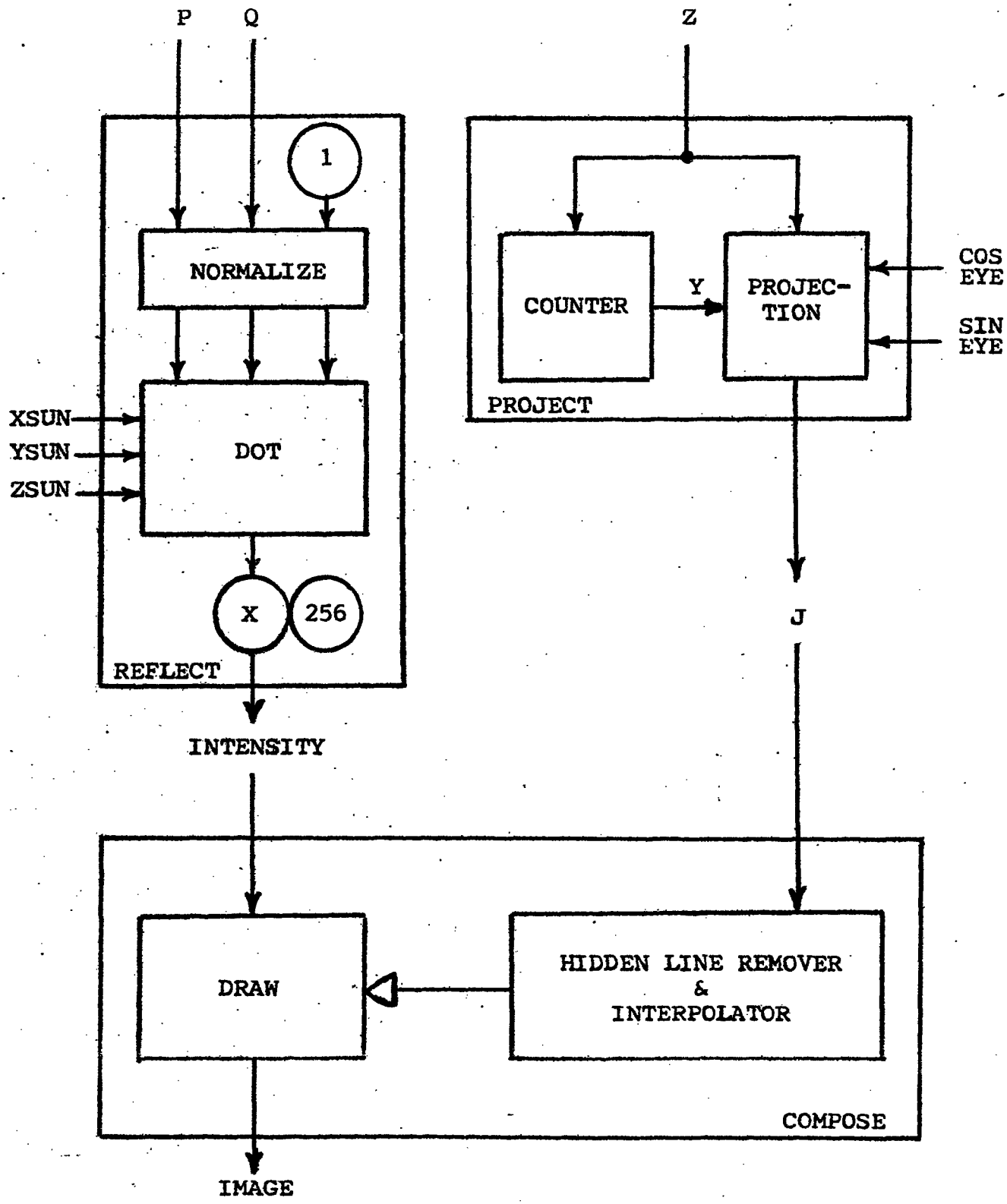

P
Q
Z
1
NORMALIZE
COUNTER
Y
PROJEC-
TION
COS
EYE
SIN
EYE
PROJECT
XSUN
YSUN
ZSUN
DOT
X
256
REFLECT
J
INTENSITY
DRAW
HIDDEN LINE REMOVER
&
INTERPOLATOR
COMPOSE
IMAGE

Figure 2

IV. THE DATA FLOW COMPUTER

The Data Flow Program described in the previous section is designed to run on a Level II Data Flow Computer. This machine, which appears in Figure 3, provides support for data structure operations in addition to the basic scalar operations and control mechanisms.

Each Instruction Cell in the Instruction Memory holds one instruction corresponding to one actor in a data flow program. Once an Instruction Cell has received all required operand values and acknowledge signals from the Distribution Network, the Cell is considered to be enabled and sends its contents in the form of an operation packet through the Arbitration Network to the appropriate Processor. The result packet produced by the Processor is transmitted through the Distribution Network to the Instruction Cells which require it as an operand, and acknowledge signals are sent to control the enabling of cells. Even though roughly 20 microseconds may be required for an instruction to be enabled, sent to the Processing Section, executed, and the results transmitted back to other Instruction Cells, the computer is capable of high performance because a large number of instructions may be in various stages of execution simultaneously [Dennis and Weng, 1977].

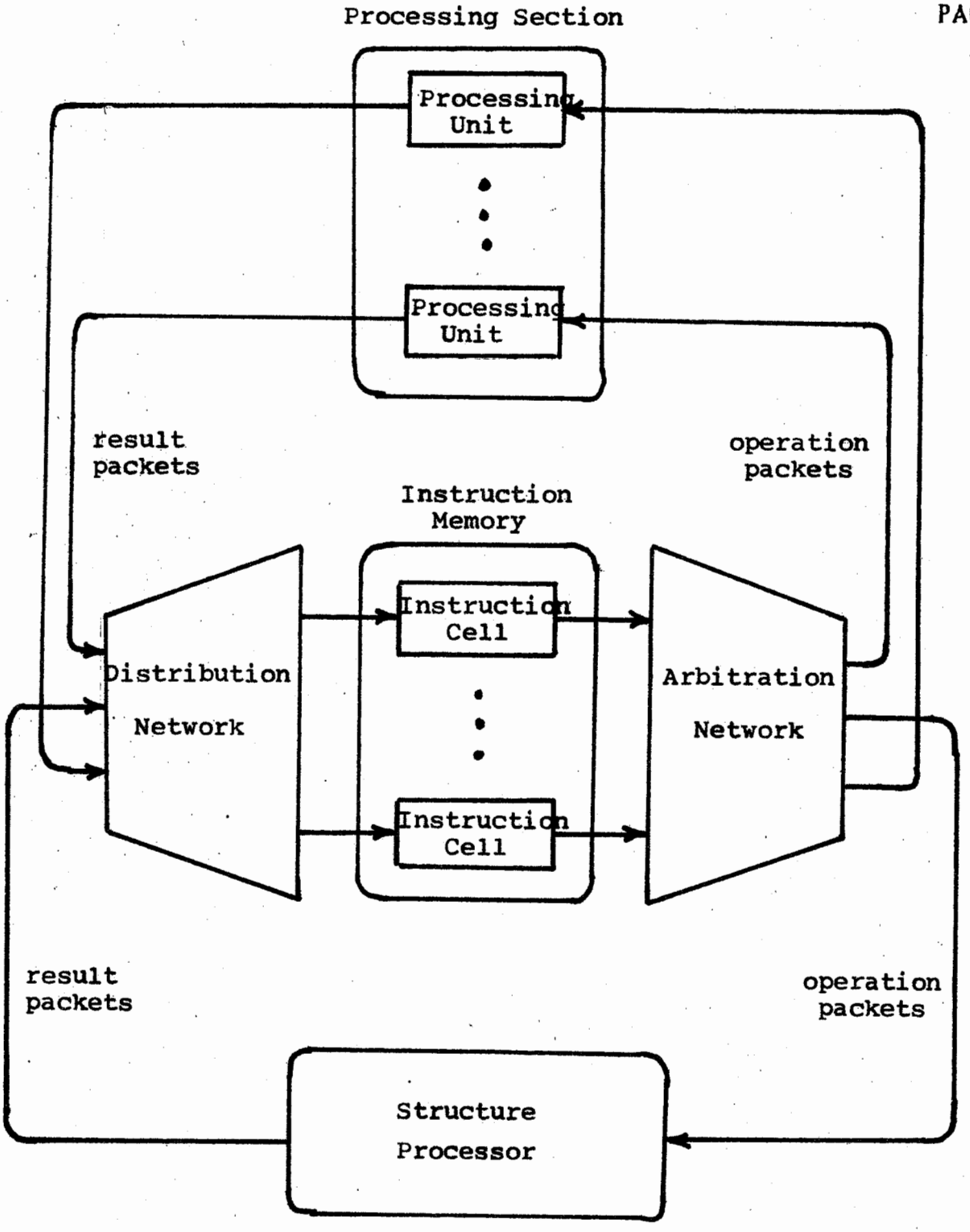


Figure 3. The Data Flow Computer

from Dennis, Misunas and Leung

page 19

V. THE STRUCTURE PROCESSOR

The Structure Processor is the unit that separates the Level I from the Level II Data Flow Computer. Figure 4 shows the organization of the Structure Processor that is assumed in this paper. It consists of a Packet Memory System and three units--the Interpret, Queue, and Transmit units.

The Packet Memory holds representations of data structures and is responsible for providing the means for storing and accessing their components, and for garbage collection. It associates a unique identifier with each structure that serves to represent that structure in all units outside the Structure Processor.

The function of the Structure Controller is to implement the data structure operations. There are three legal operation packets that it may receive--*create*, *select*, and *append*. The Interpret unit interprets these packets producing sequences of commands that it sends to the Packet Memory System. The Queue is used to store select packets while they await their values to be retrieved from memory. Result and acknowledge packets are generated by the Transmit Unit from entries containing retrieved values as they reach the end of the Queue. These packets are routed through the Distribution Network to Instruction Cells as called for by the instructions in operation packets. The Queue is necessary to assure that result packets are sent out in the same order in which their associated operation packets arrived. Otherwise the components of the arrays would be incorrectly indexed, and the program would not be determinate [Dennis and Weng, 1977].

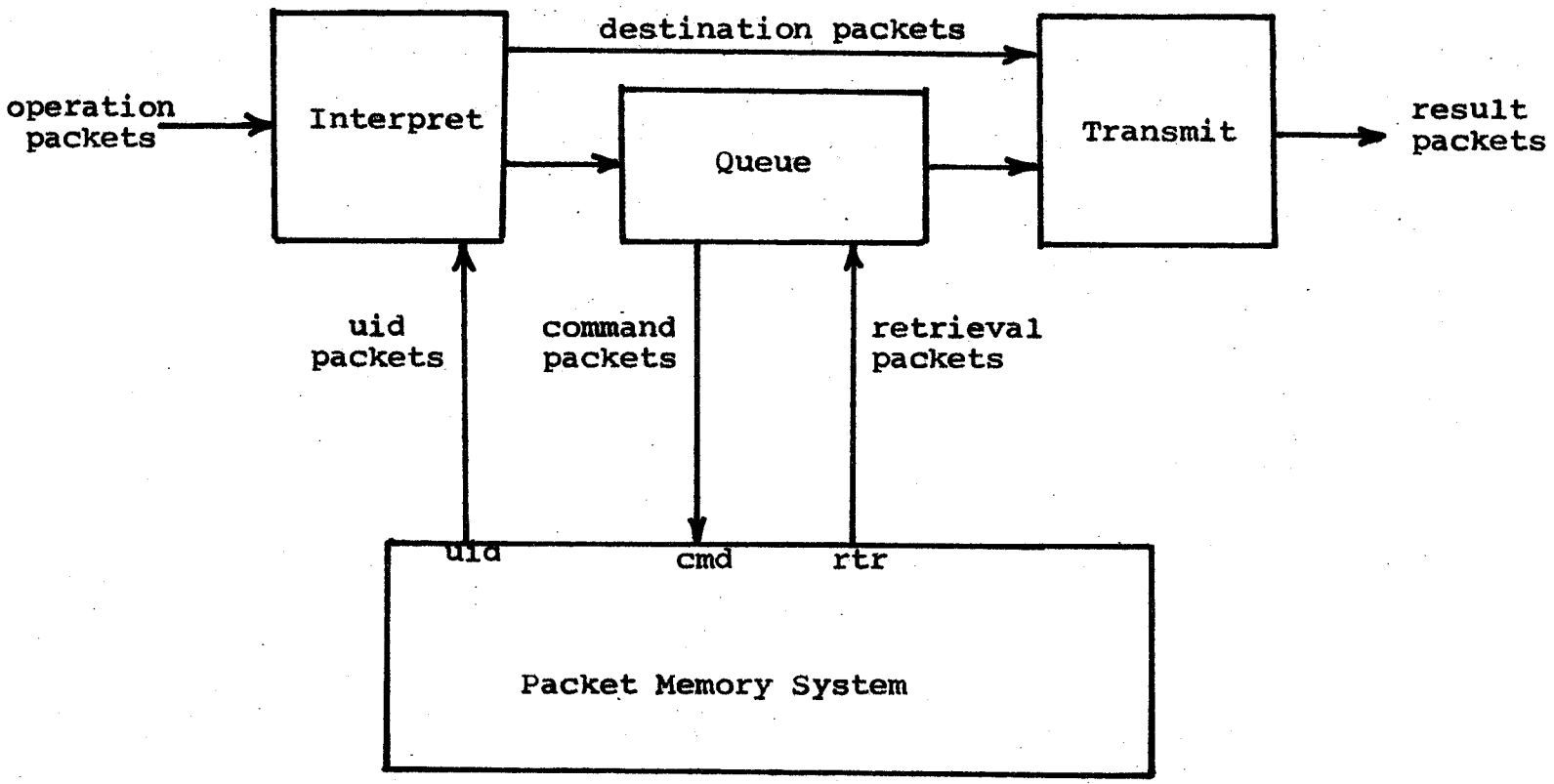


Figure 4. The Structure Processor

from Dennis and Weng, page 8

VI. THE DATA FLOW GRAPH

In order to run on a Data Flow Computer, a high-level program such as the one presented in Section III must be translated to a machine-executable data flow program. In practice, a translation program [Dennis, Misunas and Leung, p. 3] would do this automatically but such a program has not yet been written. This section shows the data flow version of the algorithm as a Data Flow Graph. Although this graph is unsafe, it is left that way in the interest of clarity.

The GET module is used to convert the arrays stored in memory to streams which can be operated upon. GET is easily implemented since all values in the arrays are used in order. Its Data Flow Graph appears in Figure 5. Figure 6 is the PUT module which receives elements from COMPOSE in the form of a stream and stores them in an array in the Structure Memory. Note that each column requires three GET's (for DTM, P, and Q) and one PUT (for IMAGE).

The Data Flow Graphs for PROJECT and REFLECT are shown in Figures 7 and 8 respectively. COMPOSE appears in Figure 9. The stream J is used to decide which values of intensity to pass on to IMAGE and which to gobble up by comparing J to the internally stored threshold, T. At any point in the computation, T is the highest value of J received so far.

The entire program consists of a copy of each of these modules for every column to be processed. The outputs of IMAGE from each column form the picture when displayed together.

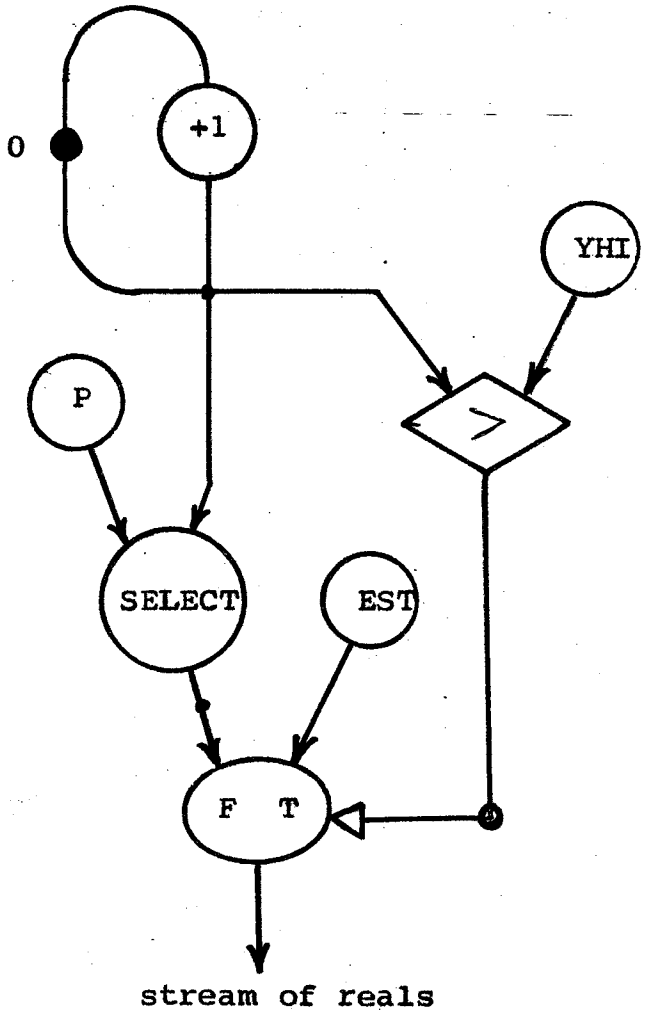


Figure 5. THE GET MODULE

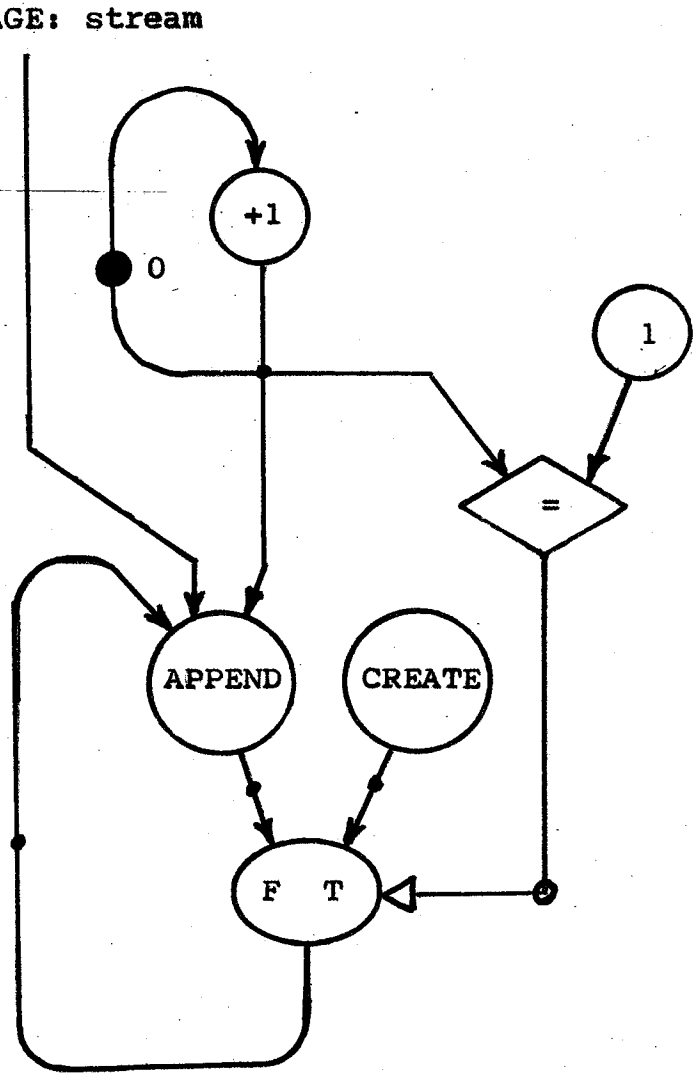
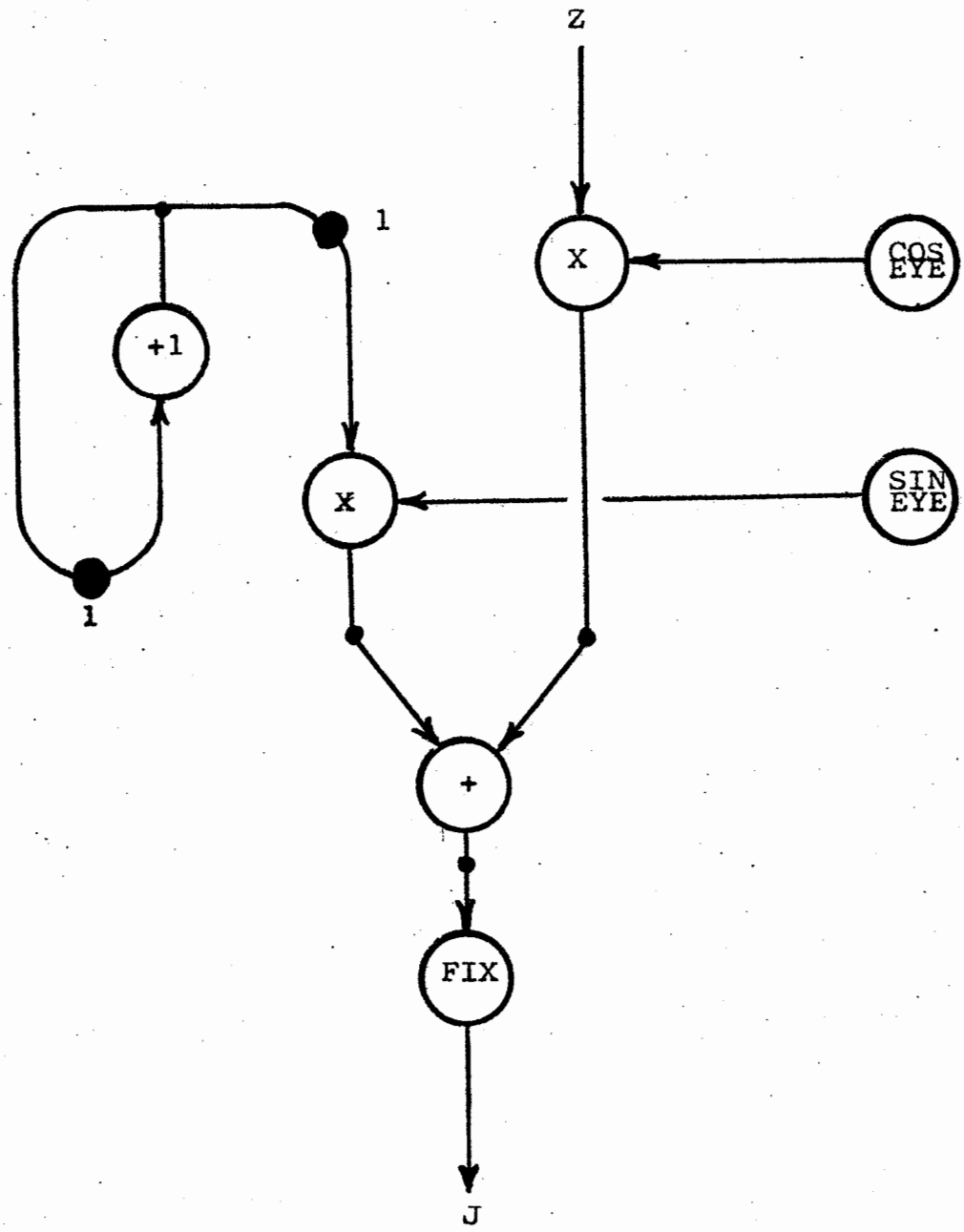


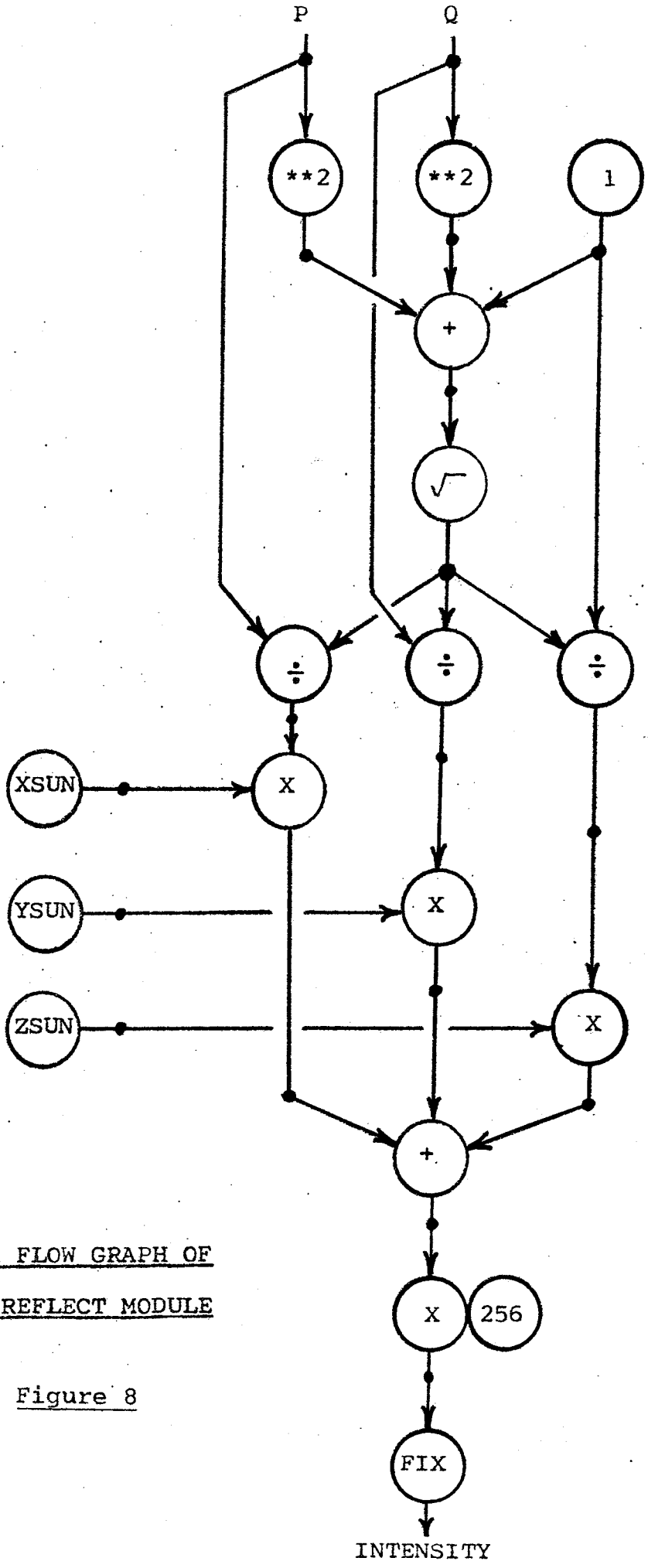
Figure 6. THE PUT MODULE

DATA FLOW GRAPH OF THE PROJECT MODULE



dots indicate initial token placements.

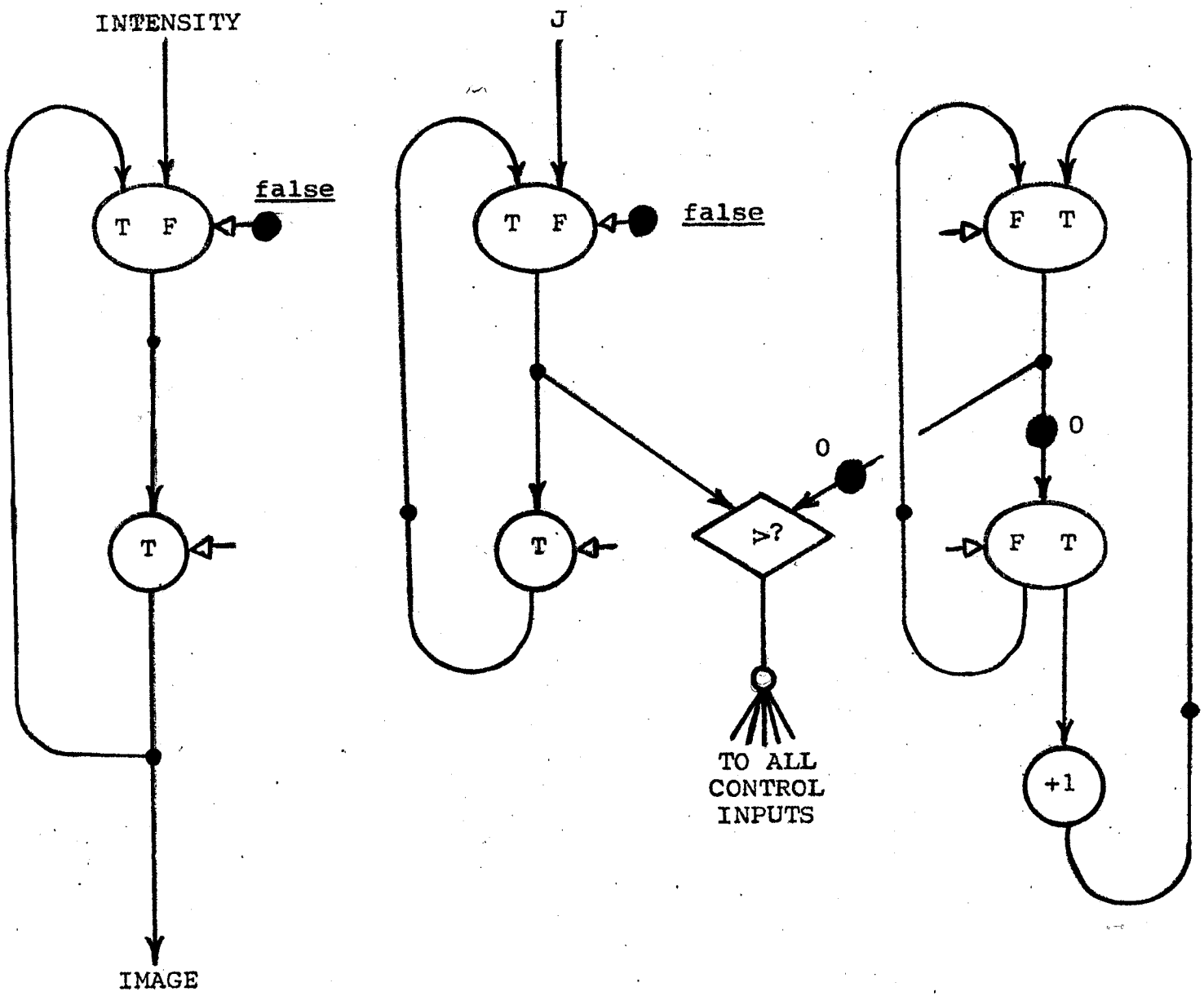
Figure 7.



DATA FLOW GRAPH OF

THE REFLECT MODULE

Figure 8



DATA FLOW GRAPH OF THE COMPOSE MODULE

FIGURE 9.

VII PERFORMANCE ANALYSIS

The goal of this section is to determine how much storage is required and at what rate pictures can be drawn. For purposes of discussion, we will assume that the dimensions of the Digital Terrain Model are 500 x 500. Thus 25,000 words must be stored for the DTM in the Structure Memory. The arrays P and Q which give the surface normal at each point are the same size as the DTM and require storage of another 50,000 words. The IMAGE array has the same width as the DTM but the height is dependent on the view angle. The largest picture is produced when the view angle is perpendicular to the surface and has dimensions equal to the DTM. Therefore, the Structure Memory must hold a total of 100,000 words.

The chart in Figure 10 shows the number of actors in each module of the computation. Since the columns of the DTM are processed concurrently, there must be a copy of each module for each of the 500 columns. These are analogous to the butterflies of the FFT Computation [Dennis, Misunas and Leung, 1977]. Since 42 Instruction Cells are required for each column, 21,000 Instruction Cells are required overall.

Now consider execution time. For each column, the computation requires processing 6000 multiplications or divisions; 4500 additions or subtractions; 8500 miscellaneous operations; and 2000 data structure operations. For the entire image then, 3 million multiplications or divisions; 2.25 additions or subtractions; 4.25 million miscellaneous operations; and 1 million data structure operations are needed. If the Data Flow Computer is to complete this processing in 0.1 seconds,

the scalar processors must be able to handle operation packets at the rate of 130 MHz and the Structure Processor must be capable of handling data structure operations at 10 MHz. These rates may be achieved by using many processors and structuring the Arbitration and Distribution Networks for concurrent transmission of many packets. For instance, if a multiplier can process packets at a throughput of 400 ns [Dennis and Weng 77], then I can process my 3 million multiplications in 0.1 seconds using 12 multipliers, assuming I can keep all the multipliers busy all the time. While this may in fact not be the case, the structure of the algorithm leads me to believe that I can come pretty close. At any rate, the addition of a few more multipliers ought to relieve any temporary backlogs of demand for multipliers. With nine 400 ns adders and a composition of about sixteen of the 200 ns miscellaneous processors, I should be able to produce a picture in 0.1 seconds.

Switching attention to the Structure Processor, we see that the memory access time for retrieval requests must not be so large that the values from the arrays are not available when they are needed. Since a picture is to be completed only once in 0.1 seconds, this requirement is easliy met. However, the Queue module of the Structure Controller must be large enough to hold all retrieval requests which have not been completed by the Packet Memory. For the arrival rate of 10 MHz, even a one millisecond retrieval delay necessitates a capacity of 10,000 entries in the Queue. Implementing the Packet Memory using storage devices which have an access time under a millisecond would decrease the required size of the Queue proportionally.

To summarize, a 500 x 500 pixel image can be computed from a 500 x 500 Digital Terrain

Model in 0.1 seconds using 28,000 Instruction Cells, about 40 Scalar Processors, and a 100K word Structure Memory.

TYPES OF ACTORS

	*/÷	+/-	MISC SCALAR	DATA STRUCTURE	INSTR CELLS
GET P	0	1	2	1	4
GET Q	0	1	2	1	4
GET DTM	0	1	2	1	4
PROJECT	2	2	1	0	5
REFLECT	10	2	1	0	13
COMPOSE	0	1	7	0	8
PUT IMAGE	0	1	2	1	4
TOTAL	12	9	17	4	42

Figure 10

VIII. CONCLUSION

In this paper I have explained the details of an algorithm for producing shaded oblique views of terrain using a Data Flow Computer. Recall that these images are to be used as visual input to a flight simulator which is why they have to be produced quickly. As shown in the previous section, the images could be produced within 0.1 seconds on a reasonably sized Level II Data Flow Computer. However, several issues were glossed over in that discussion.

Is it reasonable to assume that the processors could be kept busy? Any idle time in any processor can directly cause the entire algorithm to run much more slowly due to the data dependencies between actors. In order to keep the processors busy, there must be enough Instruction Cells enabled at any given time to continually feed the processors. Because there are 500 computations being performed independently (there is no data dependency between columns) and only 12 multipliers in the proposed system, it seems reasonable to assume that there will always be enough Multiply Instructions enabled assuming the program is live and there are no backlogs elsewhere.

Can the program execution ever become deadlocked? In order for an Instruction Cell to become enabled, it must receive all its operands from result packets through the Distribution Network. If the Distribution Network ever becomes clogged, the enabling of Cells will be inhibited and the entire operation might grind to a halt. Thus it is necessary to eliminate this potential bottleneck by designing the Distribution Network appropriately.

Is the Structure Processor fast enough? The power of the Structure Processor lies in its ability to retrieve data concurrently yet return results in the correct sequence. Thus even a Structure Processor with a slow memory can have a high throughput because of concurrent operations on that memory.

To be used as input to a flight simulator, pictures must be produced every 0.1 seconds, not just one picture in 0.1 seconds. Is this possible? Thankfully, it is not necessary to reload the Instruction Cells since the program never changes. However, one must be careful to design the system such that consecutive executions are possible. (This was not done in this paper.)

Based on all evidence I've encountered, I conclude that producing shaded images of terrain using a Data Flow Computer is feasible for providing visual input to a flight simulator.

In an actual implementation, I visualize using a huge DTM (maybe 10,000 x 10,000?) and producing an image based on a small portion of it (say, 200 x 200). Then any image would be specified by the view angle (XEYE, YEYE, ZEYE) and the region (XLO, XHI, YLO, YHI) to be displayed according to the position and course of the simulated aircraft. Additionally, there would be a special procedure which "paints" on the image features such as runways, rivers, and highways. The end product would be a motion picture of a somewhat natural-looking scene.

IX. BIBLIOGRAPHY

Ackerman, W.B., "Interconnections of Determinate Systems", Computation Structures Note 31, Laboratory for Computer Science, M.I.T., July 1977.

Ackerman, W.B., "A Structure Memory for Data Flow Computers", MIT/LCS/TR-186, August 1977.

Dennis, J.B., "First Version of a Data Flow Procedure Language", MIT/LCS/TM-61, May 1975.

Dennis, J.B., "A Language Design for Structured Concurrency", Computation Structures Note 28-1, Laboratory for Computer Science, M.I.T., February 1977.

Dennis, J.B., Misunas, D.P., Leung, K.C., "A Highly Parallel Processor Based on the Data Flow Concept", Computation Structures Group Memo 134, Laboratory for Computer Science, MIT, January 1977.

Dennis, J.B., Weng, K.S., "Application of Data Flow Computation to the Weather Problem", Computation Structures Group Memo 147, Laboratory for Computer Science, MIT, May 1977.

Misunas, D.P., "A Computer Architecture for Data Flow Processor", S.M. Thesis, M.I.T. Department of Electrical Engineering and Computer Science.

Strat, T.M., "Shaded Perspective Images of Terrain", MIT Artificial Intelligence Memo 463, March, 1978.

Weng, K.S., "Stream Oriented Computation in Recursive Data Flow Schemas", MIT/LCS/TM-68, October 1975.