TR82-001

A Comparison of Two Graphics Computer Designs

by

Joseph Kitchings Parks

A COMPARISON OF TWO MULTIPROCESSOR GRAPHICS MACHINE DESIGNS

by

Joseph K. Parks

A Thesis submitted to the faculty of The University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science.

Chapel Hill

7 May 1982

Approved by:

Dr. Henry Fuchs
Advisor

Dr. Frederick P. Brooks, Jr.
Reader

Dr. Gyula A. Magó
Reader

ABSTRACT

JOSEPH KITCHINGS PARKS. A Comparison of Two Graphics Computer Designs (under the direction of Dr. Henry Fuchs).

Currently, three dimensional graphics systems with hidden surface removal and smooth shading are large, expensive, pipelined computers with many special purpose processors. Fred Parke and Henry Fuchs have introduced designs using general purpose microprocessors working in parallel, rather than pipelined fashion. Parke's scheme divides the display screen into contiguous chunks, and assigns each chunk to a processor. Fuchs' scheme assigns adjacent points on the screen to different processors, so that all processors work on every polygon.

Parke compared these designs assuming an even distribution of data over the screen, and found that splitting the screen into contiguous chunks is always superior. However, realistic data (such as landscapes or airplanes) are not distributed evenly.

This thesis presents a comparison of these designs using data from NASA's Space Shuttle flight simulator. We find that for few processors (say 4), the Fuchs' scheme is preferred; for hundreds of processors, the Parke scheme is preferred; and for an intermediate number (say 16), the designs are relatively equal.

ACKNOWLEDGEMENTS

I wish to thank James R. Smith of the Johnson Space Center and Richard Weinberg (now with Cray Research) for the Space Shuttle database. I also thank Greg Abram for the use of his polygon transformation and clipping program, and Dr. Frederick Brooks for being a sounding board during the whole project. Dr. Henry Fuchs has been a constant source of encouragement and direction. Most of all, I thank my wife, Sandy, who has acted out of love and grace, and not justice.

CONTENTS

ACKNOWLEDGEMENTS ii

Chapter page

I. INTRODUCTION 1

Motivation 1
Description of Project 3

II. ALGORITHM DESCRIPTION AND ANALYSIS 6

The Algorithm 6
Analysis of Visible Surface Algorithm 15
Limitations of this Analysis 17

III. MACHINE DESCRIPTION 19

The Parke Splitter Machine 19
The Fuchs Interlace Machine 23
Assumptions and Limitations 29

IV. SIMULATION RESULTS AND MACHINE COMPARISON 30

Parke's Comparison 30
The Analyzed Scenes and Their Results 34
Screen Complexity: Area vs. Number of Polygons . 46
Splitter's Sensitivity to Non-uniformly Distributed Data 51
Effects of Polygon Overhead on Interlace Scheme 57
Parke's Hybrid Scheme 58

V. CONCLUSIONS 59

Summary of Simulation Results 59
Conclusions 61
Further Research 61

REFERENCES 63

Appendix page

A. PROGRAM LISTINGS IN C 65

B. PROGRAM LISTINGS IN PDP-11 ASSEMBLER 77

Code from POLYBODY1.C and POLYBODY2.C 78
Code from EDGEBODY1.C and EDGEBODY2.C 82
Code from SEGMENTBODY1.C and SEGMENTBODY2.C . . 88
Code from PIXELBODY.C 90

C. STATISTICAL CHARACTERISTICS OF SELECTED SCENES . . 92

Uniprocessor 93
Splitter--4 Processor (slowest) 97
Interlace--4 Processor (slowest) 100
Splitter--16 Processor (slowest) 103
Splitter--16 Processor (shuttle processor) . . 105
Interlace--16 Processor (slowest) 108
Interlace--16 Processor (fastest) 111
Hybrid--16 Processor (slowest) 114

LIST OF FIGURES

Figure		page
1.	Traditional and "Parallel Micro" Designs	2
2.	Polygon Examples	8
3.	Tabular Polygon Representation	9
4.	Algorithm Values After Initialization	11
5.	Polygon on Sparse Grid	14
6.	3 Simple Division Schemes	19
7.	16-Processor Parke Splitter Machine	20
8.	Diagram of Parke Splitter Machine with 4 Processors	22
9.	1x4 Interlace Pattern	23
10.	2x2 Interlace Pattern	24
11.	16-Processor Interlace Pattern	26
12.	Interlace Machine	28
13.	Splitter Time Verses Number of Processors	31
14.	Interlace Time Verses Number of Processors	33
15.	Runway with One Shuttle	35
16.	Airport at Great Distance	36
17.	Airport at Great Distance	37
18.	Airport at Medium Distance	38
19.	Airport at Medium Distance	39
20.	Airport, Close to Runway	40
21.	Airport, Close to Runway	41
22.	Shuttle, Off Left Side of Tail	42

23. Shuttle, Off Left Side of Tail 43

24. Shuttle, Cargo Bay from Cabin 44

25. Shuttle, Cargo Bay from Cabin 45

26. Example of Screen Complexity 47

27. Complexity Example--Timing Summary 49

28. Timing Summaries 52

29. Timing Summaries--2 54

30. Timing Summaries--3 55

31. 16 Processor Time as Function of Viewer Position . . 60

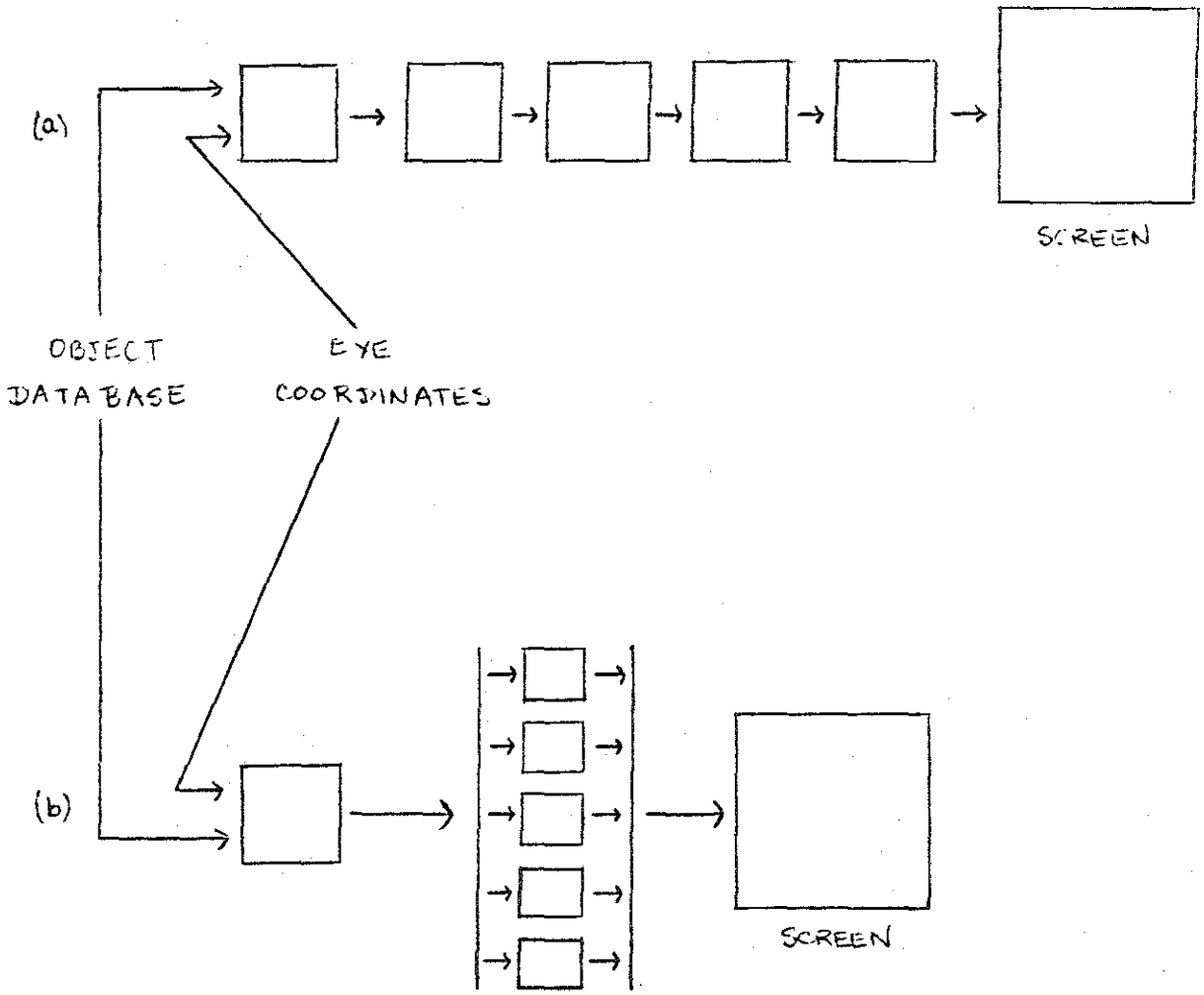
Chapter I

INTRODUCTION

1.1 MOTIVATION

The increasing popularity of computer graphics systems reflects the fact that the mind more easily grasps pictures than tables of numbers. However, one area of computer graphics which is not receiving much publicity (but is of great interest) is real-time three-dimensional modeling with hidden surface elimination (rather than wire frame images). If such a system is to be "real-time," that is to say that the display can be updated in less time than a human can perceive the changes, the system must generate the new scene in less than 1/15th of a second. This task is very expensive computationally, and cannot be done on most computers using realistic databases. Let us briefly consider its complexity. A database in such a system might be defined as a series of planar polygons (or tiles). A polygon, in turn, is a list of vertices in three-space. A "solid" object would simply be a collection of polygons (or surfaces). However, the polygons may be manipulated independently, without reference to a higher structure. The mathematical formulae for describing the perspective transformations properly (so that the display changes to convey the correct depth cues) are well-known, and can be handled easily (see [Newman79]).

The computationally expensive parts of this task occur after the polygons have been transformed. One must then decide how to manipulate the display to represent the set of transformed polygons. A typical screen might be a matrix of 512x512 picture elements (pixels). Then, for each of the 256,000 pixels, the computer system must determine which polygon is visible (the "hidden surfaces" cannot be seen); and, for that polygon the system must determine what color (or grey scale intensity) to display. This computation must be done for four pixels every microsecond, on the average, to generate the entire screen without flicker or uneven motion. The traditional method of dealing with this problem has been to build a large pipelined machine with many special purpose processors [Shohat77]. However, such machines are very expensive to build [Schac81].



TWO APPROACHES FOR 3-DIMENSIONAL GRAPHICS COMPUTERS
WITH HIDDEN SURFACE ELIMINATION AND SMOOTH SHADING

TRADITIONAL PIPELINED APPROACH (a)
AND
"PARALLEL MICRO" APPROACH OF FUCHS AND PARKE (b)

FIGURE 1

A slightly less ambitious task is to generate new scenes not in "real-time," but in "interactive-time;" that is to say, an observer would notice the change from one scene to the next, but the generation would require only a fraction of a second (say, 1/3 of a second), instead of several seconds. However, general purpose computers (e.g. a VAX 11/780) cannot generate scenes for even this requirement. Thus, a single dedicated processor is either too slow (general purpose systems), or too expensive (pipelined systems).

The advent of low cost microcomputers has made another approach possible; one could divide the display into several smaller areas and dedicate a microprocessor to each area. Each micro would then work on its own (small) area in parallel with the other micros. Thus, one avoids the high cost of many special purpose computers, but gets better performance than a uniprocessor.

Three architectures have been proposed following the "parallel micro" strategy--one by Fuchs [Fuchs77, Fuchs79] and two by Parke [Parke79a, Parke80]. Parke [Parke80] has analyzed the expected performance of these machines (see section 4.1) using the following assumptions:

1. the processors execute an algorithm similar to that described in [Suther74].

2. A uniform distribution of polygons over the screen.

However, most interesting data represent landscapes (e.g. airports or city skylines) or objects (e.g. ships, airplanes or molecules). For these types of data, assumption (2) is suspect. And since one of the schemes is especially sensitive to the distribution of polygons, an analysis based on realistic data may yield more accurate estimates of the processing speed of various designs. This thesis expands on Parke's results by comparing these architectures using realistic data. The project is described in further detail below.

1.2 DESCRIPTION OF PROJECT

One of the oldest and commonest methods used to compare computers is the technique of benchmarking. That is to say, several programs are executed on the target machines, and the time each machine takes to execute the set of programs is used as the machine's "score". A similar technique was used in this project.

In this project, the algorithm to be executed (which is discussed below) is fixed and already specified. What is

not specified is the data the machines must manipulate. Thus, we chose several views of two related databases (which are discussed below), and used these as our benchmark. The views (or "scenes") are exactly what an observer would see given that he was at a specified location (in x, y and z) looking with a given angle of view and direction. Thus, this analysis is very dependent upon the selected scenes being typical of scenes in general. However, the use of actual, generated scenes allows us to avoid making assumptions about the size of polygons, their shape, number of vertices, or their distribution over the screen, etc.

The concept of elapsed time was also a problem in this project, since physical implementations were not available. In place of seconds (or milliseconds), we have used memory cycles, since the most important single factor in the execution time of a simple instruction is the number of memory fetches required [Fuller77, p. 29]. Thus, execution time is given in terms of the number of memory fetches required (for both instructions and data) to execute the given algorithm for a given scene. Multiply and Divide instructions were assumed to require 10 memory cycles each. The PDP-11 was chosen as the base processor, in spite of the fact that it would never be used to build one of these machines (because of its limited addressing capability). However, it has influenced current 16-bit micro processors heavily, and its instruction set is very typical. And since execution times are expressed as memory cycles, they can be adapted for different speeds of processors and memories. Should a processor have a cache memory, however, the execution times would vary greatly from those calculated here; currently, few micro processors use a cache.

Given the scenes we wish to use as a benchmark, and the use of memory utilization as our timing metric, we could have simulated the generation of each scene by each of the machines we wished to compare, and actually counted the memory fetches required. However, this would have given little insight into <u>why</u> the machines behaved as they did. Therefore, another method was developed which produced results nearly identical to the strict simulation, and also aided in understanding the factors which caused the machine behavior. This method contains three steps. First, the algorithm used in the machines was analyzed, and the scene characteristics which affect execution time were identified (e.g. the number of polygons, the height of each polygon, the size of each polygon, etc). Then, a formula which describes how the algorithm depends on these characteristics was developed. The algorithm and analysis are presented in chapter 2. Second, five machines were chosen to include in the comparison, the uniprocessor case, 4- and 16-processor machines using the Fuchs scheme, and 4- and 16-processor machines using the Parke scheme. These machines (and their underlying ideas) are discussed in chapter 3.

Third, the selected scenes were generated and then processed by a simulator which extracted the statistics relevant to the algorithm characteristics. The actual comparison consists of applying these data to the algorithm analysis formulas. This is discussed in chapter 4. Chapter 5 summarizes our conclusions and gives recommendations for future designs.

One database used in this project was the NASA Space Shuttle. Thus, this analysis is very dependent on the Shuttle representing a "typical" object, as well as the scenes selected representing "typical" scenes. The Shuttle database contains about 450 polygons. The second database was a simple airport. It consists of two runways, and two shuttles sitting on one of the runways. This database contains about 900 polygons.

Chapter II

ALGORITHM DESCRIPTION AND ANALYSIS

The algorithm used by all of the processors in this project is the well-known Z-buffer algorithm. In this section, we describe this algorithm and analyze one possible implementation. The analysis calculates the number of memory fetches required to execute each major portion of the algorithm, and allows us to calculate the number of memory fetches required to display a given scene on the different machines.

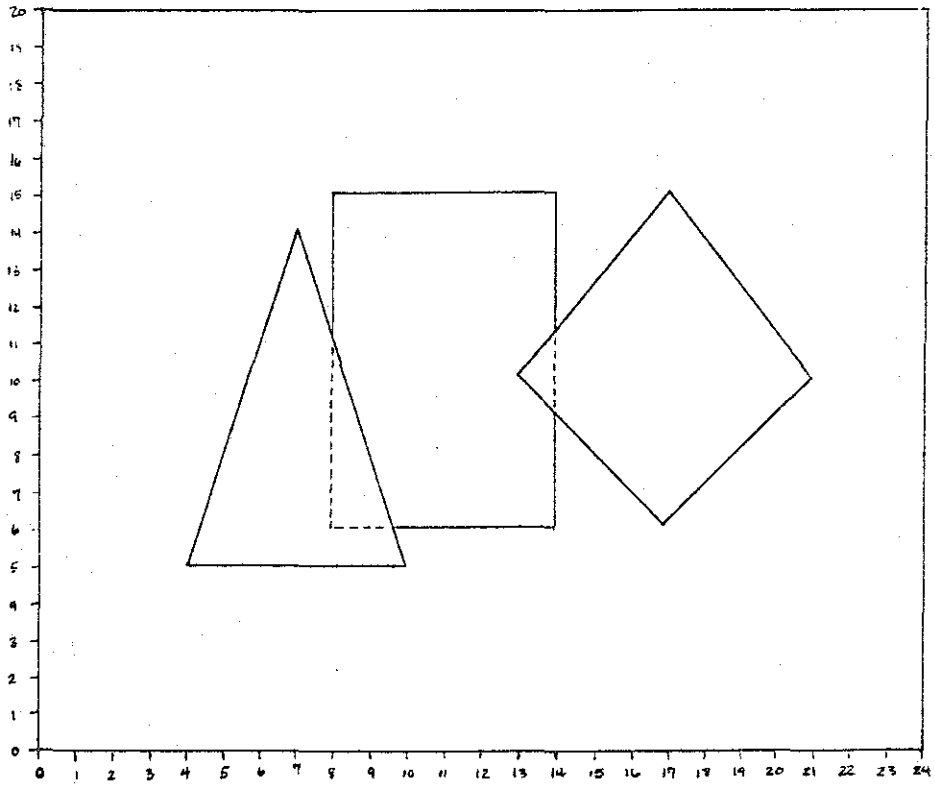
2.1 THE ALGORITHM

As mentioned earlier, the system must determine how to render the closest polygon at each pixel. The Z-buffer algorithm accomplishes this by keeping a buffer with the distance of the closest polygon at each pixel. This distance buffer (termed Z-buffer because it represents depth) can be thought of as being parallel to the frame buffer. Polygons are processed sequentially. First, the depth of each pixel covered by the new polygon is calculated. If the new polygon is closer to the observer than the value in this pixel's Z-buffer location, the new polygon depth is placed in the Z-buffer, and the new polygon's image is placed in the frame buffer. This is described in detail below:

Let a polygon be a collection of vertices and each vertex a 4-tuple of x, y, z and s, where x and y are the X and Y coordinates (respectively) of the point in the object space. Also, let z represent the distance between the point and the viewer. Finally, s is the shading or intensity value for the vertex. A polygon is defined by drawing lines from each vertex to the next, with the last vertex connected to the first. An example follows:

Vertices	poly A	poly B	poly C
1	4,5,7,10	8,15,8,9	21,10,6,11
2	10,5,7,10	8,6,8,9	17,15,6,11
3	7,14,7,10	14,6,8,9	13,10,6,11
4		14,15,8,9	17,6,6,11

(broken lines denote hidden edges)



POLYGON EXAMPLES
FIGURE 2

We will make the common assumption that all polygons are convex.[1] We will further reduce the amount of work the tiling algorithm must do by eliminating redundant vertices in polygon definitions. That is, no vertex is repeated in a polygon description; for example, the polygon ((20,30), (15,15), (15,15), (30,30)) has a redundant vertex at (15,15). We will, however, allow polygons of 1 or 2 vertices (i.e. a point or single line).

Another common assumption in three-dimensional graphic systems is that polygons are "one sided," and described consistently. This can be understood by considering a description of a flat, planar object, say a table. The table will have different polygons describing the top and bottom, because otherwise it would have no depth. And, if one is below the table, we know that only the bottom can be seen; the top cannot be seen because it is "facing" the wrong way. If one describes the "front" faces consistently (and we describe them in a counter-clockwise manner), then the backfacing polygons can be easily identified and removed just before scene generation (see [Newman79]). Thus, backfacing polygons represent another form of useless data which can be easily identified and removed, and so we assume that they will not be given to the algorithm. Both C and PDP-11 assembler listing for this tiling algorithm are in Appendices A and B.

The algorithm operates on one polygon at a time, and a polygon is represented in the algorithm in a tabular form. Consider:

```
                          x   y   z   s
                        |---|---|---|---|
vertex 0------>         |   |   |   |   |
                        |---|---|---|---|
vertex 1------>         |   |   |   |   |
                        |---|---|---|---|
                          .   .   .   .
                          .   .   .   .
                          .   .   .   .
vertex n-1 ---->        |   |   |   |   |
                        |---|---|---|---|
```

Figure 3: Tabular Polygon Representation

[1] Assuming convex polygons often simplifies graphics algorithms, and non-convex polygons can always be divided into several convex ones. See [Newman79] for more information on this topic.

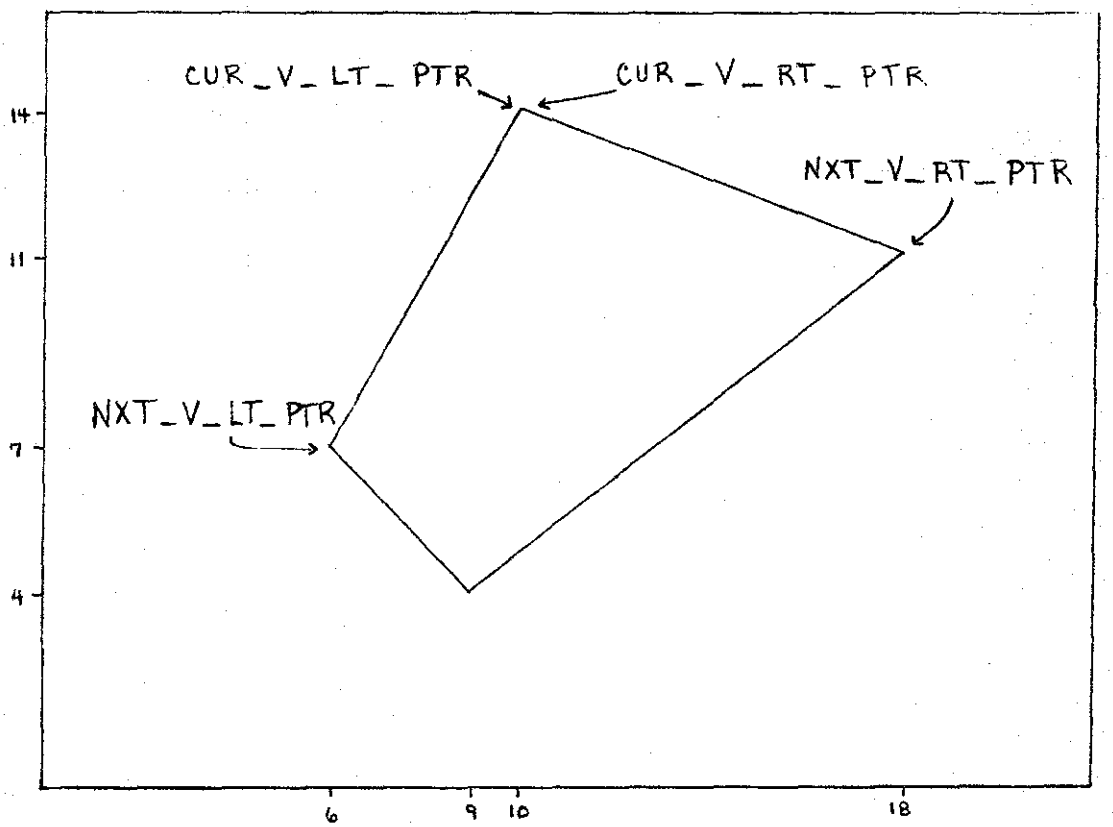
The algorithm follows:

1. Scan all vertices, finding the one with the highest y value (i.e. the topmost vertex) for the left and right sides. This becomes the current left (or right) vertex.
   Set

CUR_V_RT_PTR	pointer to current vertex, right side
CUR_V_LT_PTR	pointer to current vertex, left side
MIN_Y	minimum (i.e. lowest) y value

2. Initialize

NXT_Y_LT	next y value, left side
NXT_Y_RT	next y value, right side
NXT_V_RT_PTR	pointer to next vertex on right side
NXT_V_LT_PTR	pointer to next vertex on left side
CURRENT_Y	current y value (i.e. current scan line)
Z_ROW_PTR	pointer to current row in z buffer
IMAGE_ROW_PTR	pointer to current row in image buffer (see figure 4)



MIN_Y = 4 CURRENT_Y = 14

NXT_Y_LT = 7 NXT_Y_RT = 11

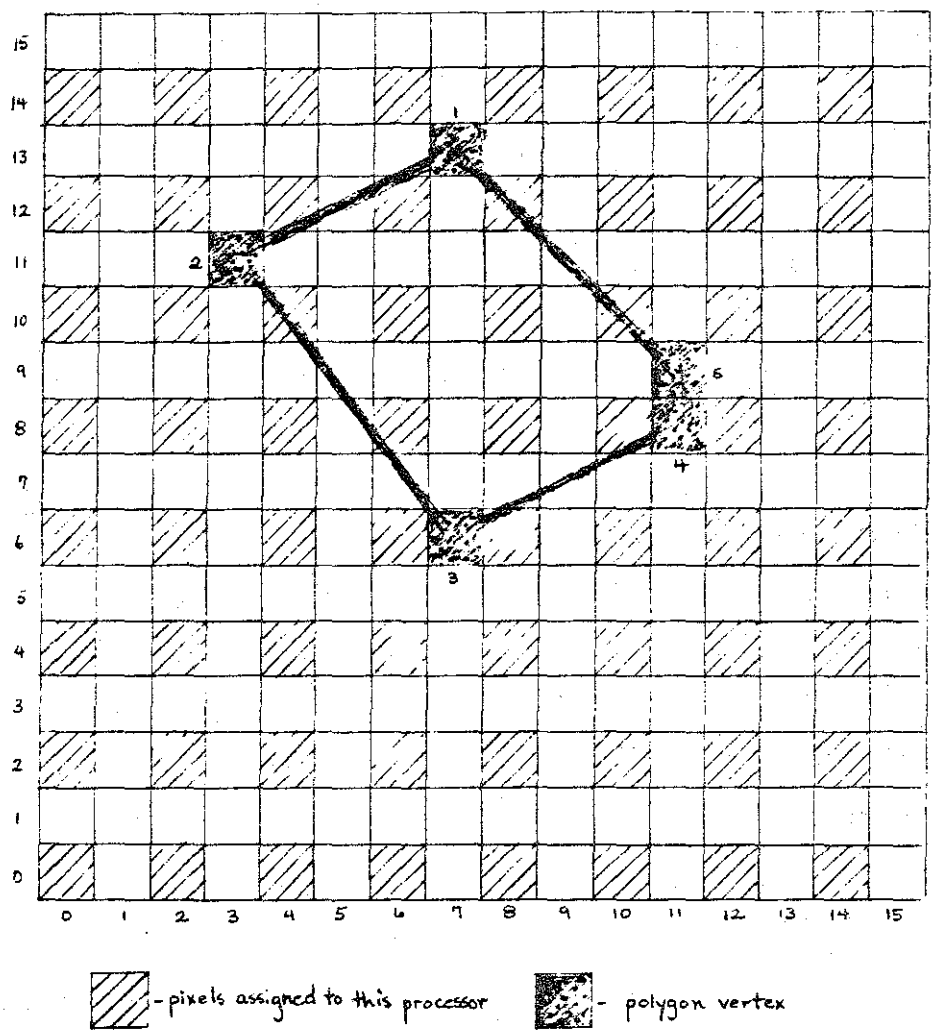
VALUES AFTER INITIALIZATION

FIGURE 4

```
3.  for each Scanline use (CURRENT_Y) to go from
    (HIGHEST_Y_VALUE) down to (MIN_Y) do

4.  if (CURRENT_Y <= NXT_Y_LT) calculate new values for
       CUR_X_LT        current x value, left side
       CUR_Z_LT           "    z   "     "    "
       CUR_S_LT           "    s   "     "    "
       NXT_Y_LT        next y value, left side
       DX_LT           delta value (i.e. increment) for
                       x left side
       DZ_LT           delta value for z, left side
       DS_LT             "     "    "  s,  "    "
       NXT_V_LT_PTR    next vertex pointer, left side
    fi

5.  if (CURRENT_Y <= NXT_Y_RT)
    calculate
       CUR_X_RT
       CUR_Z_RT
       CUR_S_RT
       NXT_Y_RT
       DX_RT
       DZ_ RT
       DS_RT
       NXT_V_RT_PTR
    fi

6.  find the y value of the next highest vertex--i.e.
    the next y value where vertex processing must be
    done.
    Set  NXT_HIGH_Y = max(NXT_Y_LT, NXT_Y_RT)

7.  for each scanline use (CURRENT_Y) to go from
    (CURRENT_Y) down to (NXT_HIGH_Y) do
    Calculate
       IMAGE_PTR = IMAGE_ROW_PTR[CUR_X_LT] current pixel
                                           in image buffer
       Z_LT_PTR  = Z_ROW_PTR[CUR_X_LT]     current pixel
                                           in z buffer
       Z_RT_PTR  = Z_ROW_PTR[CUR_X_RT]     last pixel in
                                           z buffer
       PIX_DZ                              delta for z
       PIX_DS                              delta for s
       PIX_Z                               z value
       PIX_S                               s value

8.  for each pixel use (PIX_Z_VAL) to go from (Z_LT_PTR)
    over to (Z_RT_PTR)
    if PIX_Z < valueof(PIX_Z_VAL)
       valueof(PIX_Z_VAL) = PIX_Z
       valueof(IMAGE_PTR) = PIX_S
    fi
    increment
       PIX_X by PIX_DZ
```

```
        PIX_Z by PIX_DZ
        PIX_S by PIX_DS

    end of stmt 8 'for' loop

9.  increment
        CUR_X_LT by DX_LT
        CUR_X_RT by DX_RT
        CUR_S_Lt by DS_LT
        CUR_S_RT by DS_RT
        CUR_Z_LT by DZ_LT
        CUR_Z_RT by DZ_RT
    move Z_ROW_PTR to next row of z buffer
    move IMAGE_ROW_PTR to next row of frame buffer

    end of stmt 7 'for' loop

    end of stmt 3 'for' loop
    end of algorithm
```

In this project, the X and Y step sizes (the increment to go from one Y value to the next, or one X value to the next) are not unity, but are compile-time constants. Thus, this program transforms a very fine image space into a coarse screen space by point (not area) sampling. While we will not list the (conceptually) minor changes in the algorithm required to make this change here, we will list some of the less obvious problems it raises. Consider figure 5:



POLYGON FALLING ON 4-PROCESSOR INTERLACE SCHEME GRID

FIGURE 5

1. the top line (the highest y value to be processed) is not simply the y value of the highest vertex (y=13), but the y value of the highest assigned line under the highest vertex (y=12).

2. the processing at vertex 2 (point (3,11)) must correct the values of the x, z and s (and their delta values) from those given at the vertex. Further, since movement in x (as well as y) is required, the calculations depend on the edge value from the right--which may not be known when vertex 2 is being processed.

3. on a given side, more than one vertex may require processing in moving from one y value to the next. For example, in moving from scanline (or y value) 10 to 8, the right side must process vertices 5 and 4.

4. the single pixel appearing on line 6 (at (7,6)) is not assigned to this processor, and, therefore, this processor has no processing to do on line 6.

2.2 ANALYSIS OF VISIBLE SURFACE ALGORITHM

This analysis is based on that of [Parke80]. In this paper, "timing" refers to the number of memory cycles used for both instructions and data on a PDP-11. Memory cycles were used because the most important factor in simple instruction execution time on current computers is the number of memory cycles required. Multiply and divide instructions were assumed to take 10 memory cycles. This implementation of the algorithm uses 16 bits of precision.

1. Scan all vertices, finding the one with the highest y value (i.e. the topmost vertex) for the left and right sides.

   Timing analysis: 42 memory accesses per polygon.
   24.6 memory accesses per vertex.

2. Initializations

   Timing analysis: 97 memory accesses per polygon.

3. for each Scanline use (CURRENT_Y) to go from (HIGHEST_Y_VALUE) down to (MIN_Y) do

   Timing analysis: 25 memory accesses per vertex.

```
4.  if (CURRENT_Y <= NXT_Y_LT) calculate new values for
        .
        .
        .
    fi

5.  if (CURRENT_Y <= NXT_Y_RT)
    calculate
        .
        .
        .
    fi
```

Timing analysis, left hand side and right hand side average: 257.2 memory accesses per vertex.

```
6.  find the y value of the next highest vertex--i.e.
    the next y value where vertex processing must be
    done.
```

Timing analysis: 12.5 memory accesses per vertex.

```
7.  for each scanline use (CURRENT_Y) to go from
    (CURRENT_Y) down to (NXT_HIGH_Y) do
```

Timing analysis: 85 memory accesses per scan line.

```
8.  for each pixel use (PIX_Z_VAL) to go from (Z_LT_PTR)
    over to (Z_RT_PTR)
    if PIX_Z < valueof(PIX_Z_VAL)
       valueof(PIX_Z_VAL) = PIX_Z
       valueof(IMAGE_PTR) = PIX_S
    fi
    increment
       PIX_X
       PIX_Z
       PIX_S

    end of stmt 8 'for' loop
```

Timing analysis: 15 memory accesses per scan line.
21 memory accesses per pixel.

```
9.  increment
       CUR_X_LT
       CUR_X_RT
       CUR_S_Lt
       CUR_S_RT
       CUR_Z_LT
```

```
   CUR_Z_RT move  Z_ROW_PTR to next  row of  z buffer
move IMAGE_ROW_PTR to next row of frame buffer
```

Timing analysis: 58 memory accesses per scan line.

```
end of stmt 7 'for' loop

end of stmt 3 'for' loop
end of algorithm
```

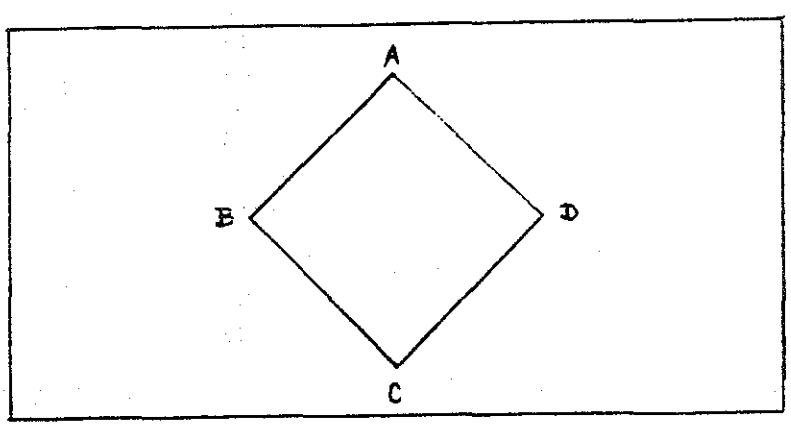
Execution time summary:

	abbreviation	memory cycles (avg.)
Polygon setup time	Gt	139
Vertex processing time	Vt	319.3
Segment processing time	St	158
Pixel processing time	Pt	21

```
Total scene processing time
   =   number of polygons * Gt
     + number of vertices * Vt
     + number of segments * St
     + number of pixels   * Pt

   = number of polygons
     * (Gt
        + average number of vertices per polygon * Vt
        + average number of segments per polygon * St
        + average number of pixels per polygon * Pt)
```

2.3 LIMITATIONS OF THIS ANALYSIS

If one considers a diamond shaped quadrilateral,



vertices A and C will be processed as BOTH left and right side vertices. Thus, the total number of vertices processed will be six, instead of four. However, when processing vertex C, neither side will use the calculated delta values. Therefore, the delta processing may be skipped for vertex C, and vertex processing now requires delta calculations for four vertices and a very small amount of processing for two vertices. Our analysis has simplified this situation to the processing of four vertices (with delta calculations), and no testing to avoid the unnecessary delta value calculations. In the results that follow, the error introduced by this simplification was less than five per cent in all scenes for the uniprocessor machine.

Chapter III

MACHINE DESCRIPTION

Obviously, one could program any general purpose computer to execute the algorithm given in Chapter 2. What is not obvious is how to distribute the work load among several processors. Both the Parke and Fuchs schemes divide the display (or screen) into disjoint areas, and then dedicate a processor to each area. The schemes differ in how the screen is divided. The Parke scheme is the simpler of the two, and will be discussed first. Much of this chapter is a condensation of material contained in [Fuchs77, Fuchs79 and Parke80] (for the Fuchs machine), and [Parke79a, Parke80] (for the Parke machine).

3.1 THE PARKE SPLITTER MACHINE

Given a certain number of microprocessors (say, 4) to execute a tiling algorithm, how might one connnect them to take advantage of parallel computation?

A simple method would be to divide the screen (image space) into contiguous blocks. Thus, we might have the following division schemes.

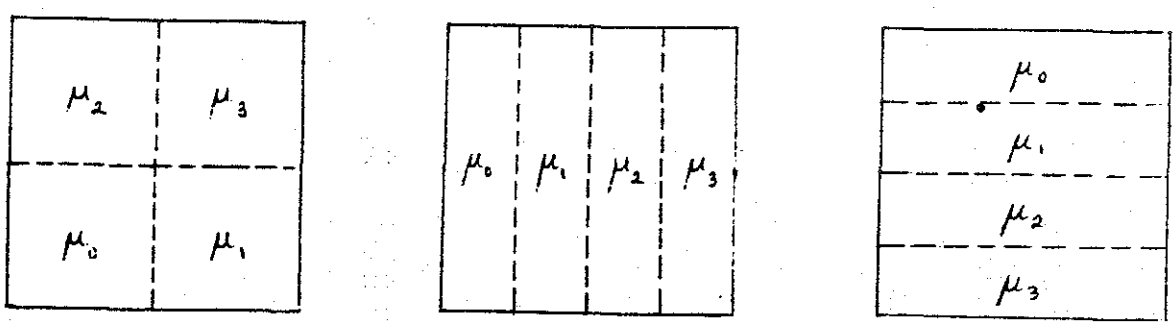


Figure 6: 3 Simple Division Schemes

Everything which falls in quadrant u0 is processed by the first microprocessor; everything which falls in quadrant u1 is processed by microprocessor 2, etc. The divisions of the image space may be vertical (b), horizontal (c) or a combination of vertical and horizontal (a). Henceforth, we will assume scheme (a) for the 4-micro Parke machine. One possible scheme (and the only one we will be considering) for a 16-processor Parke machine is in figure 7.

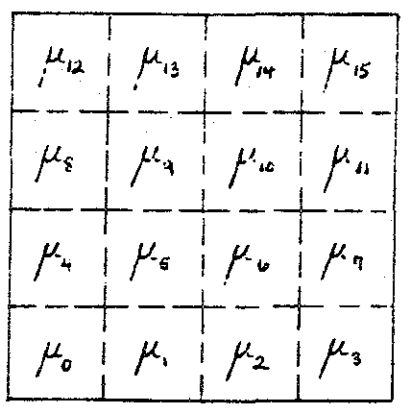


Figure 7: 16-Processor Parke Splitter Machine

The major problem in this scheme is insuring that a polygon does not cross a processor boundary. This is accomplished by a tree structure of hardware splitters which take a polygon description and output two polygons, one wholly to the left of the dividing line (or above, if the split is horizontal), and the other polygon wholly on the right (or below).

The Parke machine, then, consists of a central computer (which will perform all transformations on the polygons), a series of hardware splitters (in a tree structure), and a set of microprocessors (at the leaves of the splitter tree). The micros are then connected to a portion of a frame buffer which corresponds to that micro's portion of the screen. An illustration of a 4-processor machine is shown in figure 8.

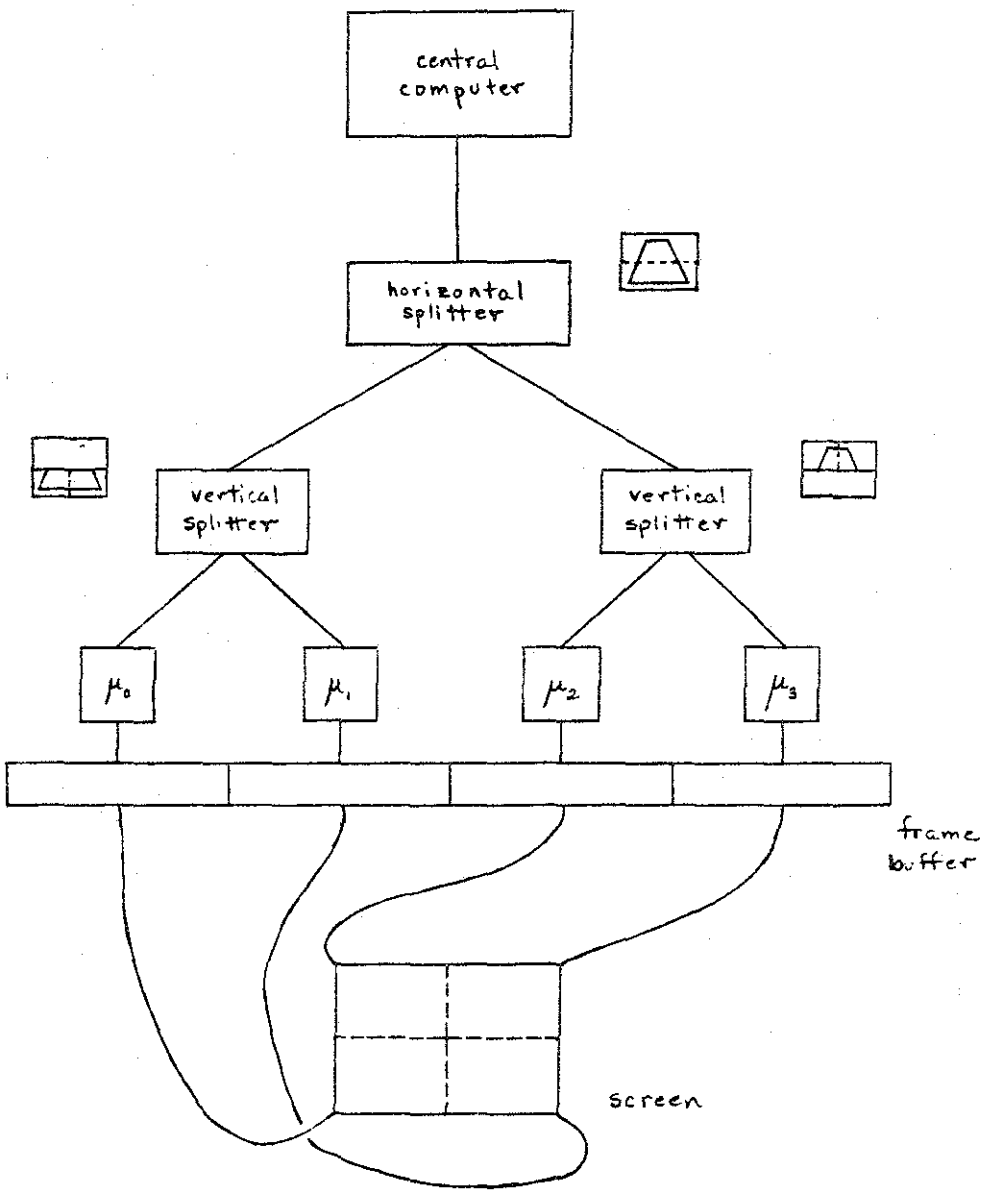


DIAGRAM OF PARKE SPLITTER MACHINE WITH FOUR PROCESSORS

FIGURE 8

3.2 THE FUCHS INTERLACE MACHINE

Instead of splitting the image space into contiguous blocks, we might divide on a pixel by pixel basis. Thus, given 4 processors, we might have a screen divided as in figure 9.

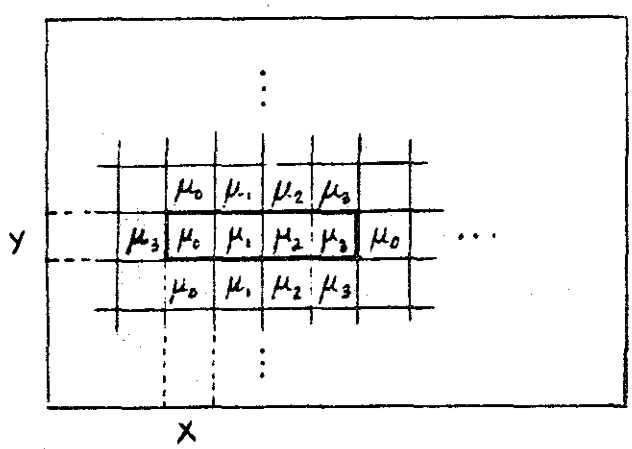


Figure 9: 1x4 Interlace Pattern

Here, pixel (x,y) is assigned to processor 1, pixel (x+1,y) to processor 2, (x+2,y) to number 3 and (x+3,y) to number 4.

Another scheme is given in figure 10.

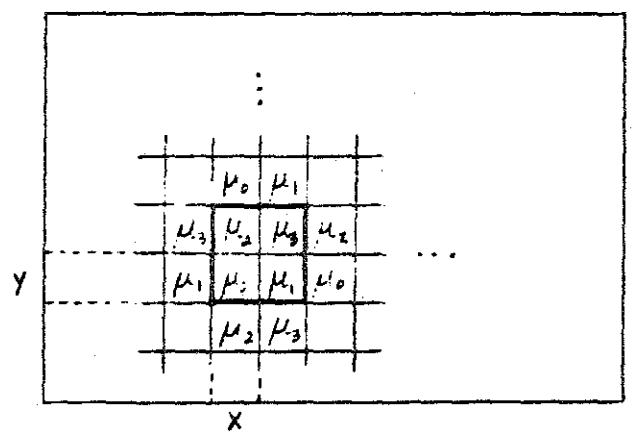
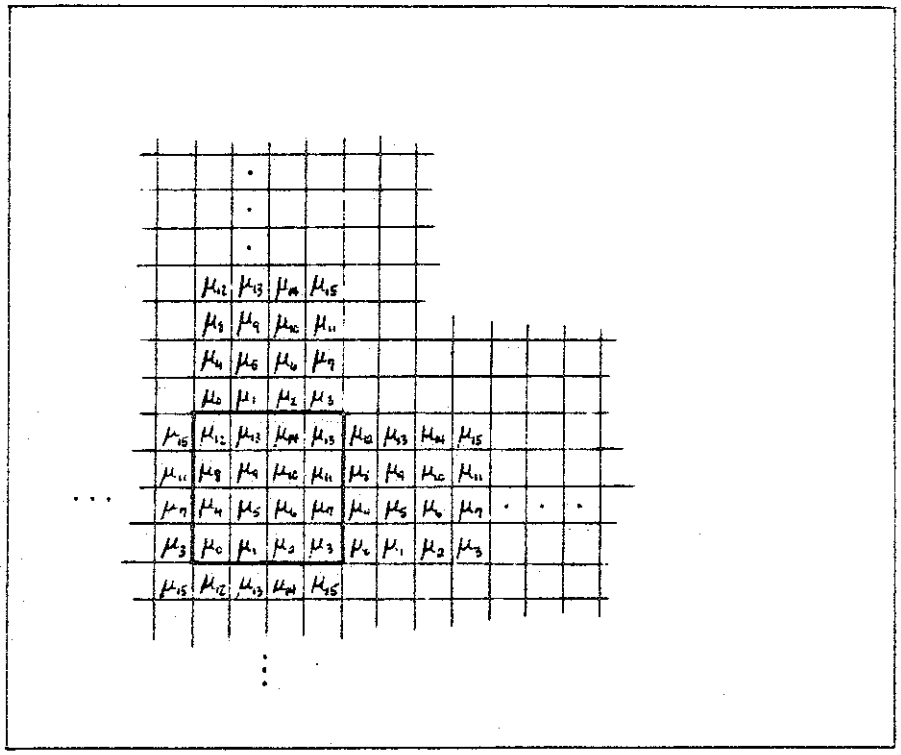


Figure 10: 2x2 Interlace Pattern

The pixel to processor assignments here are as follows: (x,y) to u0, (x+1,y) to u1, (x,y+1) to u2 and (x+1,y+1) to u3.
The above division is the one we will use for the 4-processor interlace[2] machine. The following figure shows the scheme we will use for the 16-processor machine.

---

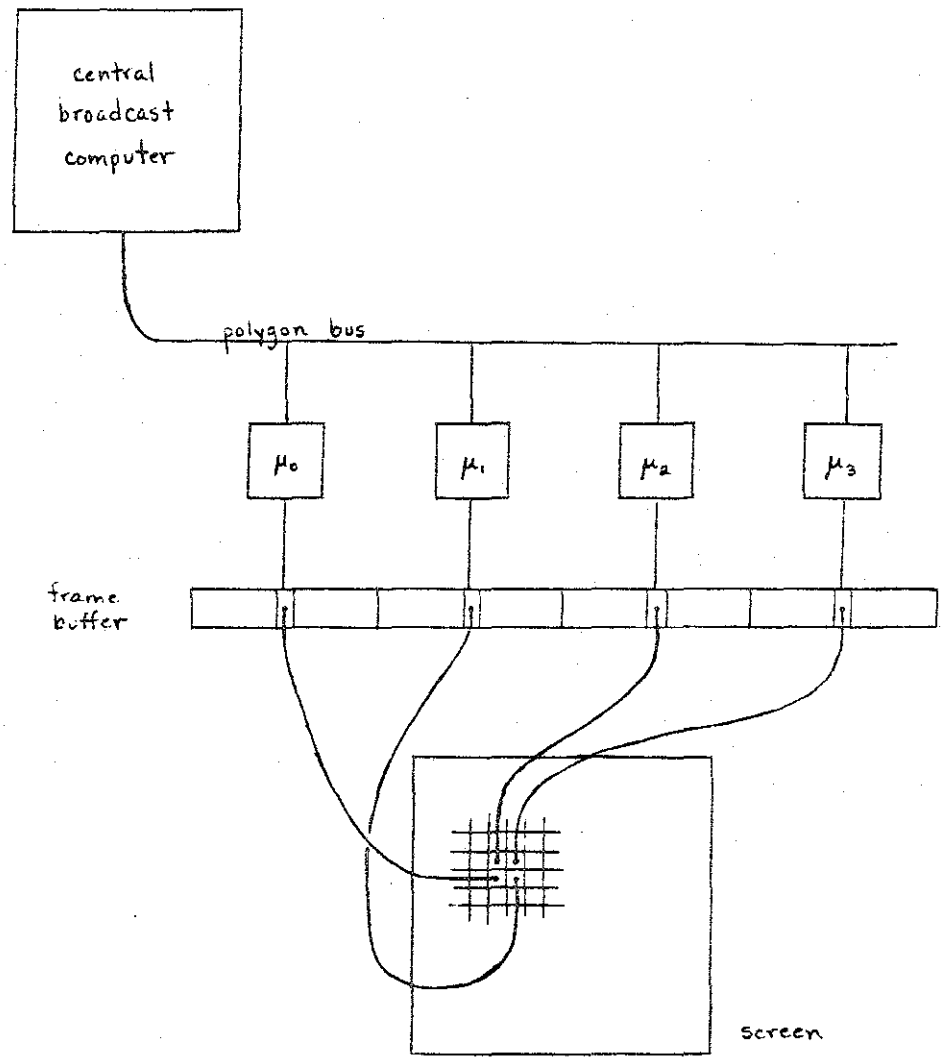
[2] We will use this designation for Fuchs' machine, to keep from confusing it with his other designs.



SIXTEEN- PROCESSOR INTERLACE PATTERN

FIGURE 11

An interlace machine, then, consists of a central computer (which plays the same role as in the splitter machine), a polygon bus, the collection of micros and a frame buffer bus. The micros are connected to both the polygon bus and the frame buffer bus. Each micro is connected to the frame buffer bus so that it has control of only the pixels assigned to it. This is shown in figure 12.



FOUR-PROCESSOR INTERLACE MACHINE

FIGURE 12

3.3 ASSUMPTIONS AND LIMITATIONS

The act of displaying a polygon is by far the most expensive computation performed by either a splitter or an interlace system. Thus, other parts of a splitter or interlace machine need not be considered in this analysis. For example, in processing a scene the splitter architecture may split each polygon several times. The time required to split a polygon is much less than the time required to display it, and so splitting time is not included in the performance analysis of the splitter architecture. However, increased hardware encoding of the visible surface algorithm could change the overall system balance, and invalidate this assumption.

Similarly, the geometric and perspective transformations performed by the central computer in each scheme are simple compared to the tiling process, and will not be a system bottleneck. Transmission time from the central computer down the bus (or through the splitter tree) is also ignored. In other words, each micro always has more polygon data available. However, in the splitter scheme, this assumption might not hold. In figure 20, for example, if the leftmost shuttle is described in a contiguous block of polygons, the tree could become saturated waiting for the (few) affected micros to process this part of the data. And, the micros which are to process the rightmost shuttle description would be standing idle, even though they have work to do. This phenomenon could increase the time required to display some scenes.

The frame buffer, also, is not included in this analysis, since the transfer time to it from a micro is far overshadowed by the processing time.

Chapter IV

SIMULATION RESULTS AND MACHINE COMPARISON

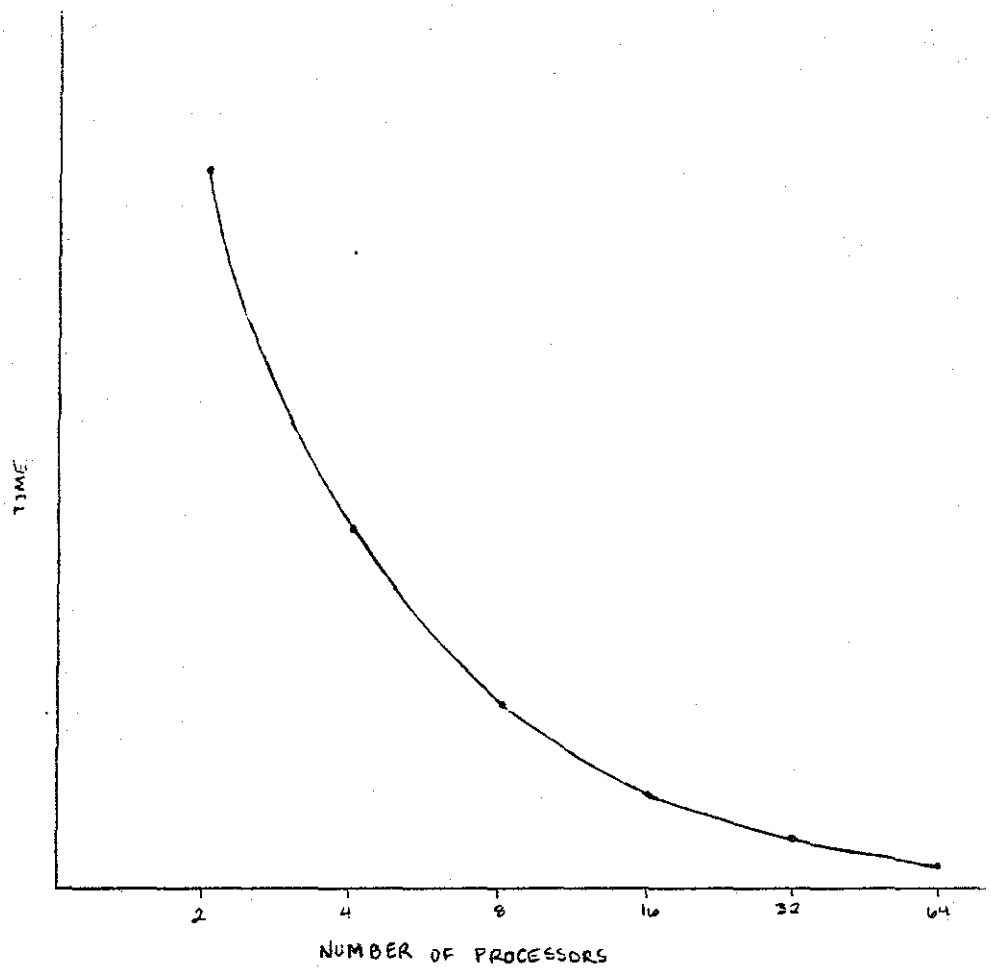
At this point we are able to compare the two designs. To minimize bias, we will be generating a variety of scenes using data supplied from the Evans and Sutherland real-time system at the Johnson Space Center. Our method will be to take each scene and simulate each processor in each machine by counting the number of pixels, line segments, edges and polygons processed. From this, we can use the timing formula derived from the algorithm analysis to calculate the total time required. The results of this process are presented in this chapter for 1-, 4- and 16-processor interlace and splitter machines. Below, we give the scenes[3] which were analyzed and their results. We then analyze the results. First, however, we will review Parke's results.

4.1 PARKE'S COMPARISON

In his comparison [Parke80], Parke assumes that the polygons to be displayed are evenly distributed over the screen. Thus, for an nxn splitter machine, 1/n**2 of the total scene would fall in each section of the screen, and each processor would do about 1/n**2 of the total work of a uniprocessor working on the same scene. That is, each processor would be responsible for (approximately) 1/n**2 of the polygons, and thus have 1/n**2 of the total vertices, 1/n**2 of the total number of segments, etc. Parke, therefore, claims that for a fixed scene, processing time and number of processors are related by graphs with the general shape of figure 13. (Basically, doubling the number of processors halves the execution time of a given scene.)

---

[3] Space Shuttle data courtesy of NASA.

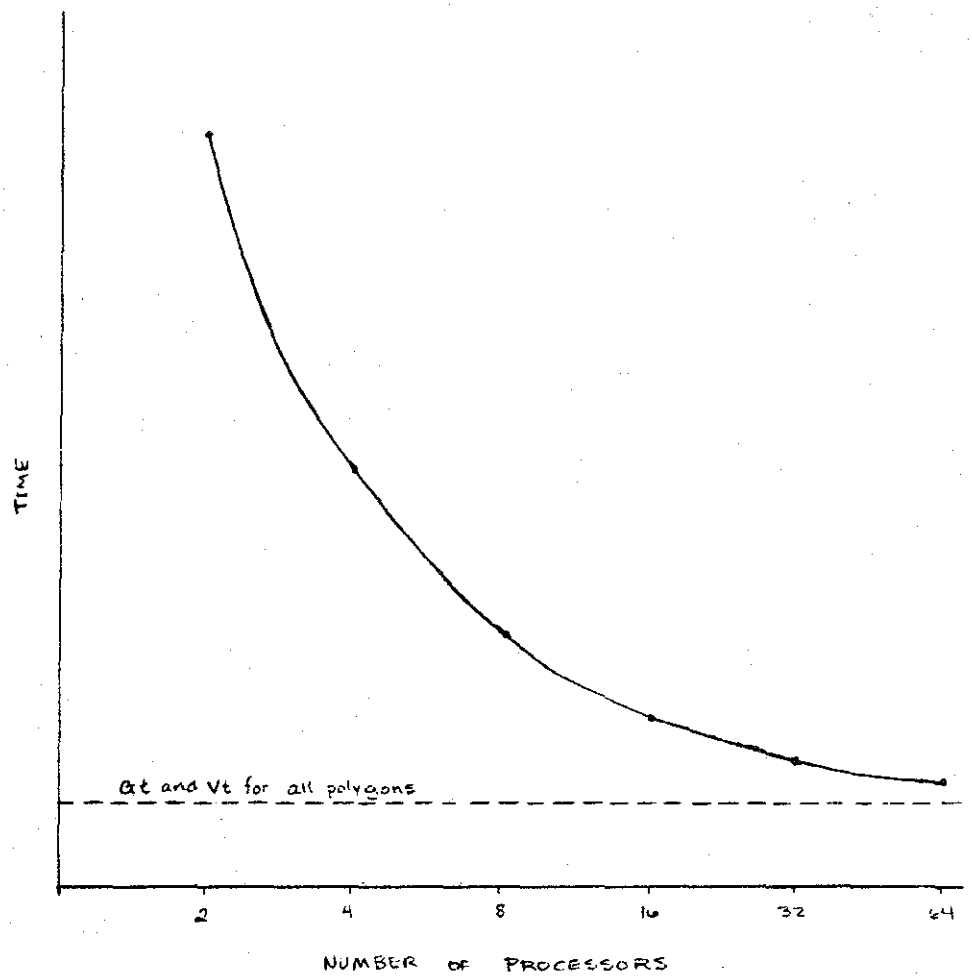


SPLITTER MACHINE TIME

execution time for a given scene as a function of number of processors

FIGURE 13

For the interlace machine, however, the timing curve does not approach zero, but instead approaches some constant which is the time required for a processor to process the polygon time (Gt) and vertex time (Vt) for each polygon in the scene. Thus, execution time graphs for interlace machines are generally shaped like figure 14.



INTERLACE MACHINE TIME

execution time for a given scene as a function of number of processors

FIGURE 14

Of course, one can construct pathological scenes for which adding processors does not significantly reduce processing time for either scheme (or both schemes). And one could also note that the splitter architecture's execution time does not actually approach zero, but instead approaches a constant which depends on the scene's highest depth complexity (namely,

$$(Gt + Vt + St + Pt) * \max(Dc).$$

However, the real question raised by Parke's work is the relevance of these graphs to machines working on real data. We now investigate this question.

4.2 THE ANALYZED SCENES AND THEIR RESULTS

We will now present the analyzed scenes, and their timing results in millions of memory cycles. We also include statistics on a third architecture, the "hybrid," which we will discuss later.

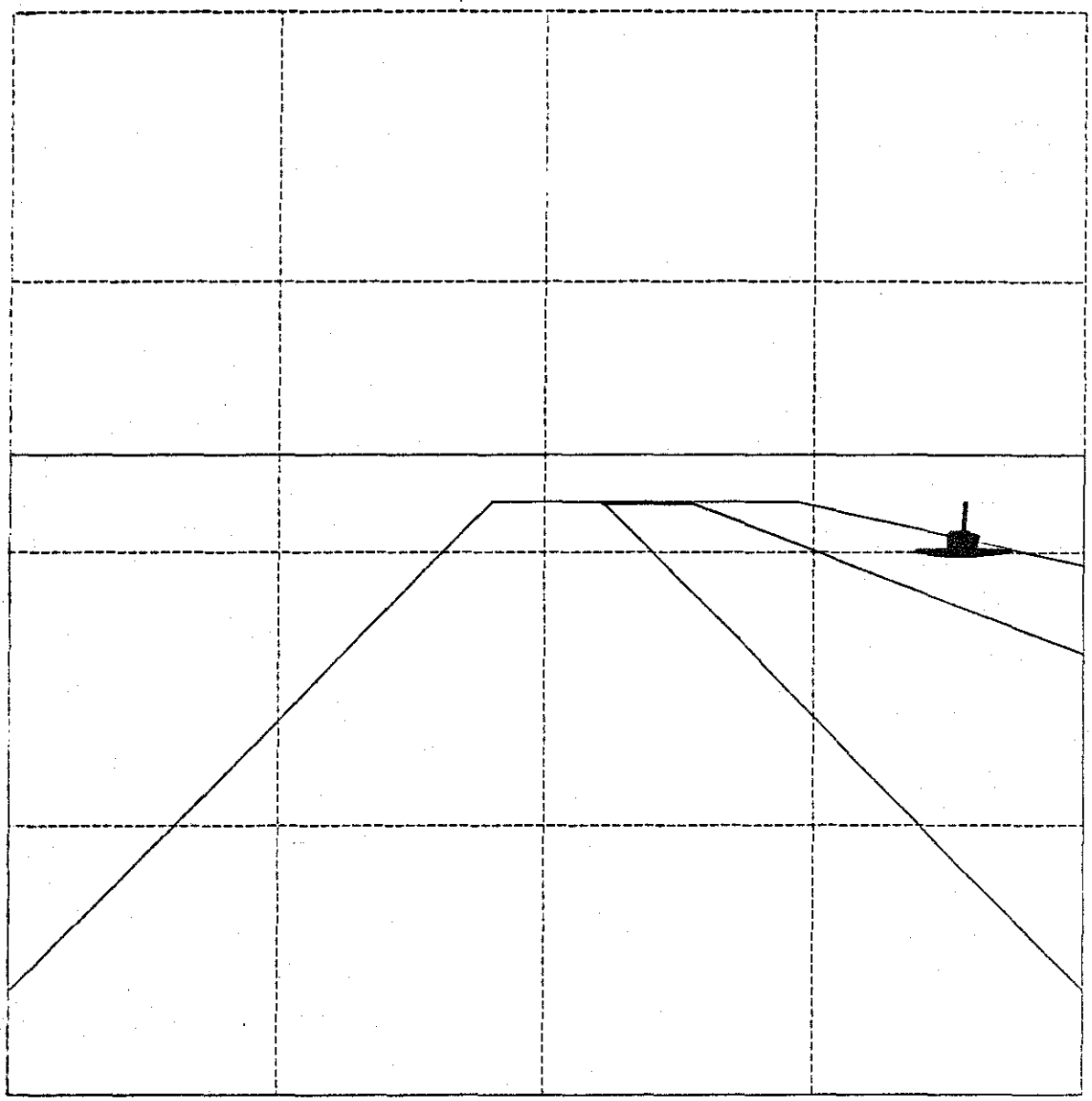
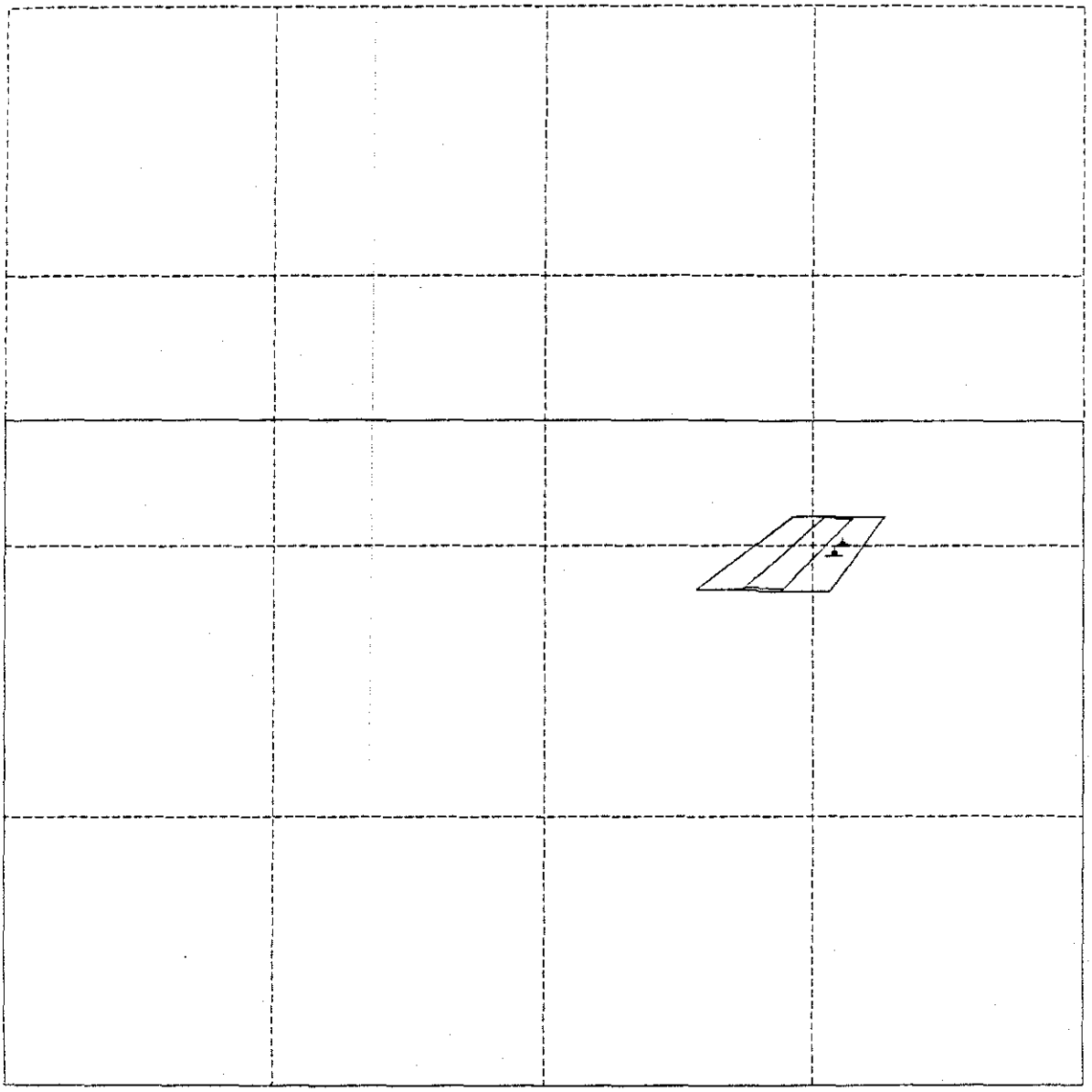


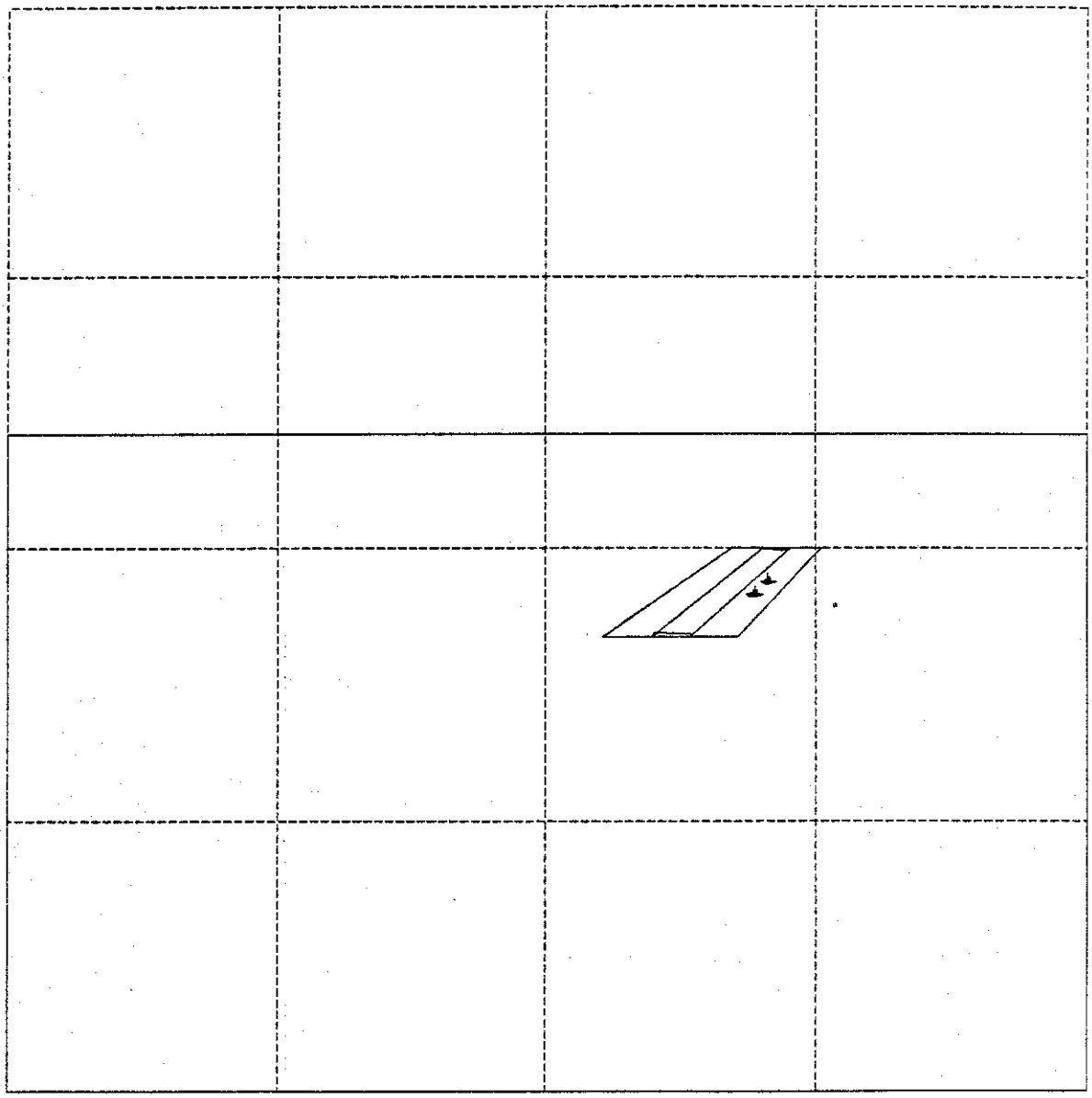
Figure 15: Runway with One Shuttle



MILLIONS OF MEMORY CYCLES REQUIRED TO DISPLAY SCENE

	uniprocessor	4-processor	16-processor
uniprocessor	3.95		
interlace		1.32	.653
splitter		1.69	.592
hybrid			.596

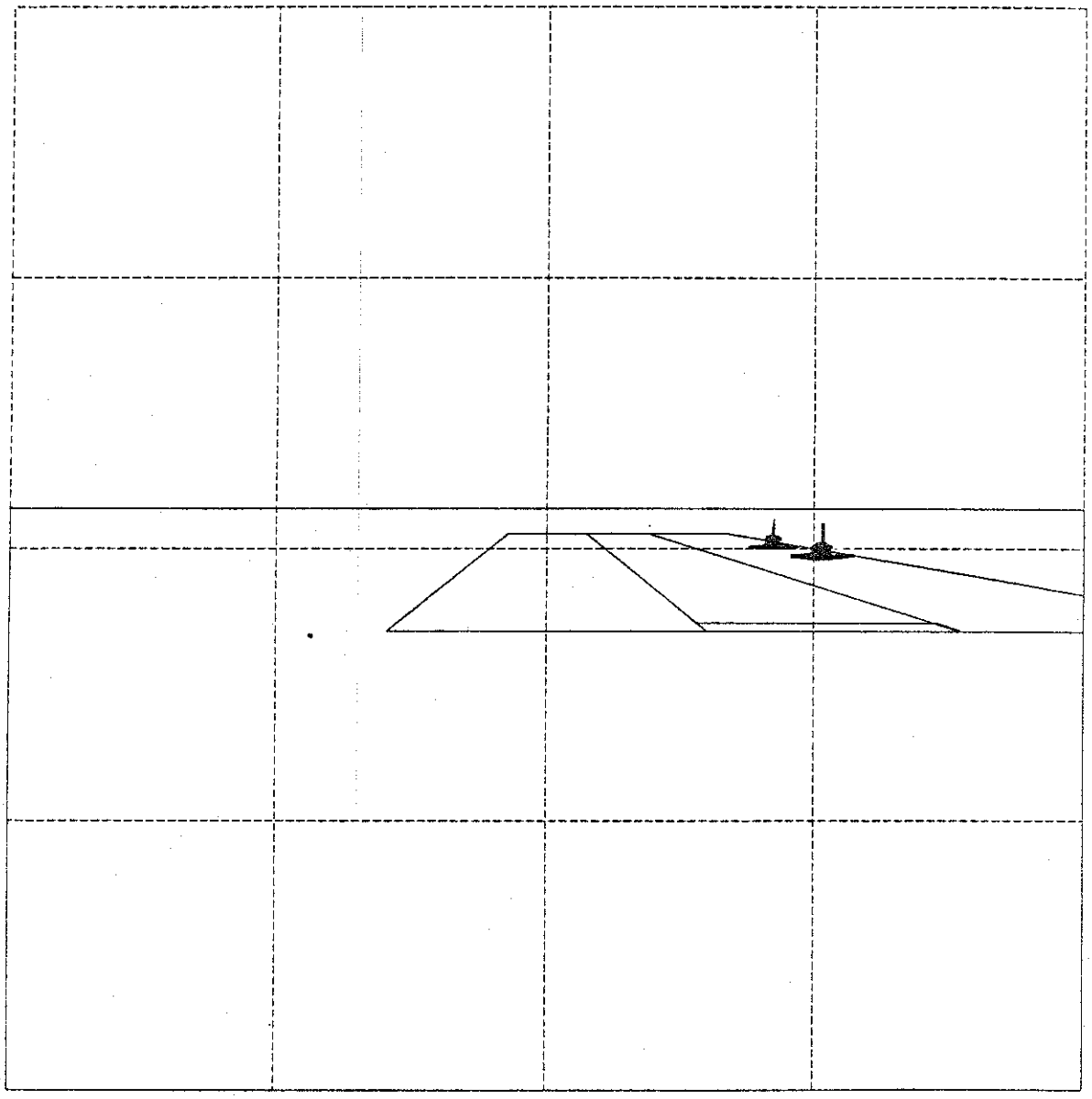
FIGURE 16 (Screen contains 488 polygons.)



MILLIONS OF MEMORY CYCLES REQUIRED TO DISPLAY SCENE

	uniprocessor	4-processor	16-processor
uniprocessor	3.89		
interlace		1.30	.644
splitter		1.80	.749
hybrid			.681

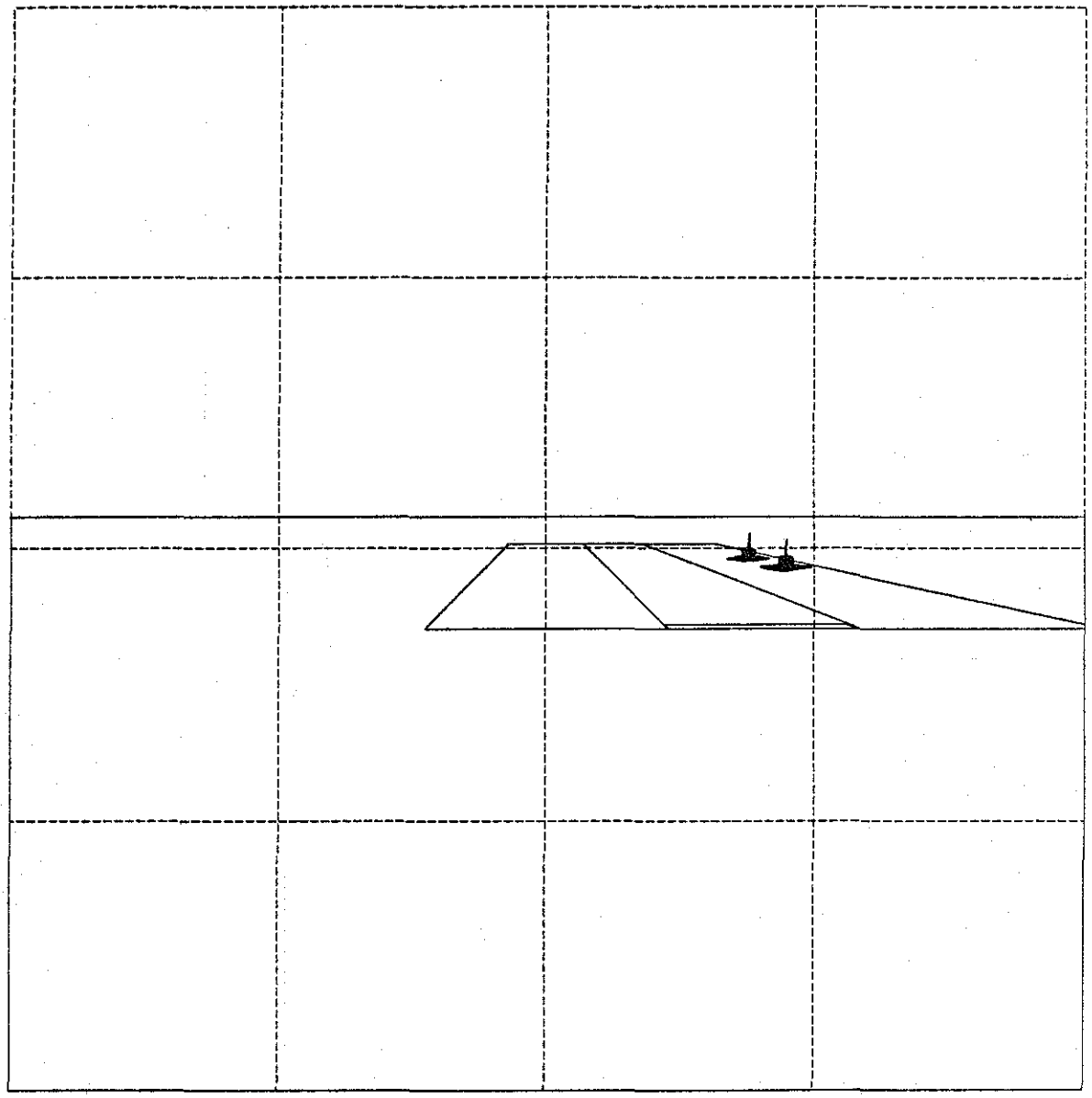
FIGURE 17 (Screen contains 489 polygons.)



MILLIONS OF MEMORY CYCLES REQUIRED TO DISPLAY SCENE

	uniprocessor	4-processor	16-processor
uniprocessor	3.91		
interlace		1.42	.791
splitter		1.85	.629
hybrid			.658

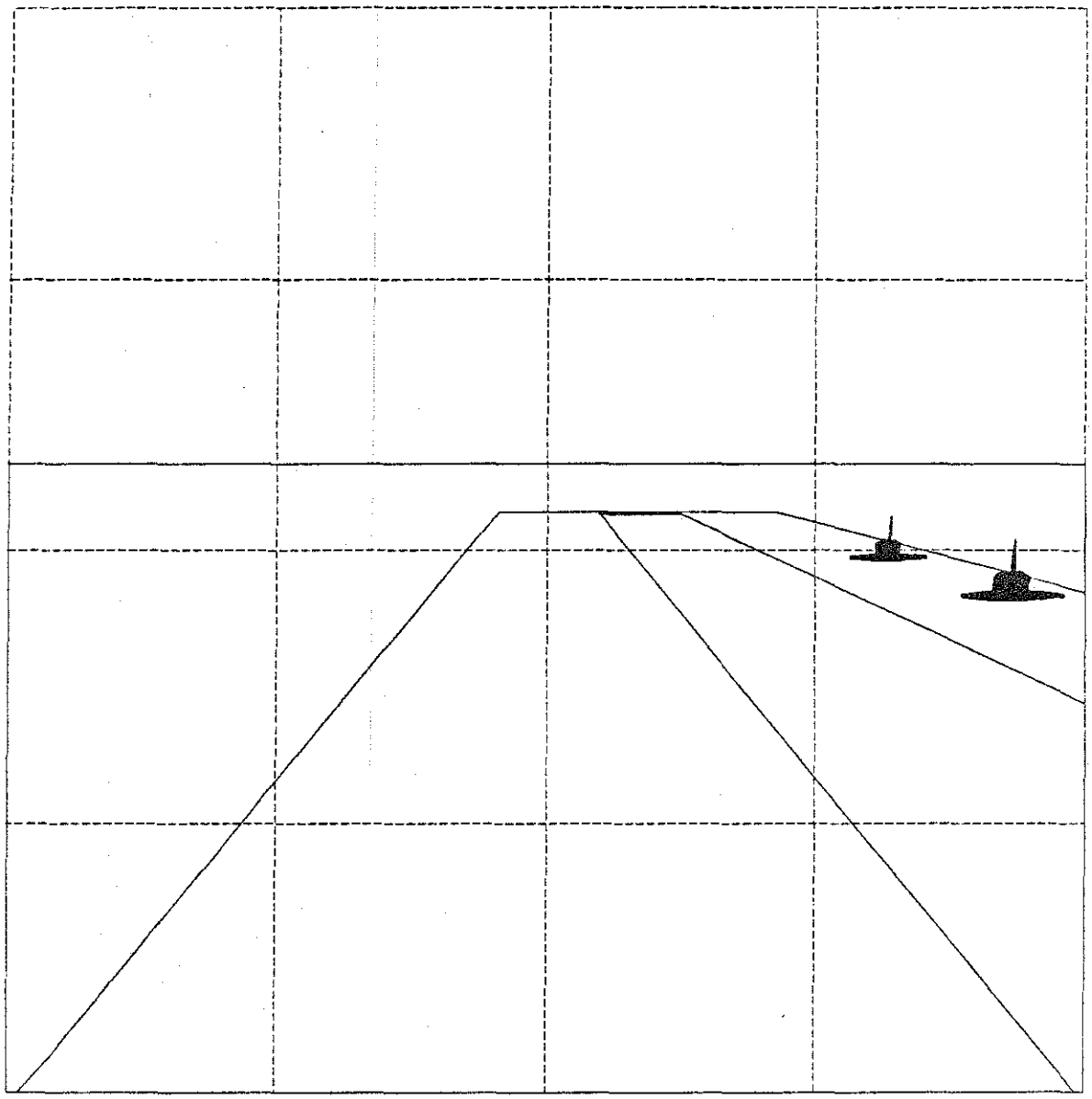
FIGURE 18 (Screen contains 499 polygons.)



MILLIONS OF MEMORY CYCLES REQUIRED TO DISPLAY SCENE

	uniprocessor	4-processor	16-processor
uniprocessor	3.78		
interlace		1.37	.761
splitter		2.16	1.06
hybrid			.941

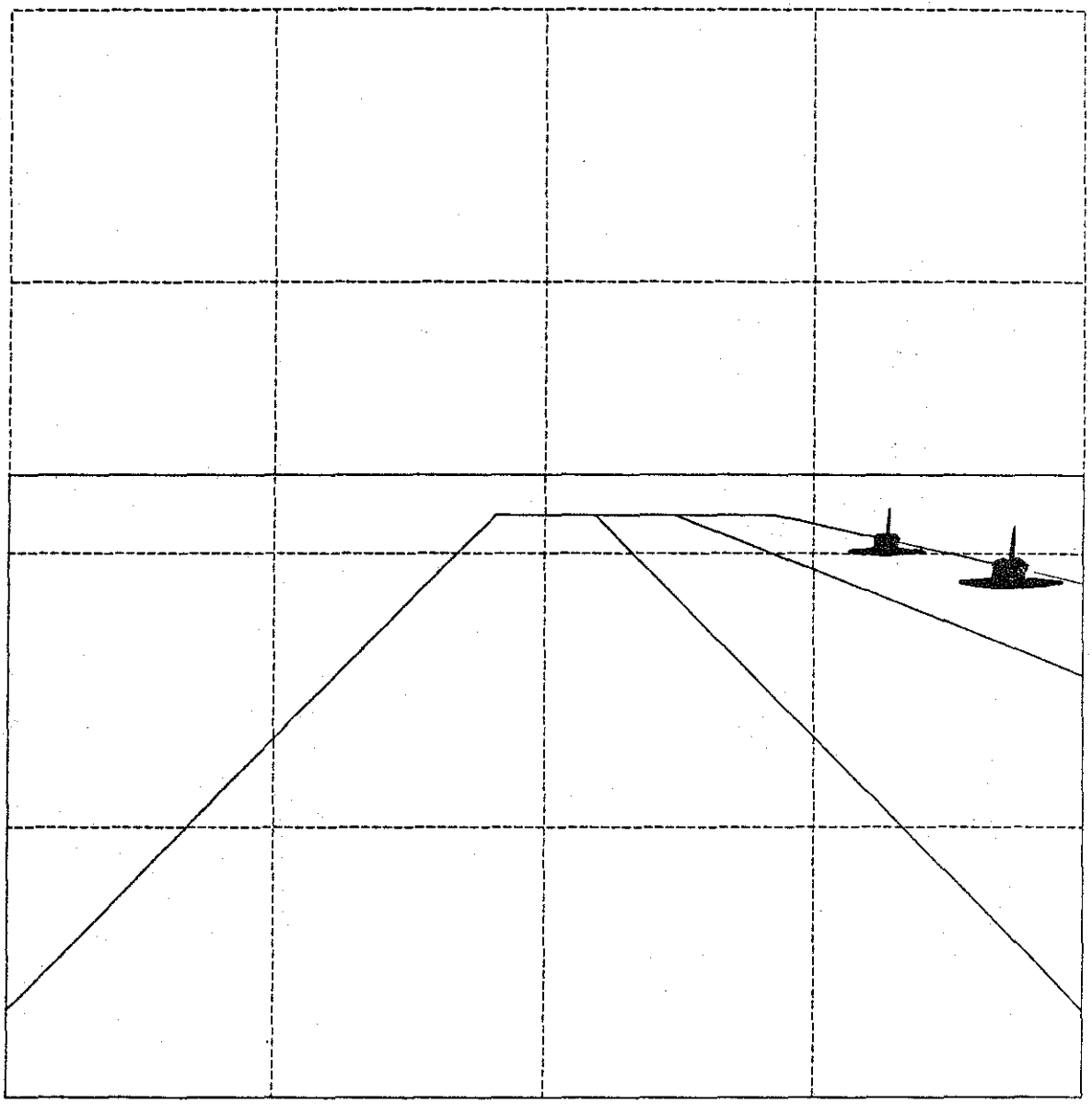
FIGURE 19 (Screen contains 498 polygons.)



MILLIONS OF MEMORY CYCLES REQUIRED TO DISPLAY SCENE

	uniprocessor	4-processor	16-processor
uniprocessor	5.99		
interlace		1.99	.966
splitter		2.87	.905
hybrid			.992

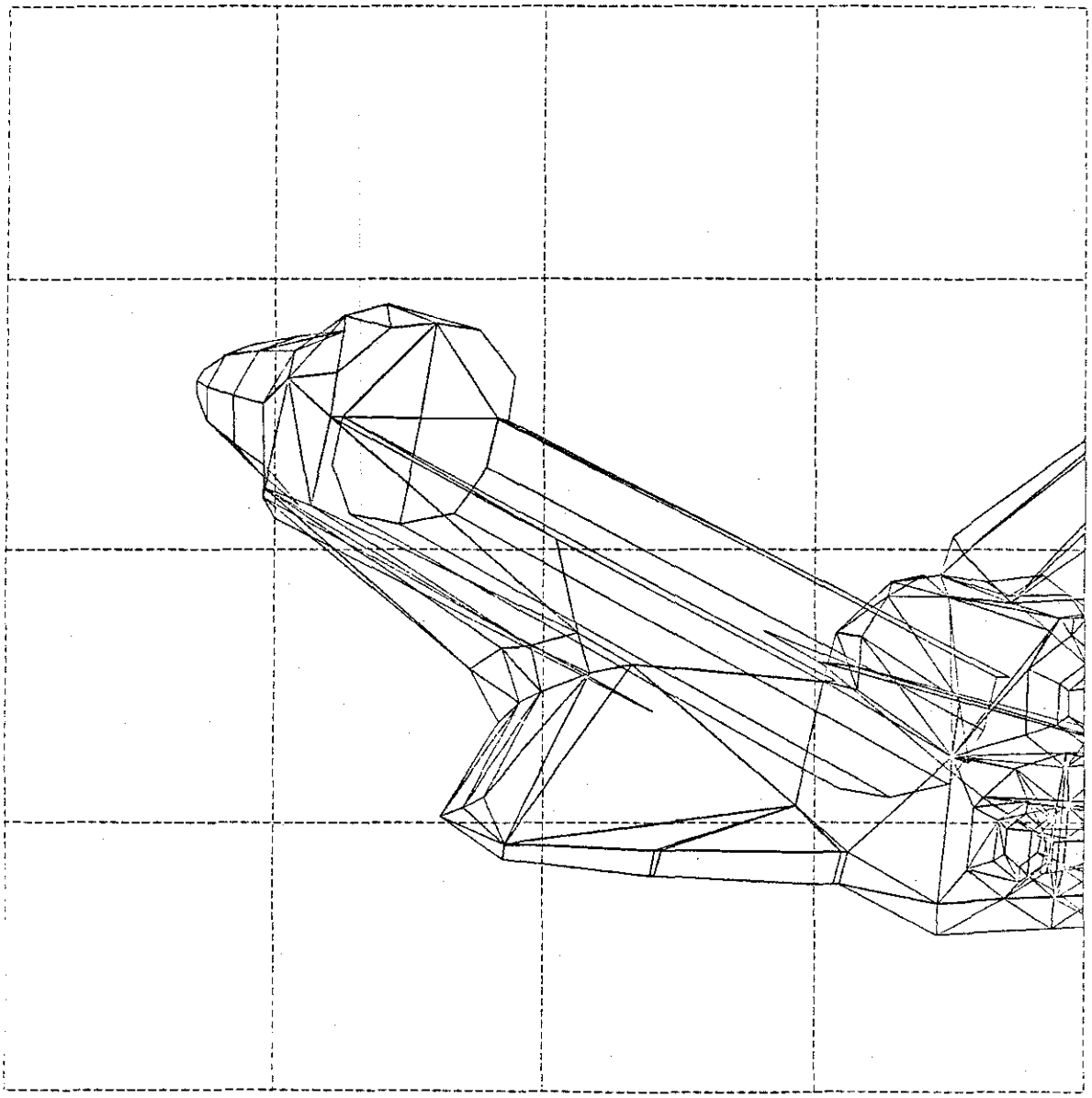
FIGURE 20 (Screen contains 492 polygons.)



MILLIONS OF MEMORY CYCLES REQUIRED TO DISPLAY SCENE

	uniprocessor	4-processor	16-processor
uniprocessor	5.83		
interlace		1.95	.959
splitter		3.01	1.14
hybrid			1.16

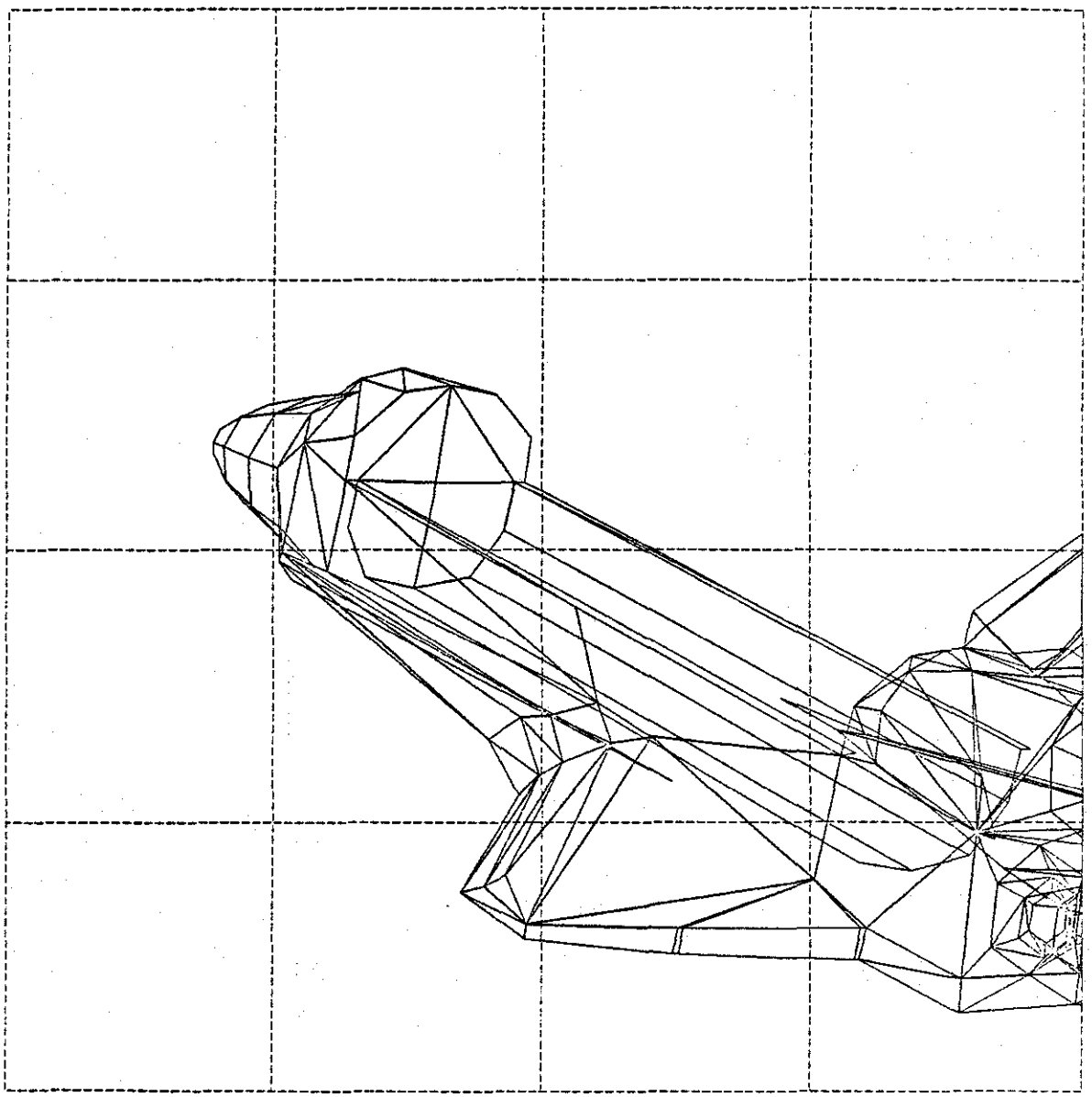
FIGURE 21 (Screen contains 492 polygons.)



MILLIONS OF MEMORY CYCLES REQUIRED TO DISPLAY SCENE

	uniprocessor	4-processor	16-processor
uniprocessor	3.32		
interlace		1.26	.620
splitter		2.16	1.07
hybrid			.878

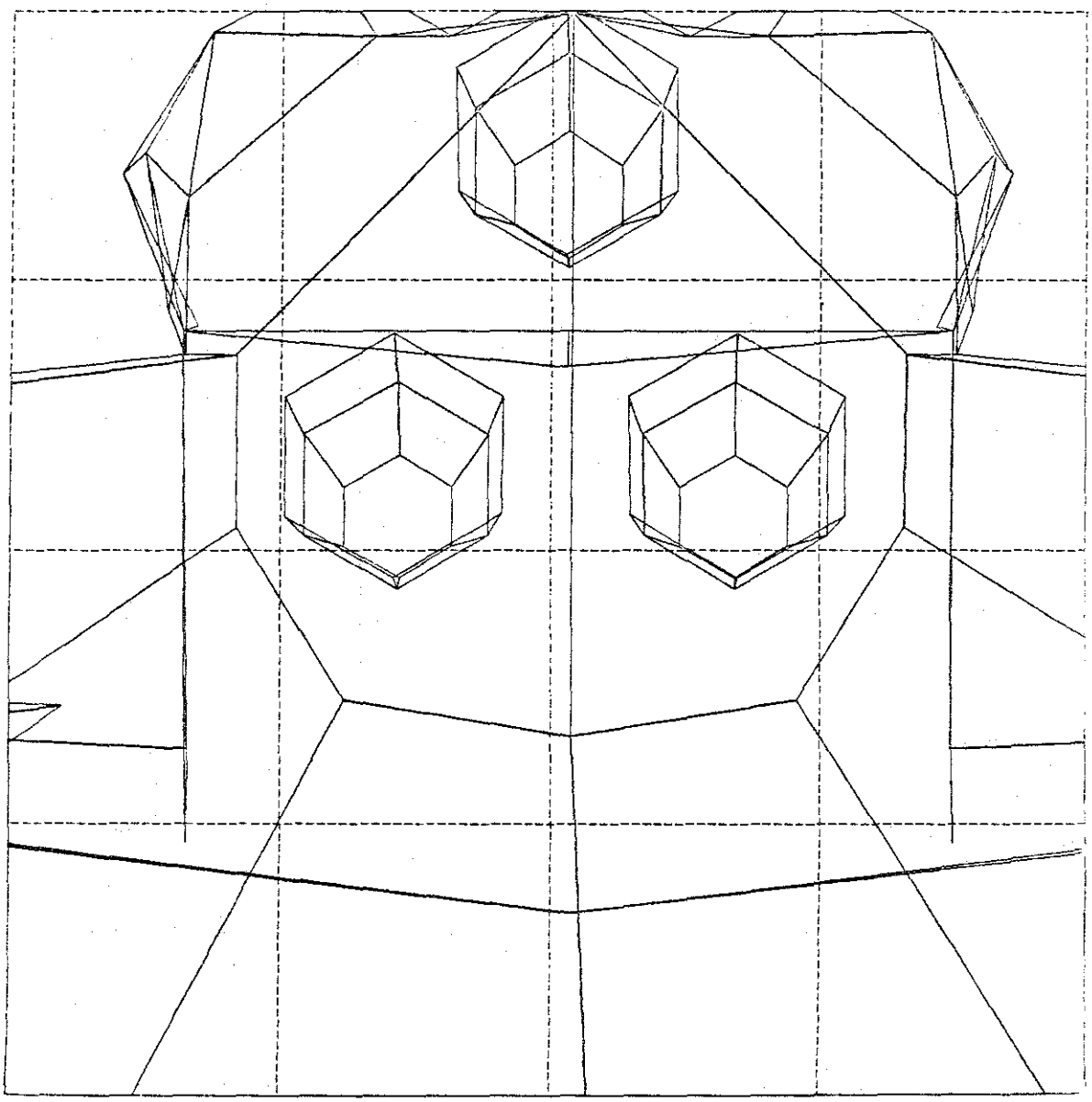
FIGURE 22 (Screen contains 196 polygons.)



MILLIONS OF MEMORY CYCLES REQUIRED TO DISPLAY SCENE

	uniprocessor	4-processor	16-processor
uniprocessor	3.17		
interlace		1.18	.569
splitter		2.21	.743
hybrid			.828

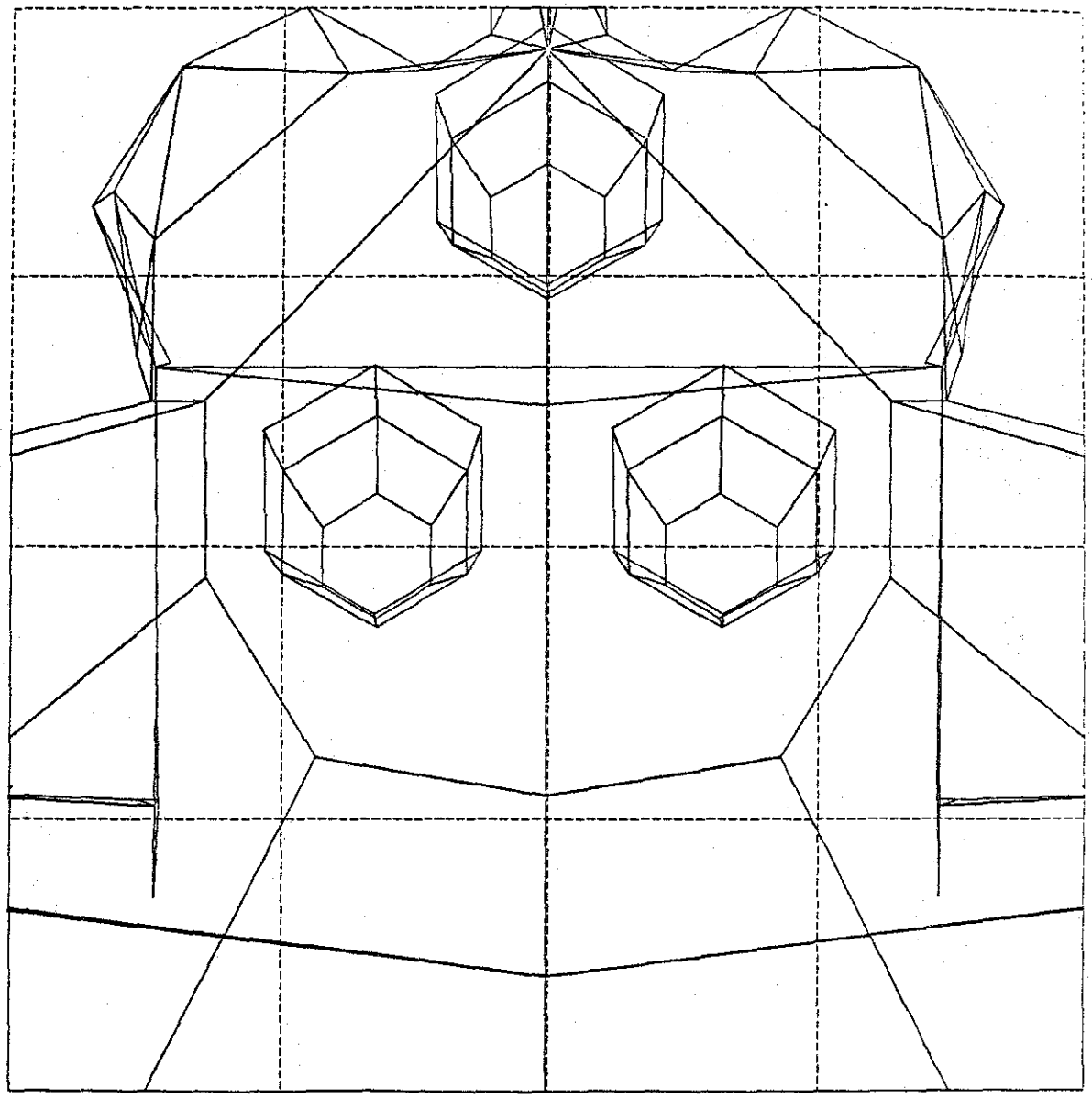
FIGURE 23 (Screen contains 172 polygons.)



MILLIONS OF MEMORY CYCLES REQUIRED TO DISPLAY SCENE

	uniprocessor	4-processor	16-processor
uniprocessor	6.80		
interlace		2.01	.703
splitter		1.97	.658
hybrid			.634

FIGURE 24 (Screen contains 86 polygons.)



MILLIONS OF MEMORY CYCLES REQUIRED TO DISPLAY SCENE

	uniprocessor	4-processor	16-processor
uniprocessor	6.77		
interlace		2.02	.720
splitter		1.83	.653
hybrid			.587

FIGURE 25 (Screen contains 94 polygons.)

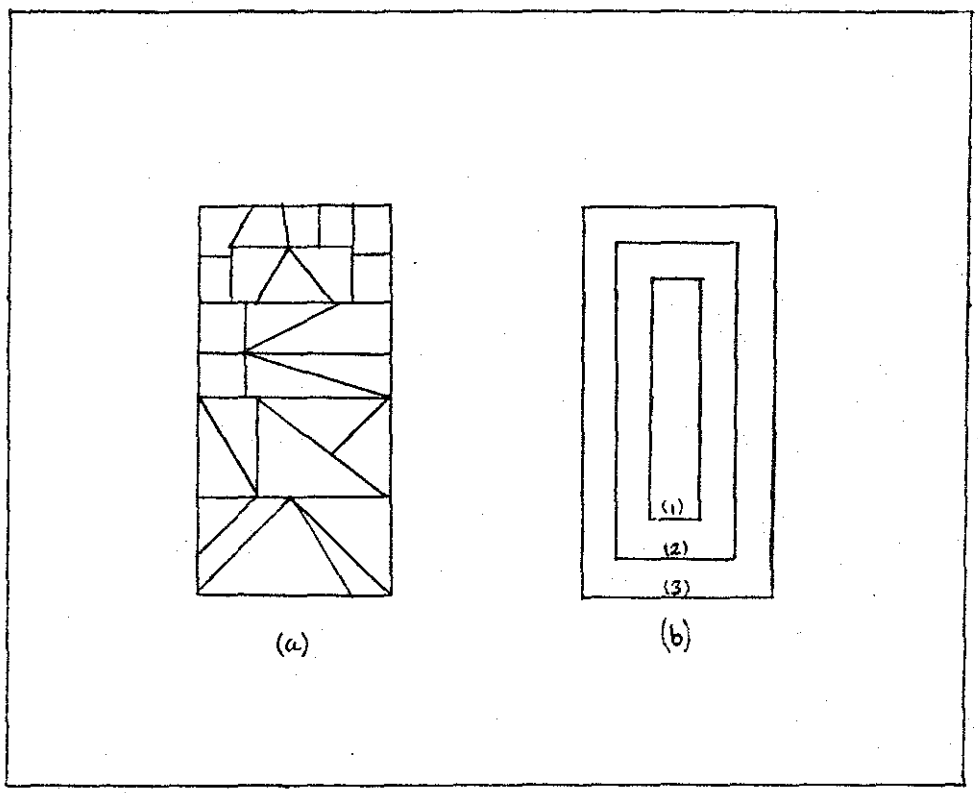
4.3 SCREEN COMPLEXITY: AREA VS. NUMBER OF POLYGONS

Suppose we were given one large and one small polygon to display, and two processors to display them. Would it be better to give each processor a polygon, or to have each processor display half of each polygon? In one case, we give one processor significantly more work to do (in terms of number of pixels to calculate). In the other, we double the polygon setup time, because both processors must set up both polygons. This section investigates the relationship between area and polygon overhead, and shows that polygon setup is relatively inexpensive when compared to displaying large polygons.

We should first consider what we mean by "screen complexity" and "distribution of polygons over the screen." One meaning of these terms refers to the placement of each polygons' center of mass on the screen. Another meaning is the distribution of depth complexity* over the screen. These concepts are related, but not identical. Consider figure 26.

---

* The depth complexity at a given pixel is the number of polygons which fall on that pixel; the depth complexity of a scene is the average number of polygons which fall over all the pixels.



(a) is a collection of non-overlapping polygons
(b) is 3 concentric polygons

(a) and (b) cover the same number of visible pixels, but polygon complexity is skewed toward (a), and depth complexity is skewed toward (b)

EXAMPLE OF SCREEN COMPLEXITY
FIGURE 26

The polygon placement is skewed to the left, but the depth complexity is skewed to the right. Detection and measurement of skewness was outside the scope of this project, although we will use the intuitive concepts.

The most obvious difference between the two schemes is that the interlace machine is relatively insensitive to non-uniform area[5] and polygon distributions, and the splitter architecture allows some processors to completely ignore some polygons (especially if the polygons are distributed uniformly).

To state this problem differently, we note that the algorithm depends on the following parameters:

number of polygons
number of vertices per polygon
height per polygon (in resolution units)
area per polygon

The interlace architecture attacks these problems from the bottom of the list, cutting the area and height of polygons in very regular and predictable ways. The splitter attacks this list from the top, reducing the number of polygons each micro must process.

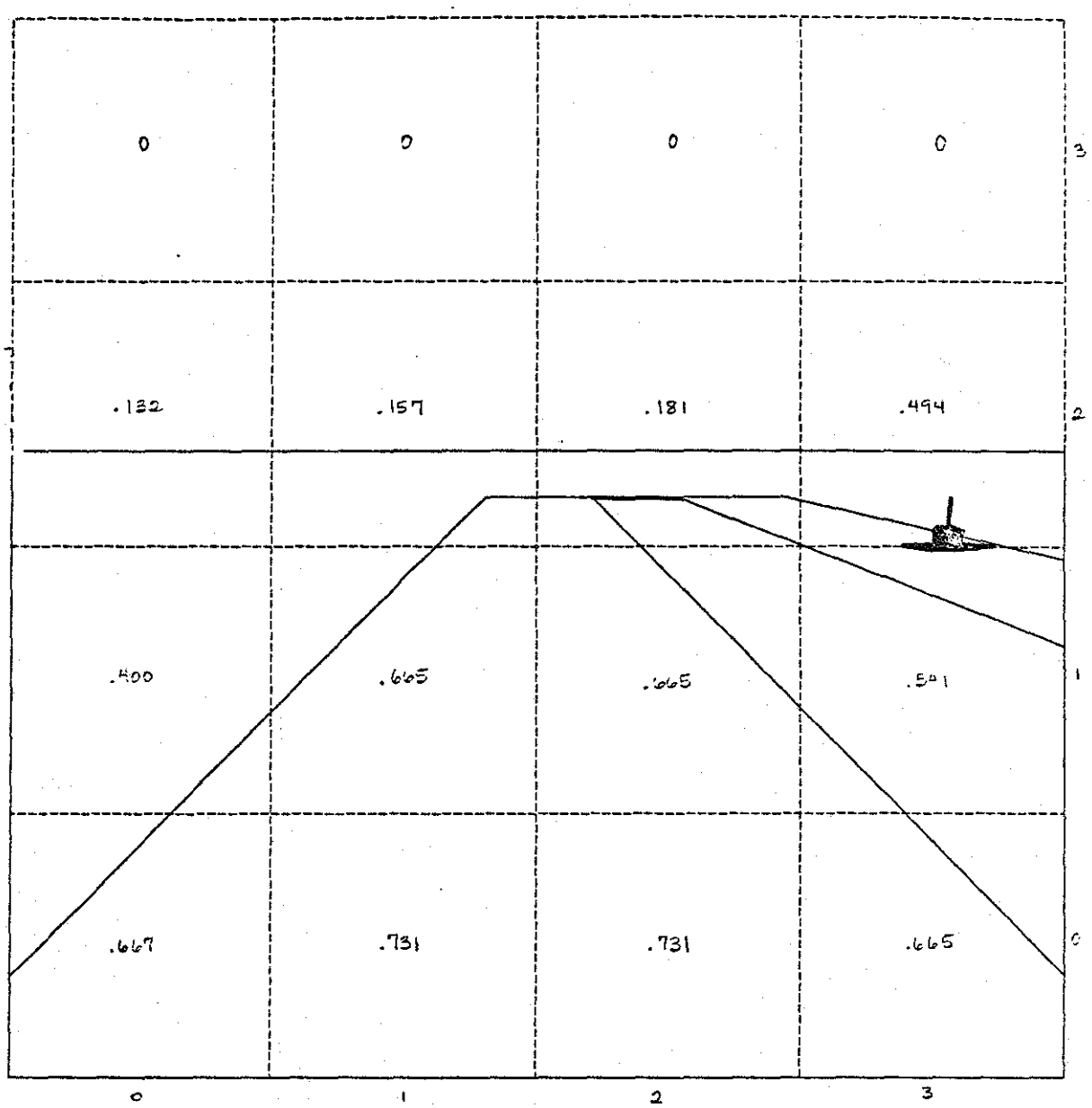
The relative importance of the number of polygons and total polygon area can be illustrated by figure 15.

As figure 27 shows, the 16-processor splitter machine does not spend most of its time working on section (3,2), as one might expect. Sections (1,0) and (2,0) require more time. This parodox can be understood by examining the relationship between area and polygon setup time.

---

[5] By area we mean the number of pixels a polygon covers. Thus, a single point has an area of 1. Depth complexity (Dc) and average area per polygon are related by the formula

Dc * Resolution_in_X * Resolution_in_Y
= number_of_polygons * average_area_per_polygon.



PROCESSING TIMES FOR EACH PROCESSOR IN A 16-MICRO SPLITTER MACHINE (IN MILLIONS OF MEMORY CYCLES)

FIGURE 27

Suppose we have a polygon which is 128x128 square on a 16-processor splitter machine (i.e. this polygon fills a micro's entire portion of the screen). How many polygons of one vertex (i.e. single points) can be processed in the time required to process one large polygon? We have

$$Gt + 4Et + 128St + (128**2)Pt = n(Gt + Et + St + Pt).$$

This system reduces to

$$n = (Gt + 4Et + 128St + (128**2)Pt) / (Gt + Et + St + Pt) = 573.$$

Thus, over 1000 point polygons can be processed in the time required to process the two polygons (runway and landscape) in sections (1,0) and (2,0).

If the small polygons are triangles whose area is 0.01 that of the total region (i.e. 12.8x12.8 pixels, average), the equation becomes

$$n = (Gt + 4Et + 128St + (128**2)Pt) / (Gt + 3Et + 12.8St + (12.8**2)Pt) = 55.$$

The point of this discussion is that, of the two kinds of complexity (number of polygons and total area), many, many small polygons are required to equal the complexity (in terms of processing time) of a very few large ones. Thus, reducing the area per processor is more important than simply reducing the number of polygons per processor; and if very few micros are to be used (say, around 4), distributing the total area to be processed is probably more important than attempting to reduce the number of polygons each processor must handle. The interlace scheme very effectively distributes total area among all processors, while the splitter scheme may or may not, depending on the particular scene to be processed. In fact, in the scenes analyzed below, the 4-processor interlace scheme was superior to the four processor splitter in all but two cases.

4.4 SPLITTER'S SENSITIVITY TO NON-UNIFORMLY DISTRIBUTED DATA

Since screen complexity is so important, one would expect the interlace architecture to have an advantage over the splitter architecture for four processor machines. However, when the number of processors is increased to 16, both machines have reduced the number of pixels each micro must cover from 64k to 16k. The question becomes: Has the splitter sufficiently reduced the size of each micro's screen? And, has the interlace scheme begun to encounter its problems with polygon overhead because each processor must examine each polygon? The answer to both these questions is unclear for the 16-processor machines, and the comparison of them is inconclusive. To examine the sensitivity of the 16-processor splitter to very slight scene changes, several scenes were selected, and then modified slightly to yield especially favorable and unfavorable divisions of the screen. To summarize the results, a slight change in scene caused as much as 68% increase in the time required to process two very similar scenes. These results are in figures 28, 29 and 30.

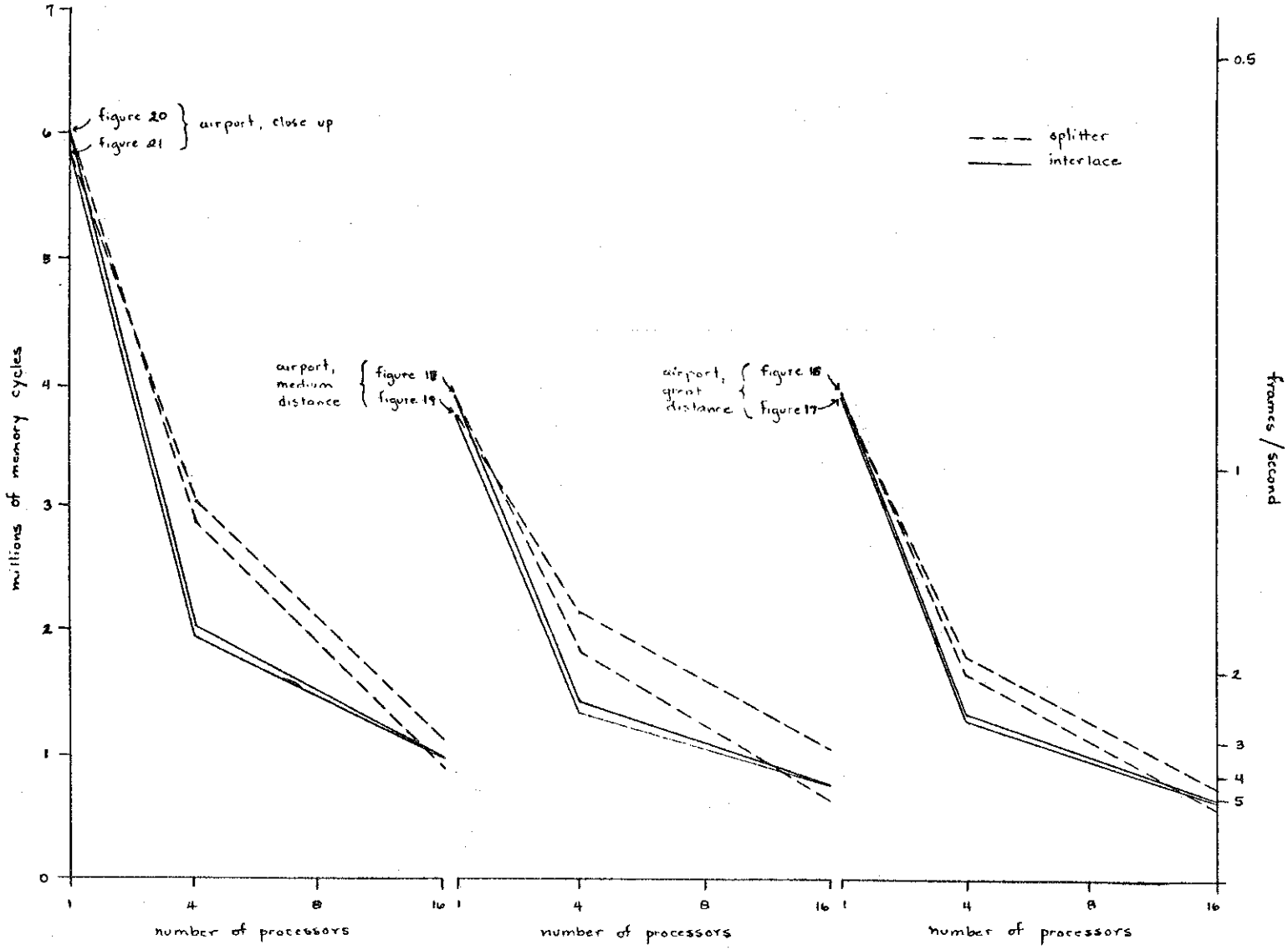

millions of memory cycles
frames / second
figure 20
figure 21
airport, close up
airport, medium distance
figure 18
figure 19
airport, great distance
figure 16
figure 17
splitter
interlace
number of processors
number of processors
number of processors

FIGURE 28 A

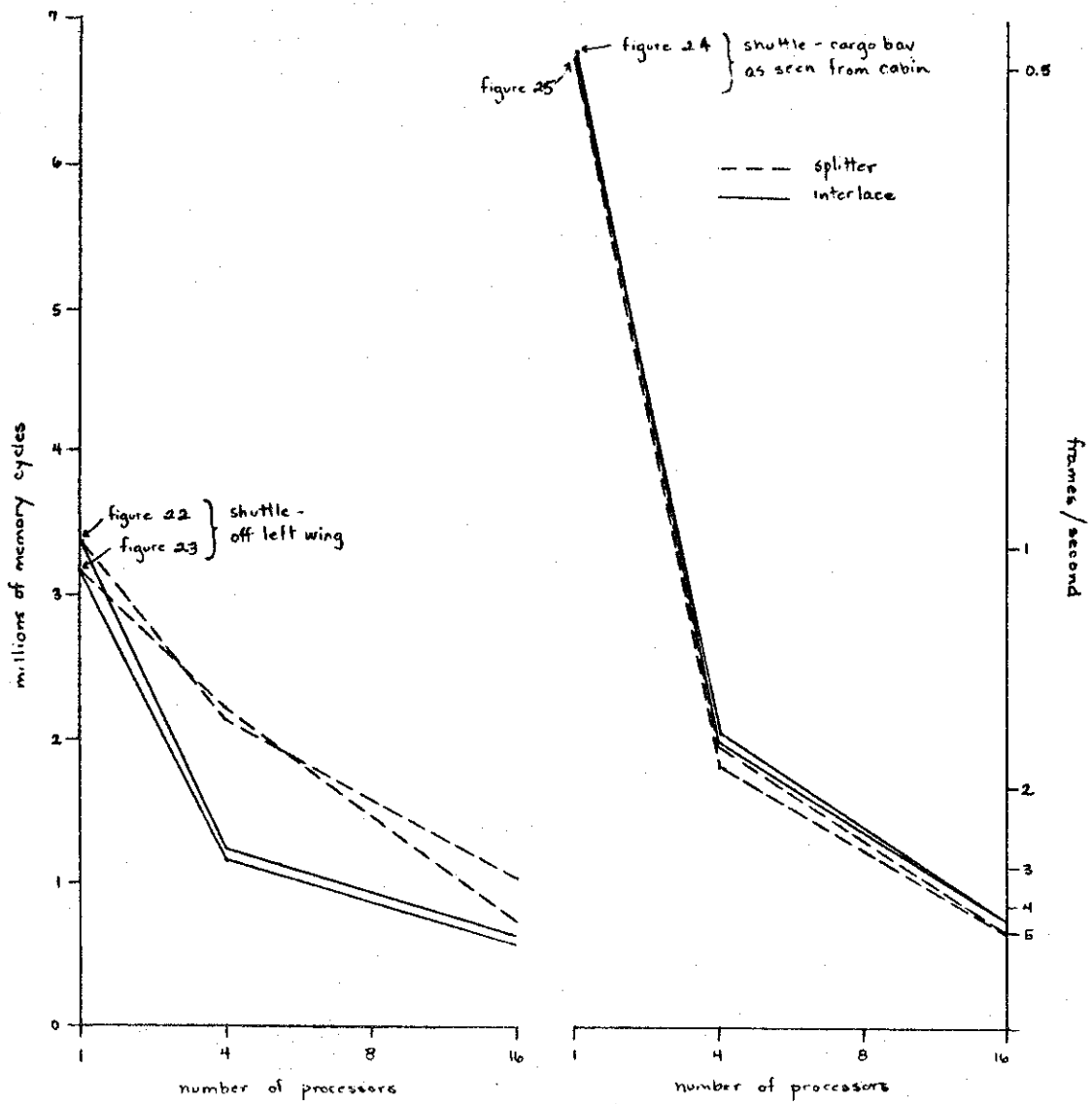

millions of memory cycles
0
1
2
3
4
5
6
7
figure 22
figure 23
shuttle - off left wing
figure 24
figure 25
shuttle - cargo bay as seen from cabin
splitter
interlace
frames / second
0.5
1
2
3
4
5
1
4
8
16
number of processors
1
4
8
16
number of processors

FIGURE 28 B

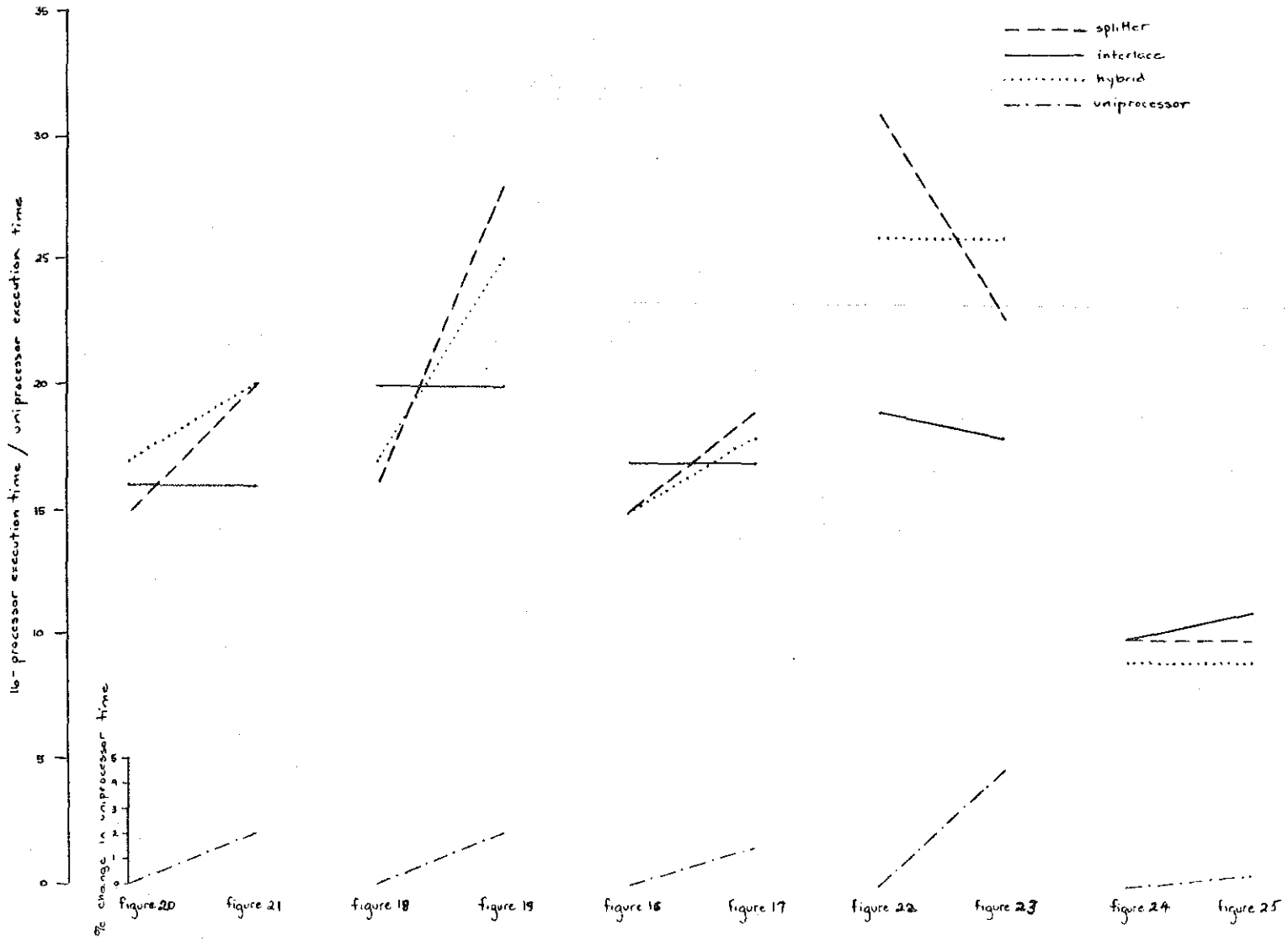

splitter
interlace
hybrid
uniprocessor
multi-processor execution time / uniprocessor execution time
35
30
25
20
15
10
5
0
% change in uniprocessor time
figure 20
figure 21
figure 18
figure 19
figure 16
figure 17
figure 22
figure 23
figure 24
figure 25

FIGURE 29

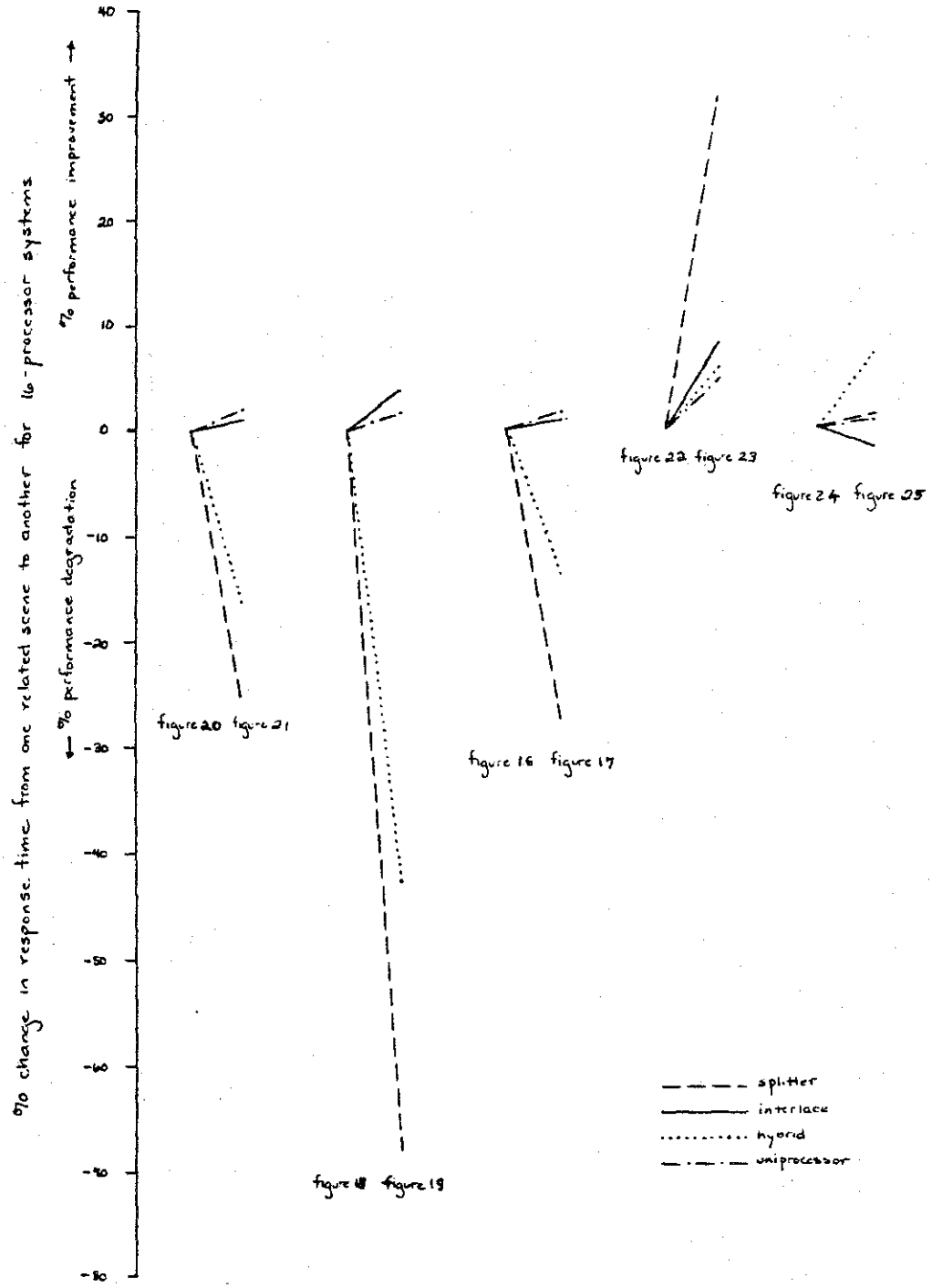

% change in response time from one related scene to another for 16-processor systems
% performance improvement →
← % performance degradation
40
30
20
10
0
-10
-20
-30
-40
-50
-60
-70
-80
figure 20 figure 21
figure 18 figure 19
figure 16 figure 17
figure 22 figure 23
figure 24 figure 25
splitter
interlace
hybrid
uniprocessor

FIGURE 30

The performance results for the 16 processor interlace and splitter machines working on figures 16 through 25 are given in figures 28, 29 and 30. Figure 28 shows the amount of time required by each processor. The times given in seconds and frames per second are assuming 300 nsec memory, and that memory cycle time exactly equals processing time.

As the "execution time" graphs (figure 28) show, the splitter architecture may be very effective in dividing a given scene, but a small change in that scene may degrade its performance dramatically. The interlace pattern, on the other hand, is always between these two extremes. To put this increase into absolute numbers for scenes 18 and 19 (the scenes with the largest percentage variation), the "good" split would have required .189 seconds to display on a 16-processor splitter. The slower "poor" split would require .318 seconds (with both estimates assuming 300 nsec. memory cycle time, and that the execution time exactly equals the memory cycle time). The "good" split would yield around 5 frames per second; the poor one only 3. By contrast, the interlace machine would produce around 4 frames per second for both scenes; the differences between the processing times are insignificant.

One should remember that the purpose of these small scene changes was to investigate how the 16-processor splitter compared with the 16-processor interlace scheme. Thus, the figures for the 4-processor splitter architecture should not be overemphasized. Still, the interested reader may want to compare these graphs to those in [Parke80].

Figure 29 shows in more detail how the 16-processor machines reacted to slight scene changes relative to the uniprocessor model. In going from figure 18 to 19, the splitter's time increased from 16% to 28% of the uniprocessor's time. The interlace scheme's greatest variation was 1%. Figure 29 also shows the percentage change in uniprocessor time; the largest variation was 4.5%.

Figure 30 attempts to show how changes in the processing time of similar scenes would be perceived by a user at a display. For example, assume that someone is using a 16-processor interlace system to display figure 18. This scene would take around .237 seconds to generate. If the user moves quickly to figure 19, the processing time decreases to .228 seconds. This represents a relative performance improvement of around 4%. A splitter system's performance would change from .189 seconds per frame to .318 seconds per frame, a relative performance degradation of 68%.

In figure 30, the splitter scheme's relative performance changes dramatically, while the interlace scheme's perfor-

mance changes very little for small scene changes. The percentage change in uniprocessor times have also been given in figure 30, in the absence of a good measure of scene complexity. Of course, the uniprocessor's performance changes little.

4.5 EFFECTS OF POLYGON OVERHEAD ON INTERLACE SCHEME

As Parke notes ([Parke80]), the main problem with the interlace scheme is that each processor must process each polygon. The effects of this can be seen in the landscape scene statistics of the 16-processor interlace machine. In all but one scene the slowest processor spent over 50% of its time in polygon setup and edge (vertex) processing. As the number of polygons in the scene increases, or as the number of processors increase, this effect will be more and more pronounced. In fact, one can roughly estimate the time required for a given processor interlace machine. Consider a n**2 processor machine, with an nxn interlace pattern. The average polygon area per processor will be 1/n**2 that of the uniprocessor system. And the average height per polygon will be reduced by 1/n. If we extrapolate to a 256 processor machine (in a 16x16 interlace pattern) operating on these same scenes, 75% of each processor's time will be spent in polygon overhead and vertex processing. Adding more processors can only improve performance by 25%, at most. Thus, the interlace scheme quickly encounters the problems of diminishing returns for many processors.

4.6 PARKE'S HYBRID SCHEME

To summarize each machine's weaknesses, the splitter suffers from non-uniform data distributions which overload individual processors. The interlace machine pays very high overhead costs because each micro must process each polygon.

One scheme which attempts to solve these problems is Parke's hybrid scheme [Parke80]. A 16-processor hybrid computer splits the screen into several large chunks (say 4) and then has a number of processors (say 4) assigned to each chunk in an interlace fashion. As appealing as this might seem at first, this scheme is not markedly superior to either the straight interlace or splitter schemes. The reason is that it splits the screen into large chunks (and the chunks can have significantly different amounts of work to do), and then pays for each processor in the chunk to process each polygon. In other words, this scheme contains the elements of the worst of both worlds, as well as the best.

Chapter V

CONCLUSIONS

5.1 SUMMARY OF SIMULATION RESULTS

The scenes of the shuttle proper demonstrate clearly the strengths and weaknesses of the two schemes. In cases where screen complexity is spread relatively evenly over the screen (e.g. the cargo bay), the splitter is clearly the better scheme. In cases where complexity is hopelessly skewed (e.g. the shuttle profile), the interlace scheme is preferred.

The airport landscape scenes demonstrate a middle ground, where neither machine is clearly superior. If one imagines figures 16 through 21 to be snapshots taken from a plane approaching a runway, then figure 31 attempts to plot execution time as a function of the plane's position on this approach path. Execution times for similar scenes are connected to show how performance changes with small changes in scene. The reader is cautioned that often the change in processing time is more attributed to a change in target than a change in the position of the viewer alone.

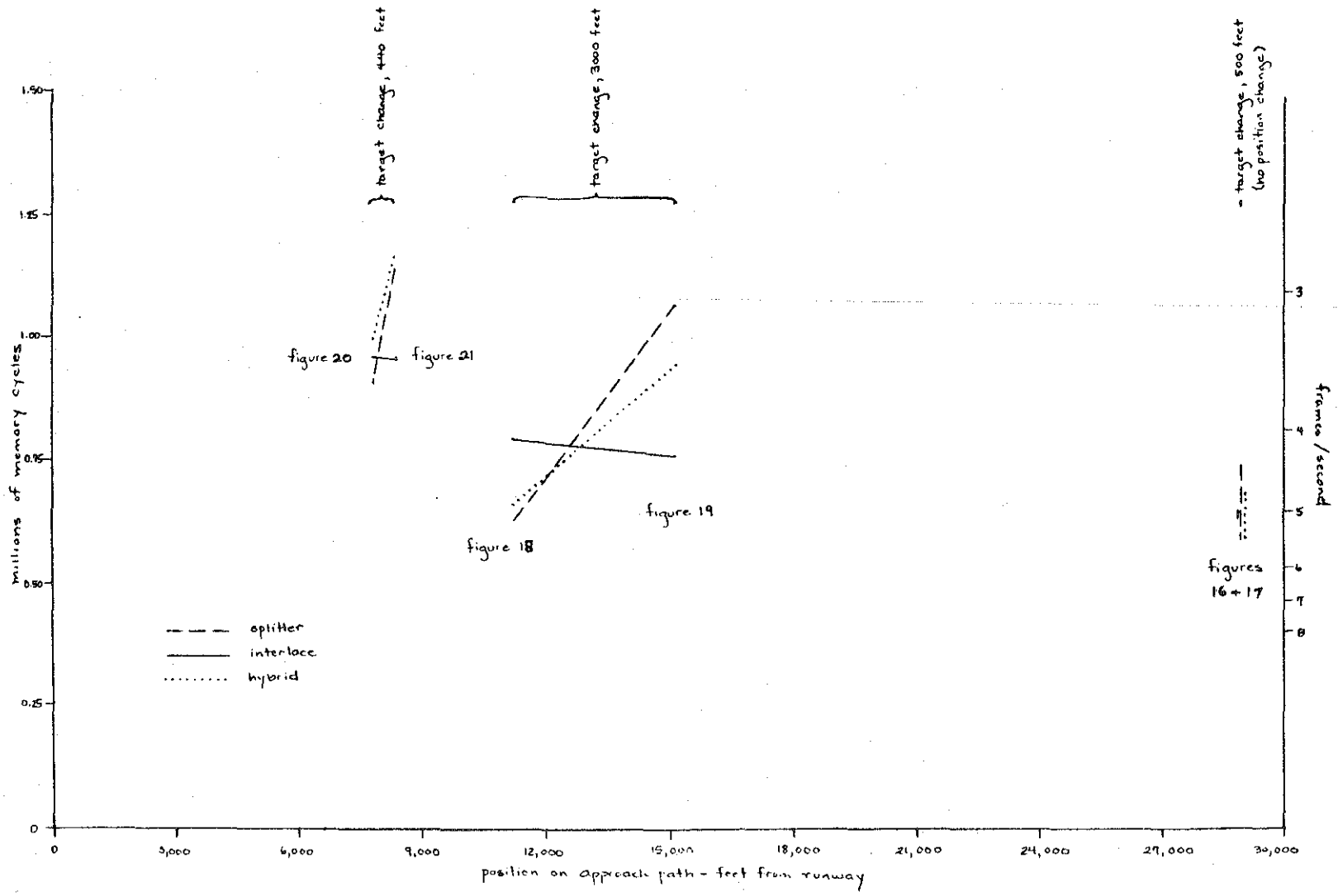

target change, 440 feet
target change, 3000 feet
target change, 500 feet (no position change)
figure 20
figure 21
figure 18
figure 19
figures 16 + 17
millions of memory cycles
frames / second
1.50
1.25
1.00
0.75
0.50
0.25
0
3
4
5
6
7
8
0
3,000
6,000
9,000
12,000
15,000
18,000
21,000
24,000
27,000
30,000
position on approach path - feet from runway
optifier
interlace
hybrid

FIGURE 31

In these scenes, the splitter reacted to small changes in the scene by large changes in processing time. The interlace scheme was not greatly affected by these small changes.

5.2 CONCLUSIONS

As noted earlier, the single most important problem is the total number of pixels a processor must handle. Thus, one does not want to split the screen into large, contiguous chunks, because the complexity of the scene (both in the number of polygons and amount of area) can vary greatly with a very small variation in scene. Since the interlace scheme effectively divides the screen area, for few processors (say 4) the interlace pattern is preferred. As the number of processors increases to 256, the interlace pattern clearly spends too much of its time in polygon setup and edge processing; thus the splitter is preferred if an individual processor is responsible for a relatively small area of the screen. If one is to build a machine with an intermediate number of processors (say, 16), the choice (at least from these results) is less clear; the two schemes are fairly close. The splitter scheme still suffers from the large area per processor problem, and thus, its times for similar scenes vary widely. On the other hand, the times for the interlace scheme vary little; however, this scheme is starting to show the effects of the polygon overhead problem.

5.3 FURTHER RESEARCH

Although the statistical characteristics of typical scenes can strongly influence the performance of certain graphics machines, very little work has been done in this area. With the exception of [Suthe72], almost nothing is known about graphics data. Hence, one designer created a scheme which depends heavily on a uniform distribution of polygons over the screen, and another explicitly assumed the opposite. To combat this scarcity of information, we have included typical raw scene statistics in an appendix. Prior knowledge about the nature of graphics data can only help in the design of future machine. While these statistics are hardly a definitive work, they may provide a base for more work later.

One problem we were not able to solve was finding a metric which would relate the statistics for a uniprocessor system to a splitter system. In section 4.5, we developed an analytical method for estimating the time required for an interlace processor to display a given scene, given the sta-

tistics (number of polygons, average height, average width, average area) of the scene as a whole. We could not find a simple technique for estimating the time required for the splitter architecture, because the splitter depends on the placement of the polygons over the screen. The two dimensional clustering of both depth complexity and number of polygons implies that the scene must be split, and each section analyzed separately.

Clark and Hanna in [Clark80] have introduced a scheme similar to the interlace architecture. Their system is designed for VLSI displays, but could be easily expanded to execute a Z-buffer algorithm. Given a model expressing their system's processing time in terms of data characteristics, their system's performance could be modeled easily using the techniques of this project. Of course, since they are using a radically different implementation (custom VLSI chips instead of programmed general purpose microprocessors), the bottlenecks of their system may be completely different from those in splitter and interlace systems. Even so, a study of the characteristics of typical images to be generated would be very useful for the design of future custom display schemes.

Henry Fuchs [Fuchs80] has suggested that his interlace architecture could be improved by freeing each micro from doing polygon setup and many of the edge calculations. The thrust of this idea is that the polygons could be broadcast to the micros with (for example) the top vertex already located, and all the edge increments already calculated. This could be accomplished by giving each processor different polygons. Each micro then does the common setup on its polygons, and broadcasts them to other micros in semi-digested form at the appropriate time. This modification would alleviate the scheme's polygon setup time problem, but would require more transmission time and memory. This topic merits further study.

As mentioned in section 2.3, some vertices must be processed by both the left and right hand side code. However, this effect can be compensated for by testing to determine whether or not the delta calculations must, in fact, be done. While the net result of these two facts is negligible for the uniprocessor machine, the multiprocessor machines can be affected greatly.

Specifically, if one tests to see whether or not delta calculations are required for an interlace machine, the time required to process the scene of figure 18 drops by around 20%. This topic, also, merits futher research.

REFERENCES

[Clark80] J. H. Clark, M. R. Hannah, "Distributing Processing in a High-Performance Smart Image Memory," Lambda, Volume 1, Number 3, Fourth Quarter, 1980.

[Fuchs77] H. Fuchs, "Distributing a Visible Surface Algorithm Over Multiple Processors," Proceedings of the 1977 ACM Annual Conference, Seattle, Washington, October 1977.

[Fuchs79] H. Fuchs, B. Johnson, "An Expandable Multiprocessor Architecture for Video Graphics," Proceedings of the 6th Symposium on Computer Architecture, April 1979.

[Fuchs80] H. Fuchs, personal communication.

[Fuller77] S. H. Fuller, W. E. Burr, "Measurement and Evaluation of Alternative Computer Architectures," Computer, October, 1977.

[Newman79] W. M. Newman, R. F. Sproull, Principles of Interactive Computer Graphics, McGraw-Hill, 2nd Edition, 1979.

[Parke79a] F. I. Parke "A Parallel Architecture for Shaded Graphics," Technical Report, Computer Engineering Dept., Case Western Reserve University, Jan 1979.

[Parke79b] F. I. Parke, "Performance Analysis of Z-Buffer Convex Tiler Based Shaded Image Generation," Technical Report CES 79-15, Computer Engineering Department, Case Western Reserve University, October 1979.

[Parke80] F. I. Parke, "Simulation and Expected Performance Analysis of Multiple Processor Z-Buffer Systems," SIGGRAPH 1980, pp 48-53.

[Schac81] B. J. Schachter, "Computer Image Generation for Flight Simulation," IEEE Computer Graphics and Applications, October, 1981.

[Shohat77] M. Shohat, J. Florence, "Application of Digital Image Generation to the Shuttle Mission Simulation," Proceedings of the 1977 Summer Computer Simulation Conference, 1977.

[Suther74] I. E. Sutherland, R. F. Sproull, R. A. Schumacker, "A Characterization of Ten Hidden Surface Algorithms," Computing Surveys, March 1974.

Appendix A

PROGRAM LISTINGS IN C

This appendix contains a C routine which implements the Z-buffer algorithm described in Chapter 2. The routine contains 8 modules which are simply concatenated together. The modules (in order) are:

variables.c
polybody1.c
edgebody1.c
segmentbody1.c
pixelbody.c
segmentbody2.c
edgebody2.c
polybody2.c

The module "variables.c" contains variable declarations. Generally, most of the processing done by polybody1 concerns polygon setup, and most done by edgebody1 concerns edge (i.e. vertex) processing, etc.

variables.c

```
/* number of dimensions per vertex (x, y, z, shading) */
#define vertex_size     4

/* subscript of 'x' values in poly */
#define x 0

/* subscript of 'y' values in poly */
#define y 1

/* subscript of 'z' values in poly */
#define z 2

/* subscript of 'sh' values in poly */
#define sh 3

/* starting points, step size, and image resolution */
#define xstart  0
#define ystart  0
#define xstep   4
#define ystep   4
#define xsize   128
#define ysize   128

/* variables concerned mainly with pixel calculation */
int     *pi_im_ptr,     /* ptr to image buffer for current pixel */
        *pi_z_b_ptr,    /* ptr to z buffer for current pixel */
        *pi_z_s_ptr,    /* ptr to z buffer for last pixel to paint */
        pi_z,           /* distance for current polygon point */
        pi_dz,          /* increment for pi_z */
        pi_sh,          /* shading for current polygon point */
        pi_dsh;         /* increment for shading value (pi_sh) */

/* variables concerned with segment calculation */
int     seg_y,          /* current y value (row designator) */
        seg_z_l,        /* z value for left scanline endpoint */
        seg_z_r,        /* z value for right scanline endpoint */
        seg_sh_l,       /* shading value--left endpoint of scanline */
        seg_sh_r,       /* shading value--right endpoint of scanline */
        seg_x_l,        /* leftmost x value for current scanline */
        seg_x_r,        /* rightmost x value for current scanline */
        seg_i,seg_j;    /* temporaries */
int                     /* row pointers */
        (*seg_zr_ptr)[xsize/xstep],     /* ptr to current row of z buffer */
        (*seg_ir_ptr)[xsize/xstep];     /* ptr to current row of image buffer */

/* variables for edge calculation */
int     ed_dx_l,        /* delta value for x intercept, left side */
        ed_dx_r,        /* delta value for x intercept, right side */
        ed_dz_l,        /* delta value for z, left side */
        ed_dz_r,        /* delta value for z, right side */
        ed_dsh_l,       /* delta value for shading, left side */
        ed_dsh_r,       /* delta value for shading, right side */
        ed_x_l_skip,    /* distance to next assigned pixel (left side) */
        ed_x_r_skip,    /* same as above, right side */
        ed_y_l_skip,    /* distance to next assigned line (left side) */
        ed_y_r_skip,    /* distance to next assigned line (right) */
```

variables.c

```
        ed_c_y_l,       /* current left vertex's y value */
        ed_c_y_r,       /* current right vertex's y value */
        ed_n_y_l,       /* next left vertex's y value */
        ed_n_y_r,       /* next right vertex's y value */
        *ed_c_v_l_ptr,  /* current left vertex pointer */
        *ed_c_v_r_ptr,  /* current right vertex pointer */
        *ed_n_v_l_ptr,  /* next left vertex pointer */
        *ed_n_v_r_ptr,  /* next right vertex pointer */
        *ed_mx_v_ptr,   /* pointer to last vertex in poly array */
        ed_i;           /* scratch variable */

int                     /* variables for polygon setup */
        po_x_l,         /* x value of top leftmost vertex */
        po_x_r,         /* x value of top rightmost vertex */
        po_min_y,       /* y value of bottom of polygon */
        *po_poly_ptr,   /* temporary pointer into polygon */
        po_n_vert,      /* number of vertices in this polygon */
        poly[10][vertex_size],  /* area to store a polygon */
        po_y,           /* y value loop temp */
        po_i,po_j,po_k; /* temporaries */

int     i;              /* temporaries */

int     image[ysize/ystep][xsize/xstep],        /* image buffer */
        z_buf[ysize/ystep][xsize/xstep];        /* z buffer */
```

polybodyl.c

```
/* This section reads in number of vertices (po_n_vert) and
 * the polygon vertices.  The polygon is stored in "poly."
 * This code also finds the topmost right and left side
 * vertices.
 */


/* Macro to go around a polygon counterclockwise (i.e. scan
 * to the left).
 */
#define vertl(ptr)      ((ptr>=ed_mx_v_ptr)? &poly[0][0]: ptr+vertex_size)

/* Macro to go around a polygon clockwise (i.e. scan to the right).
 */
#define vertr(ptr)      ((ptr<=poly) ? ed_mx_v_ptr : ptr-vertex_size)

/* Macro for group algebra calculation to move from any given line to the
 * next interesting line.
 */
#define groupy(line)    (((line) < 0) ? ystep + (line) : (line))


/* Read the number of vertices. 'eof' means quit and go home. */

while (scanf("%d", &po_n_vert) != EOF)
        {
        for (po_i = 0; po_i < po_n_vert; po_i++)
                {
                po_poly_ptr = &poly[po_i][0];

                /* Read x y z sh and point code (which is tossed). */
                scanf("%d %d %d %d %*d",&po_poly_ptr[x],&po_poly_ptr[y],
                                &po_poly_ptr[z], &po_poly_ptr[sh]);
                }

        /* Find high and low vertices for both left and right sides.
         * Since we assume the polygons are described in a counter-
         * clockwise orientation, "down" the structure poly goes
         * counterclockwise and thus comes to the top of the polygon
         * from the right.
         */
        ed_c_v_r_ptr = ed_c_v_l_ptr = ed_mx_v_ptr = &poly[po_n_vert - 1][0];
        po_min_y = po_y = ed_mx_v_ptr[y];       /* highest y value so far */
        po_x_l = po_x_r = ed_mx_v_ptr[x];
        po_poly_ptr = ed_mx_v_ptr - vertex_size;

        while (po_poly_ptr >= &poly[0][0])
                {
                po_k = po_poly_ptr[y];
                if (po_k > po_y)
                        {
                        ed_c_v_r_ptr = ed_c_v_l_ptr = po_poly_ptr;
                        po_y = po_k;
                        po_x_l = po_x_r = ed_c_v_l_ptr[x];
                        }
```

polybody1.c

```
                    else {
                         /* Since we haven't hit a new 'high', check to
                          * see if we're going along a horizontal (top)
                          * edge.  If so, make sure that we keep left &
                          * right pointers correct.
                          */
                         if (po_k == po_y)
                                 {
                                 /* case of poly_ptr[x] */
                                 if (po_poly_ptr[x] < po_x_l)
                                         {
                                         ed_c_v_l_ptr = po_poly_ptr;
                                         po_x_l = po_poly_ptr[x];
                                         }
                                    else if (po_poly_ptr[x] > po_x_r)
                                         {
                                         ed_c_v_r_ptr = po_poly_ptr;
                                         po_x_r = po_poly_ptr[x];
                                         }
                                 }
                            else if (po_min_y > po_k) po_min_y = po_k;
                         }
                    po_poly_ptr -= vertex_size;
                    }

/* Initialize values for first go through edge code. */
        ed_n_y_l = ed_n_y_r = ysize + 1;
        ed_n_v_r_ptr = vertr(ed_c_v_r_ptr);
        ed_n_v_l_ptr = vertl(ed_c_v_l_ptr);
        seg_y = ed_c_v_l_ptr[y];
        i = groupy(ystart - (seg_y % ystep));
        if (i != 0) seg_y -= ystep - i;

/* Set up pointers to current row of z and image buffers.
 * i.e. set up pointers to top row of current polygon.
 */
        i = seg_y / ystep;
        seg_zr_ptr = &z_buf[i][0];
        seg_ir_ptr = image[i];
```

edgebody1.c

```
/* macro to calculate the next vertex along the left edge
 * of the polygon.
 */
#define nextl(ptr)      (ptr > ed_mx_v_ptr) ? &poly[0][0]: ptr


/* Macro to calculate the next vertex along the right edge
 * of the polygon.
 */
#define nextr(ptr)      (ptr <= (&poly[1][x])) \
                                ? ed_mx_v_ptr \
                                : ptr - (vertex_size << 1)

/* Macro for max and min functions. */
#define max(i,j)        (i<j) ? j: i
#define min(i,j)        (i<j) ? i: j

/* Macro for group algebra calculation of distance from current line to next
 * interesting line.
 */
#define groupx(a)       (((a) < 0) ? xstep + (a) : (a))


/* Loop to do all affected segments--
 * while the left side y values are still going down,
 * continue the processing.  When they start going
 * back up, we know we've rounded the bottom of the
 * polygon and are through.
 * The test is made after the left edge is updated,
 * instead of a more conventional loop control.
 */

while (seg_y >= po_min_y)
        {

        /* set up left edge if necessary */
        if (seg_y <= ed_n_y_l)
             do {
                seg_x_l = *ed_c_v_l_ptr++;
                ed_c_y_l = *ed_c_v_l_ptr++;
                                        /* how far away is the next
                                         * interesting line?
                                         */
                ed_y_l_skip = ed_c_y_l - seg_y;
                pi_z = seg_z_l = *ed_c_v_l_ptr++;
                pi_sh = seg_sh_l = *ed_c_v_l_ptr++;
                ed_c_v_l_ptr = ed_n_v_l_ptr;
                ed_n_y_l = ed_n_v_l_ptr[y];
                ed_i = ed_c_y_l - ed_n_y_l;
                ed_dx_l = (*ed_n_v_l_ptr++ - seg_x_l) / ed_i;
                seg_x_l += ed_dx_l * ed_y_l_skip;
                                        /* force to the correct pixel
                                         * on our next line.
                                         */
                ed_x_l_skip = groupx(xstart - (seg_x_l % xstep));
                seg_x_l += ed_x_l_skip;
```

edgeboóyl.c

```
        eá_áx_l *= ystep;
        eá_n_v_l_ptr++; /* skip y */
        eá_áz_l = (*eá_n_v_l_ptr++ - seg_z_l) / eá_i;
                                /* repeat above x calculations for
                                 * z anâ sh.
                                 */
        pi_z = seg_z_l += eá_áz_l * eá_y_l_skip;
        eá_áz_l *= ystep;
        eá_ásh_l = (*eá_n_v_l_ptr++ - seg_sh_l) / eá_i;
        pi_sh = seg_sh_l += eá_ásh_l * eá_y_l_skip;
        eá_ásh_l *= ystep;
        eá_n_v_l_ptr = nextl(eá_n_v_l_ptr);
        } while ((seg_y < eá_n_y_l) && (eá_n_y_l < eá_c_y_l));


/* set up right edge if necessary */
if (seg_y <= eá_n_y_r)
     áo {
        seg_x_r = *eá_c_v_r_ptr++;
        eá_c_y_r = *eá_c_v_r_ptr++;
        eá_y_r_skip = eá_c_y_r - seg_y;
        seg_z_r = *eá_c_v_r_ptr++;
        seg_sh_r = *eá_c_v_r_ptr++;
        eá_c_v_r_ptr = eá_n_v_r_ptr;
        eá_n_y_r = eá_n_v_r_ptr[y];
        eá_i = eá_c_y_r - eá_n_y_r;
        eá_áx_r = (*eá_n_v_r_ptr++ - seg_x_r) / eá_i;
        seg_x_r += eá_áx_r * eá_y_r_skip;
        eá_áx_r *= ystep;
        eá_n_v_r_ptr++; /* skip y */
        eá_áz_r = (*eá_n_v_r_ptr++ - seg_z_r) / eá_i;
        seg_z_r += eá_áz_r * eá_y_r_skip;
        eá_áz_r *= ystep;
        eá_ásh_r = (*eá_n_v_r_ptr++ - seg_sh_r) / eá_i;
        seg_sh_r += eá_ásh_r * eá_y_r_skip;
        eá_ásh_r *= ystep;
        eá_n_v_r_ptr = nextr(eá_n_v_r_ptr);
        } while ((seg_y < eá_n_y_r) && (eá_n_y_r < eá_c_y_r));

seg_i = max(eá_n_y_l, eá_n_y_r);
if (eá_x_l_skip != 0)
        {
        pi_z = seg_z_l = seg_z_l + (eá_x_l_skip
                        * ((seg_z_r - seg_z_l) / (seg_x_r - seg_x_l)));
        pi_sh = seg_sh_l = seg_sh_l + (eá_x_l_skip
                        *((seg_sh_r - seg_sh_l) / (seg_x_r - seg_x_l)));
        eá_x_l_skip = 0;
        }
```

segmentbody1.c

```
/* This is the segment (or scanline) section.
 * It sets everything up so that the pixel code can march
 * along the current segment (or scanline, if you prefer).
 * This involves positioning pointers into the z and image
 * buffers for the first and last pixels to be considered,
 * setting up the z and sh values and their delta values
 * (i.e. z and sh's increments).
 */

do      {
        pi_z_s_ptr = seg_zr_ptr[0];
        pi_z_s_ptr = pi_z_b_ptr = &pi_z_s_ptr[seg_x_l/xstep];
        pi_im_ptr = seg_ir_ptr[0];
        pi_im_ptr = &pi_im_ptr[seg_x_l/xstep];
        seg_j = (seg_x_r - seg_x_l);
        pi_dz = ((seg_z_r - seg_z_l) / seg_j) * xstep;
        pi_dsh = ((seg_sh_r - seg_sh_l) / seg_j) * xstep;
        pi_z_s_ptr += seg_j / xstep;

        /* inner loop   | */
        /*              V */
```

pixelbody.c

```
/* This section of code paints the pixels (if appropriate)
 * across the current scan line.
 */

for ( ;pi_z_b_ptr <= pi_z_s_ptr; pi_z_b_ptr++)
        {
        if (pi_z < *pi_z_b_ptr)
                {
                *pi_z_b_ptr = pi_z;
                *pi_im_ptr = pi_sh;
                }
        pi_im_ptr++;
        pi_sh += pi_dsh;
        pi_z += pi_dz;
        }
```

segmentbody2.c

```
/*              ^ */
/* inner loop | */

pi_x_l = seg_x_l += ed_dx_l;
seg_x_r += ed_dx_r;
pi_z = seg_z_l += ed_dz_l;
seg_z_r += ed_dz_r;
pi_sh = seg_sh_l += ed_dsh_l;
seg_sh_r += ed_dsh_r;
seg_zr_ptr--;
seg_ir_ptr--;
seg_y -= ystep;
} while (seg_y > seg_i);
```

eágebody2.c

polybocy2.c

```
}
```

Appendix B

PROGRAM LISTINGS IN PDP-11 ASSEMBLER

This appendix contains the PDP/11 assembler code which implements the Z-buffer algorithm described in Chapter 2. This code was generated by the C compiler on a Version 7 UNIX[6] system, and then modified by hand to improve its execution efficiency. Beside each statement, a pair of numbers appears. The first number refers to the number of memory cycles required to fetch the instruction (assuming 16 bit fetches). The second number refers to the number of memory cycles required to fetch (or store) the instruction's data. For example, consider

```
MOV    *R1,R2     /2     1.
```

To execute this instruction, two 16-bit words of instruction must be fetched, and one 16-bit data word must be fetched.

Multiply and divide instructions are marked with "M" and "D", respectively. Both were assumed to require 10 memory cycles to fetch their instruction and data and to execute.

Commentary beside the instructions will give the reader some guide to the decisions made when the analysis was not straightforward. For example, the statistical data do not distinguish between left and right side vertices. But left and right side vertices do require different amounts of time to process because of a polygon's representation in memory. In this particular case, we assumed that left and right side vertices were equally probable, and so a simple average of the execution times was sufficient.

Each major section of code is labeled POLY, VERTEX, SEG. or PIX. Blocks marked POLY are executed on a per polygon basis. Blocks marked VERTEX are executed for each vertix (or edge). Simarily, SEG. refers to segment (or scan line) processing, and PIX. refers to pixel processing.

---

[6] UNIX is a trademark of Bell Telephone Laboratories.

B.1 CODE FROM POLYBODY1.C AND POLYBODY2.C

The following section of assembler code is the result of the POLYBODY1.C and POLYBODY2.C from Appendix A. No other modules are considered.

poly.s.as.ana

```
L8:mov po_n_ver(r5), r0          /2     1        poly setup      POLY.
ash $3, r0                       /2     0
add r5, r0                       /1     0
add $poly, r0                    /2     0
mov r0, ed_mx_v_(r5)             /2     1

mov r0, ed_c_v_l(r5)             /2     1

mov r0, ed_c_v_r(r5)             /2     1

mov ed_mx_v_(r5), r4             /2     1
mov 2(r4), r0                    /2     1
mov r0, po_y(r5)                 /2     1

mov r0, po_min_y(r5)             /2     1

mov (r4), r0                     /1     1
mov r0, po_x_r(r5)               /2     1

mov r0, po_x_l(r5)               /2     1

add $-vertex_size, r4            /2     0
mov r5, r0                       /1     0
add $poly, r0                    /2     0

L11:
                                            scan vertices   VERTEX.
cmp r4, r0                       /1     0  <<<<|use 5
jlo     L12                      /1     0      |
mov 2(r4), r2                    /2     1  <<<<|

cmp po_y(r5), r2                 /2     1      if
jge     L13                      /1     0

mov r4, ed_c_v_l(r5)             /2     1  <<<<|probability of finding a
                                                |new 'highest' vertex is
mov r4, ed_c_v_r(r5)             /2     1      |assumed to be .25.
                                                |
mov r2, po_y(r5)                 /2     1      |
                                                |
mov *ed_c_v_l(r5), r0            /2     2      |
mov r0, po_x_r(r5)               /2     1      |
                                                |
mov r0, po_x_l(r5)               /2     1      |
jbr     L14                      /1     0  <<<<|

L13:cmp po_y(r5), r2             /2     1  <<<<| else
jne     L15                      /1     0  <<<<|execution probability = .25.

cmp po_x_l(r5), *r4              /2     2  <<<<|if new leftmost vertex
jle     L16                      /1     0  <<<<|execution probability = .1.

mov r4, ed_c_v_l(r5)             /2     1  <<<<|process new leftmost vertex
mov *r4, po_x_l(r5)              /2     2      |execution probability = .0
jbr     L17                      /1     0  <<<<|
```

poly.s.as.ana

```
L16:cmp po_x_r(r5), *r4          /2      2   <<<<|if new rightmost vertex
jge     L18                      /1      0   <<<<|execution probability = .1

mov r4, ed_c_v_r(r5)             /2      1   <<<<|process new rightmost vertex
mov *r4, po_x_r(r5)              /2      2       |execution probability = .1
L18:L17:jbr     L19              /1      0   <<<<|

L15:cmp r2, po_min_y(r5)         /2      1   <<<<|if found new lowest vertex
jge     L20                      /1      0       |execution probability = .5
mov r2, po_min_y(r5)             /2      1   <<<<|

L20:L19:L14:
sub $(vertex_size*2), r4         /2      0
jbr     L11                      /1      0

L12:mov $201, r0                 /2      0       init values     POLY.
mov r0, ed_n_y_r(r5)             /2      1

mov r0, ed_n_y_l(r5)             /2      1

mov r5, r2                       /1      0
add $-254, r2                    /2      0
cmp ed_c_v_r(r5), r2             /2      1
jhi     L10000                   /1      0
mov ed_mx_v_(r5), r0             /2      1
jbr     L10001                   /1      0
L10000:mov ed_c_v_r(r5), r0      /2      1
add $-(vertex_size*2), r0        /2      0
L10001:mov r0, ed_n_v_r(r5)      /2      1

cmp ed_mx_v_(r5), ed_c_v_l(r5)   /3      2

jhi     L10002                   /1      0
jbr     L10003                   /1      0
L10002:mov ed_c_v_l(r5), r0      /2      1
add $(vertex_size*2)10, r0       /2      0
L10003:mov r0, ed_n_v_l(r5)      /2      1

mov 2(r0), seg_y(r5)             /3      2

mov $xstart, r0                  /2      0
mov $xstep, r2                   /2      0
neg     r2                       /1      0
mov seg_y(r5), r3                /2      1
bic r2, r3                       /1      0
sub r3, r0                       /1      0
jge     L10004                   /1      0
add $4, r0                       /2      0
L10004:
L10005:mov r0, r4                /1      0

jeq     L21                      /1      0
mov $ystep, r0                   /2      0
sub r4, r0                       /1      0
sub r0, seg_y(r5)                /2      1
```

poly.s.as.ana

```
L21:mov seg_y(r5), r1          /2      1
sxt     r0                     /1      0
div $ystep, r0                 /D
ash $6, r0                     /2      0
add r5, r0                     /1      0
mov r0, r4                     /1      0
add $z_buf, r0                 /2      0
mov r0, seg_zr_p(r5)           /2      1

add $image, r4                 /2      0
mov r4, seg_ir_p(r5)           /2      1

jbr     L4                     /1      0
L5:L3:
```

B.2 CODE FROM EDGEBODY1.C AND EDGEBODY2.C

The following section of assembler code is the result of the EDGEBODY1.C and EDGEBODY2.C from Appendix A. No other modules are considered.

edge.s.as.ana

```
L4:
mov seg_y(r5), r3                /2      1       loop control    VERTEX.
cmp po_min_y(r5), r3             /2      1
jgt     L5                       /1      0

cmp ed_n_y_l(r5), r3             /2      1       if (lhs)
jlt     L6                       /3      0

mov ed_c_v_l(r5), r4             /2      1       then section    LHS
L9:mov (r4)+, seg_x_l(r5)        /3      2

mov (r4)+, ed_c_y_l(r5)          /3      2
mov ed_c_y_l(r5), r0             /2      1
sub r3, r0                       /1      0
mov r0, ed_y_l_s(r5)             /2      1

mov (r4)+, r0                    /2      1
mov r0, seg_z_l(r5)              /2      1

mov r0, pi_z(r5)                 /2      1

mov (r4)+, r0                    /2      1
mov r0, seg_sh_l(r5)             /2      1

mov r0, pi_sh(r5)                /2      1

mov ed_n_v_l(r5), r4             /2      1
mov r4, ed_c_v_l(r5)             /2      1

mov 2(r4), ed_n_y_l(r5)          /3      2

mov ed_c_y_l(r5), r2             /2      1
sub ed_n_y_l(r5), r2             /2      1
mov r2, ed_i(r5)                 /2      1

mov (r4)+, r1                    /2      1
sub seg_x_l(r5), r1              /2      1
sxt     r0                       /1      0
div r2, r0                       /D
mov r0, ed_dx_l(r5)              /2      1

mov r0, r1                       /1      0
mul ed_y_l_s(r5), r1             /M
add r1, seg_x_l(r5)              /2      1

mov $xstart, r0                  /2      0
mov $xstep, r2                   /2      0
neg     r2                       /1      0
mov seg_x_l(r5), r3              /2      1
bic r2, r3                       /1      0
sub r3, r0                       /1      0
jge     L10000                   /1      0
add $xstep, r0                   /2      0
L10000:
L10001:mov r0, ed_x_l_s(r5)      /2      1
```

edge.s.as.ana

```
add r0, seg_x_l(r5)              /2      1

mov ed_dx_l(r5), r0              /2      1
ash $xstep, r0                   /2      0
mov r0, ed_dx_l(r5)              /2      1

mov (r4)+, r1                    /2      1
sub seg_z_l(r5), r1              /2      1
sxt     r0                       /1      0
div ed_i(r5), r0                 /D
mov r0, ed_dz_l(r5)              /2      1

mov r0, r1                       /1      0
mul ed_y_l_s(r5), r1             /M
add r1, seg_z_l(r5)              /2      1

mov seg_z_l(r5), pi_z(r5)        /3      2

mov ed_dz_l(r5), r0              /2      1
ash $ystep, r0                   /2      0
mov r0, ed_dz_l(r5)              /2      1

mov (r4)+, r1                    /2      1
sub seg_sh_l(r5), r1             /2      1
sxt     r0                       /1      0
div ed_i(r5), r0                 /D
mov r0, ed_dsh_l(r5)             /2      1

mov r0, r1                       /1      0
mul ed_y_l_s(r5), r1             /M
add r1, seg_sh_l(r5)             /2      1

mov r1, pi_sh(r5)                /2      1

mov ed_dsh_l(r5), r0             /2      1
ash $ystep, r0                   /2      0
mov r0, ed_dsh_l(r5)             /2      1

cmp ed_mx_v_(r5), r4             /2      1

jhis    L10002                   /1      0
mov r5, r0                       /1      0   <<<<|assume average time
add $poly, r0                    /2      0       |through this section
jbr     L10003                   /1      0       |is 5 memory cycles
L10002:mov r4, r0                /1      0       |
L10003:mov r0, ed_n_v_l(r5)      /2      1   <<<<|

L7:
mov seg_y(r5), r3                /2      1
cmp ed_n_y_l(r5), r3             /2      1

jle     L10004                   /1      0
cmp ed_c_y_l(r5), ed_n_y_l(r5)   /3      2   <<<<|ignore looping,
jgt     L9                       /3      0   <<<<|use 7 cycles
                                                 end of lhs
```

edge.s.as.ana

```
L10004:L8:L6:
cmp ed_n_y_r(r5), r3            /2      1       if (rhs)        VERTEX

jlt     L10                     /3      0
L13:
mov ed_c_v_r(r5), r4            /2      1       then section    RHS
mov r4, seg_x_r(r5)             /2      1

mov (r4)+, r0                   /2      1
mov r0, ed_c_y_r(r5)            /2      1
sub r3, r0                      /1      0
mov r0, ed_y_r_s(r5)            /2      1

mov (r4)+, seg_z_r(r5)          /3      2

mov (r4)+, seg_sh_r(r5)         /3      2

mov ed_n_v_r(r5), r4            /2      1
mov r4, ed_c_v_r(r5)            /2      1

mov 2(r4), ed_n_y_r(r5)         /3      2

mov ed_c_y_r(r5), r2            /2      1
sub ed_n_y_r(r5), r2            /2      1
mov r2, ed_i(r5)                /2      1

mov (r4)+, r1                   /2      1
sub seg_x_r(r5), r1             /2      1
sxt     r0                      /1      0
div r2, r0                      /D
mov r0, ed_dx_r(r5)             /2      1

mov r0, r1                      /1      0
mul ed_y_r_s(r5), r1            /M
add r1, seg_x_r(r5)             /2      1

L14:mov ed_dx_r(r5), r0         /2      1
ash $ystep, r0                  /2      0
mov r0, ed_dx_r(r5)             /2      1

tst     (r4)+                   /2      1
mov (r4)+, r1                   /2      1
sub seg_z_r(r5), r1             /2      1
sxt     r0                      /1      0
div ed_i(r5), r0                /D
mov r0, ed_dz_r(r5)             /2      1

mov r0, r1                      /1      0
mul ed_y_r_s(r5), r1            /M
add r1, seg_z_r(r5)             /2      1

mov ed_dz_r(r5), r0             /2      1
ash $ystep, r0                  /2      0
mov r0, ed_dz_r(r5)             /2      1

mov (r4)+, r1                   /2      1
```

edge.s.as.ana

```
sub seg_sh_r(r5), r1            /2      1
sxt     r0                      /1      0
div ed_i(r5), r0                /D
mov r0, ed_dsh_r(r5)            /2      1

mov r0, r1                      /1      0
mul ed_y_r_s(r5), r1            /M
add r1, seg_sh_r(r5)            /2      1

mov ed_dsh_r(r5), r0            /2      1
ash $ystep, r0                  /2      0
mov r0, ed_dsh_r(r5)            /2      1

mov r5, r0                      /1      0
add $-244, r0                   /2      0
cmp r4, r0                      /1      0
jhi     L10007                  /1      0
mov ed_mx_v_(r5), r0            /2      1   <<<<|this section estimated to
jbr     L10006                  /1      0       |take 6.5 memory cycles,
L10007:mov r4, r0               /1      0       |average.
add $-(vertex_size*2), r0       /2      0       |
L10008:mov r0, ed_n_v_r(r5)     /2      1   <<<<|

L11:cmp ed_n_y_r(r5), seg_y(r5) /3      2

jle     L10009                  /1      0
cmp ed_c_y_r(r5), ed_n_y_r(r5)  /3      2

jgt     L13                     /3      0   <<<<|ignore looping, use 1
                                                end rhs

L10009:L12:L10:
cmp ed_n_y_r(r5), ed_n_y_l(r5)  /3      2       seg_i = ...     VERTEX
jle     L10010                  /1      0
mov ed_n_y_r(r5), r0            /2      1   <<<<|this section estimated
jbr     L10011                  /1      0       |to take 6.5 memory
L10010:mov ed_n_y_l(r5), r0     /2      1       |cycles, average.
L10011:mov r0, seg_i(r5)        /2      1   <<<<|

tst     ed_x_l_s(r5)            /2      1       if
jeq     L15                     /1      0
mov seg_z_r(r5), r1             /2      1   <<<<|then section
sub seg_z_l(r5), r1             /2      1       |(probability of
sxt     r0                      /1      0       | entering this section
mov seg_x_r(r5), r2             /2      1       | is assumed to be .5;
sub seg_x_l(r5), r2             /2      1       | time required, 21.5
div r2, r0                      /D              | memory cycles plus
mov r0, r1                      /1      0       | one multiply and one
mul ed_x_l_s(r5), r1            /M              | divide, average.)
add seg_z_l(r5), r1             /2      1       |
mov r1, seg_z_l(r5)             /2      1       |
                                                |
mov r1, pi_z(r5)                /2      1       |
                                                |
mov seg_sh_r(r5), r1            /2      1       |
sub seg_sh_l(r5), r1            /2      1       |
```

edge.s.as.ana

```
sxt     r0                      /1      0       |
div r2, r0                      /D              |
mov r0, r1                      /1      0       |
mul ed_x_1_s(r5), r1            /M              |
add seg_sh_1(r5), r1            /2      1       |
mov r1, seg_sh_1(r5)            /2      1       |
                                                |
mov r1, pi_sh(r5)               /2      1       |
                                                |
clr     ed_x_1_s(r5)            /2      1   <<<<|end if
L15:
jbr     L4                      /2      0
L5:L3:
```

B.3 CODE FROM SEGMENTBODY1.C AND SEGMENTBODY2.C

The following section of assembler code is the result of the SEGMENTBODY1.C and SEGMENTBODY2.C from Appendix A. No other modules are considered.

segment.s.as.ana

```
L6:
mov seg_x_l(r5), r4              /2      1       pointers        SEG.
mov r4, r1                       /1      0
sxt     r0                       /1      0
div $(xstep/2), r0               /D
mov r0, r2                       /1      0
add seg_zr_ptr(r5), r0           /2      1
mov r0, pi_z_b_p(r5)             /2      1
mov r0, pi_z_s_p(r5)             /2      1

mov r2, r0                       /1      0
add seg_ir_ptr(r5), r0           /2      1
mov r0, pi_im_pt(r5)             /2      1

mov seg_x_r(r5), r3              /2      1       delta values
sub r4, r3                       /1      0

mov seg_z_r(r5), r1              /2      1
sub seg_z_l(r5), r1              /2      1
sxt     r0                       /1      0
div r3, r0                       /D
ash $2, r0                       /2      0
mov r0, pi_dz(r5)                /2      1

mov seg_sh_r(r5), r1             /2      1
sub seg_sh_l(r5), r1             /2      1
sxt     r0                       /1      0
div r3, r0                       /D
ash $2, r0                       /2      0
mov r0, pi_dsh(r5)               /2      1

mov r3, r1                       /1      0       more with pointers
sxt     r0                       /1      0
div $(xstep/2), r0               /D
add r0, pi_z_s_p(r5)             /2      1

/
/inner loop, here
/

add ed_dx_l(r5), seg_x_l(r5)     /3      2       calc. next row values
add ed_dx_r(r5), seg_x_r(r5)     /3      2
add ed_dz_l(r5), seg_z_l(r5)     /3      2
mov seg_z_l(r5), pi_z(r5)        /3      2
add ed_dz_r(r5), seg_z_r(r5)     /3      2
add ed_dsh_l(r5), seg_sh_l(r5)   /3      2
mov seg_sh_l(r5), pi_sh(r5)      /3      2
add ed_dsh_r(r5), seg_sh_r(r5)   /3      2
sub $100, seg_zr_p(r5)           /3      1
sub $100, seg_ir_p(r5)           /3      1
sub $ystep, seg_y(r5)            /3      1

L4:cmp seg_i(r5), seg_y(r5)      /3      2       loop control
jlt     L6                       /1      0
L5:L3:
```

B.4 CODE FROM PIXELBODY.C

The following section of assembler code is the result of the PIXELBODY.C from Appendix A. No other modules are considered.

pixel.s.as.ana

```
mov pi_z_s_p(r5), r4     /2      1       setup pointers  SEG.
mov pi_z_b_p(r5), r3     /2      1
mov pi_im_pt(r5), r2     /2      1
mov pi_z(r5), r1         /2      1
mov pi_sh(r5), r0        /2      1

L4:
cmp r4, r3               /1      0       loop control    PIX.
jlo     L5               /1      0
cmp r1, *r3              /1      1       test z buffer
jge     L7               /1      0
mov r1, *r3              /1      1       replace
mov r0, *r2              /1      1

L7:
add pi_ash(r5), r0       /2      1       update values
add pi_az(r5), r1        /2      1

L6:cmp (r2)+, (r3)+      /3      2       update pointers

jbr     L4               /1      0       loop control
L5:L3:
```

Appendix C

STATISTICAL CHARACTERISTICS OF SELECTED SCENES

We will now give a few samples of the statistics collected on the scenes. For each scene, statistics were collected for each processor in a uniprocessor, 4- and 16-processor splitter and interlace schemes, and 16-processor hybrid schemes. Thus, statistics were collected for 57 processors per scene.

In the statistics, the following abbreviations are used:

1. v data: number of vertices per polygon

2. y data: polygon height in scan lines assigned to this processor (i.e. the number of segments processed for this polygon)

3. x data: not used

4. l data: segment length in pixels assigned to this processor

5. D data: depth complexity per pixel

6. area stats: number of pixels covered by each polygon

7. vertex time: average number of vertices per polygon * Vt

8. segment time: the average number of segments per polygon * St

9. pixel time: the average number of pixels per polygon * Pt

10. poly overhead: Gt

11. avg poly time: vertex time + edge (i.e. segment) time + pixel time + poly overhead

All of the above times refer to memory cycles.

The notation

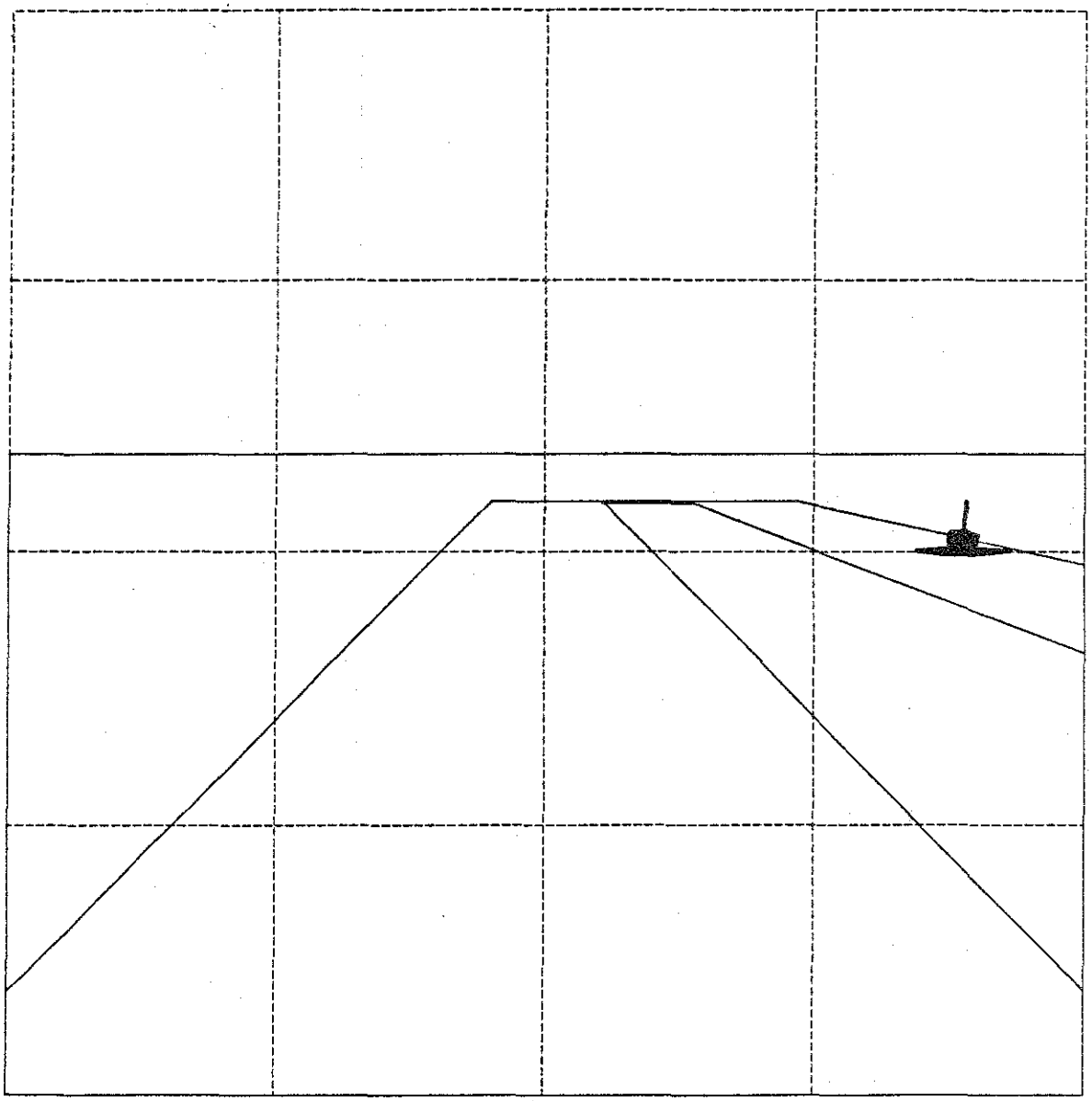
[n] = m

indicates that m data points had value n. Thus, for vertex data, [3] = 10 means that 10 triangles (a 3 vertex polygon) were processed by this processor for this scene.

We have arbitrarily chosen figure 15 to use as an example. Below are the statistics from a few processors working on that scene. Each chosen processor will be identified, and then its statistics listed.

C.1 UNIPROCESSOR

The following statistics are from a uniprocessor working on figure 15.



106.0.list

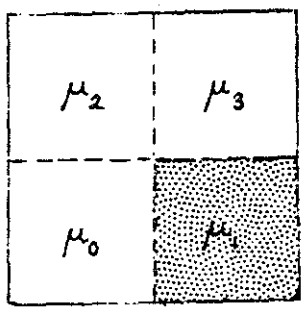
```
v data
248 data points
avg=2.947581,   std.dev=0.696531,       var=0.485155
distribution
        [1]=2   [2]=54  [3]=153         [4]=34  [5]=4   [6]=1


y data
248 data points
avg=5.689516,   std.dev=26.242748,      var=688.681824
distribution
        [1]=31  [2]=95  [3]=68  [4]=23  [5]=7   [6]=11  [11]=2  [14]=4
 [15]=4  [73]=1  [280]=1         [302]=1

 no x stats


l data
1411 data points
avg=178.046768, std.dev=222.982391,     var=49721.144531
distribution
        [1]=462         [2]=155         [3]=64  [4]=28  [5]=23  [6]=1
  [7]=4   [8]=2   [9]=1   [10]=3  [11]=6  [12]=6  [13]=4  [15]=1 [17]=1
[20]=1 [22]=1 [25]=1 [28]=1 [30]=1 [33]=1 [35]=1 [38]=1 [41]=1 [42]=1
[43]=1 [44]=1 [46]=1 [48]=1 [51]=1 [53]=2 [54]=2 [55]=1 [56]=2 [57]=1
[58]=1 [59]=2 [60]=1 [61]=1 [62]=3 [64]=2 [65]=1 [66]=2 [67]=1 [68]=1
[69]=2 [70]=2 [72]=3 [74]=2 [75]=1 [76]=2 [77]=1 [78]=2 [80]=3 [81]=1
[82]=2 [84]=2 [85]=1 [86]=2 [87]=1 [88]=2 [90]=3 [91]=1 [92]=1 [93]=1
[94]=2 [95]=2 [96]=1 [97]=1 [98]=2 [99]=1 [100]=1 [101]=2 [102]=1 [103]=2
[104]=1 [105]=1 [106]=3 [108]=1 [109]=2 [110]=1 [111]=1 [112]=1 [114]=1
[116]=1 [118]=1 [120]=1 [122]=1 [124]=1 [126]=1 [128]=1 [130]=1 [132]=1
[134]=1 [136]=1 [138]=1 [140]=1 [142]=1 [144]=1 [146]=1 [148]=1 [150]=1
[152]=1 [154]=1 [156]=1 [158]=1 [160]=1 [162]=1 [164]=1 [166]=1 [168]=1
[170]=1 [172]=1 [174]=1 [176]=1 [178]=1 [180]=1 [182]=1 [184]=1 [186]=1
[188]=1 [190]=1 [192]=1 [194]=1 [196]=1 [198]=1 [200]=1 [202]=1 [204]=1
[206]=1 [208]=1 [210]=1 [212]=1 [214]=1 [216]=1 [218]=1 [220]=1 [222]=1
[224]=1 [226]=1 [228]=1 [230]=1 [232]=1 [234]=1 [236]=1 [238]=1 [240]=1
[242]=1 [244]=1 [246]=1 [248]=1 [250]=1 [252]=1 [254]=1 [256]=1 [258]=1
[260]=1 [262]=1 [264]=1 [266]=1 [268]=1 [270]=1 [272]=1 [274]=1 [276]=1
[278]=1 [280]=1 [282]=1 [284]=1 [286]=1 [288]=1 [290]=1 [292]=1 [294]=1
[296]=1 [298]=1 [300]=1 [302]=1 [304]=1 [306]=1 [308]=1 [310]=1 [312]=1
[314]=1 [316]=1 [318]=1 [320]=1 [322]=1         [324]=1         [326]=1
[328]=1 [330]=1 [332]=1 [334]=1 [336]=1 [338]=1 [340]=1 [342]=1 [344]=1
[346]=1 [348]=1 [350]=1 [352]=1 [354]=1 [356]=1 [358]=1 [360]=1 [362]=1
[364]=1 [366]=1 [368]=1 [370]=1 [372]=1 [374]=1 [376]=1 [378]=1 [380]=1
[382]=1 [384]=1 [386]=1 [388]=1 [390]=1 [392]=1 [394]=1 [396]=1 [398]=1
[400]=1 [402]=1 [404]=1 [406]=1 [408]=1 [410]=1 [412]=1 [414]=1 [416]=1
```

106.0.list

```
[418]=1 [420]=1 [422]=1 [424]=1 [426]=1 [428]=1 [430]=1 [432]=1 [434]=1
[436]=1 [438]=1 [440]=1 [442]=1 [444]=1 [446]=1 [448]=1 [450]=1 [452]=1
[454]=1 [456]=1 [458]=1 [460]=1 [462]=1 [464]=1 [466]=1 [468]=1 [470]=1
[472]=1 [474]=1 [476]=1 [478]=1 [480]=1 [482]=1 [484]=1 [486]=1 [488]=1
[490]=1 [492]=1 [494]=1 [496]=1 [498]=1 [500]=1 [502]=1 [504]=1 [506]=1
[508]=1 [510]=1 [512]=352


D data
262144 data points
avg=0.958344,   std.dev=0.902131,       var=0.813840
distribution
        [0]=107520      [1]=59386       [2]=94945       [3]=35  [4]=42
 [5]=31  [6]=68  [7]=22  [8]=21  [9]=24  [10]=22         [11]=10 [12]=8
[13]=4 [14]=2 [16]=2 [17]=1 [19]=1


area stats
number of points 248
mean=1013.000793
variance=128303424.000000,     std.dev.=11327.110352


vertex time 941.162476 (4.047641%): segment time 898.943542 (3.866071%):
pixel time 21273.017578 (91.488503%): poly overhead 139.000000 (0.597795%)

avg poly time 23252.123047: total scene time 5766526.500000


 end of stats
```

C.2 SPLITTER--4 PROCESSOR (SLOWEST)

The following statistics are from the lower right processor of a 4 processor splitter machine (micro number 1).



106.p210.list

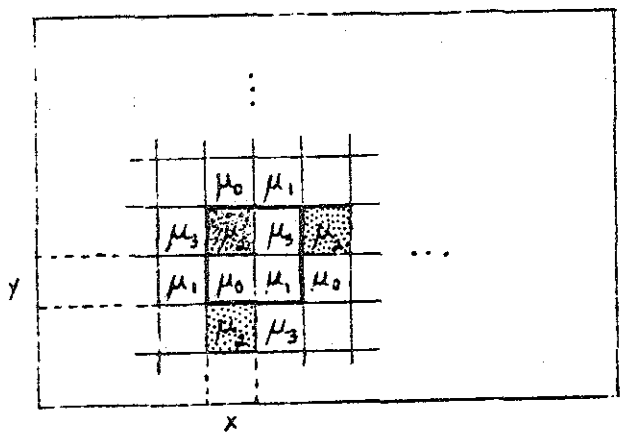
```
v data
72 data points
avg=2.263889,   std.dev=0.985915,       var=0.972029
distribution
        [1]=21  [2]=17  [3]=29  [4]=4   [5]=1


y data
72 data points
avg=9.277778,   std.dev=42.077713,      var=1770.534058
distribution
        [1]=41  [2]=18  [3]=10  [49]=1  [256]=2


 no x stats


l data
668 data points
avg=169.362274, std.dev=99.872993,      var=9974.614258
distribution
        [1]=58  [2]=7   [3]=9   [4]=14  [5]=7   [6]=1   [7]=2   [8]=2
  [9]=1   [10]=3  [11]=4  [12]=2  [13]=2  [15]=1  [17]=1  [20]=1 [22]=1
[25]=1 [28]=1 [30]=1 [33]=1 [36]=1 [38]=1 [41]=1 [43]=1 [46]=1 [49]=1
[50]=1 [51]=2 [52]=1 [53]=1 [54]=2 [55]=1 [56]=1 [57]=2 [58]=1 [59]=2
[60]=1 [61]=1 [62]=2 [63]=1 [64]=2 [65]=1 [66]=1 [67]=2 [68]=1 [69]=1
[70]=2 [71]=1 [72]=2 [73]=1 [74]=1 [75]=2 [76]=1 [77]=1 [78]=2 [79]=1
[80]=2 [81]=1 [82]=1 [83]=2 [84]=1 [85]=2 [86]=1 [87]=1 [88]=2 [89]=1
[90]=1 [91]=2 [92]=1 [93]=2 [94]=1 [95]=1 [96]=2 [97]=1 [98]=1 [99]=3
[100]=1 [101]=3 [102]=1 [103]=2 [104]=2 [105]=1 [106]=3 [107]=2 [108]=1
[109]=3 [110]=1 [111]=1 [112]=2 [113]=1 [114]=1 [115]=1 [116]=1 [117]=1
[118]=1 [119]=1 [120]=1 [121]=1 [122]=1 [123]=1 [124]=1 [125]=1 [126]=1
[127]=1 [128]=1 [129]=1 [130]=1 [131]=1 [132]=1 [133]=1 [134]=1 [135]=1
[136]=1 [137]=1 [138]=1 [139]=1 [140]=1 [141]=1 [142]=1 [143]=1 [144]=1
[145]=1 [146]=1 [147]=1 [148]=1 [149]=1 [150]=1 [151]=1 [152]=1 [153]=1
[154]=1 [155]=1 [156]=1 [157]=1 [158]=1 [159]=1 [160]=1 [161]=1 [162]=1
[163]=1 [164]=1 [165]=1 [166]=1 [167]=1 [168]=1 [169]=1 [170]=1 [171]=1
[172]=1 [173]=1 [174]=1 [175]=1 [176]=1 [177]=1 [178]=1 [179]=1 [180]=1
[181]=1 [182]=1 [183]=1 [184]=1 [185]=1 [186]=1 [187]=1 [188]=1 [189]=1
[190]=1 [191]=1 [192]=1 [193]=1 [194]=1 [195]=1 [196]=1 [197]=1 [198]=1
[199]=1 [200]=1 [201]=1 [202]=1 [203]=1 [204]=1 [205]=1 [206]=1 [207]=1
[208]=1 [209]=1 [210]=1 [211]=1     [212]=1         [213]=1         [214]=1
[215]=1 [216]=1 [217]=1 [218]=1 [219]=1 [220]=1 [221]=1 [222]=1 [223]=1
[224]=1 [225]=1 [226]=1 [227]=1 [228]=1 [229]=1 [230]=1 [231]=1 [232]=1
[233]=1 [234]=1 [235]=1 [236]=1 [237]=1 [238]=1 [239]=1 [240]=1 [241]=1
[242]=1 [243]=1 [244]=1 [245]=1 [246]=1 [247]=1 [248]=1 [249]=1 [250]=1
[251]=1 [252]=1 [253]=1 [254]=1 [255]=1 [256]=306
```

106.p210.list

```
D data
65536 data points
avg=1.726288,   std.dev=0.480864,       var=0.231231
distribution
        [1]=18264       [2]=47184       [3]=17  [4]=26  [5]=3   [6]=18
 [7]=4   [8]=4   [9]=5   [10]=5  [11]=3  [12]=3


area stats
number of points 72
mean=1571.305786
variance=84465440.000000,      std.dev.=9190.507813


vertex time 722.859680 (2.046302%): segment time 1465.888916 (4.149701%):
pixel time 32997.421875 (93.410507%): poly overhead 139.000000 (0.393487%)

avg poly time 35325.171875: total scene time 2543412.500000


 end of stats
```

C.3 INTERLACE--4 PROCESSOR (SLOWEST)

The following statistics are from the slowest processor in a 4-processor interlace configuration. Micro number 2 is the slowest.



106.f201.list

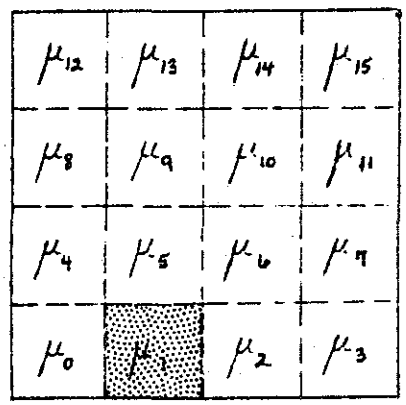
```
v data
248 data points
avg=2.947581,   std.dev=0.696531,       var=0.485155
distribution
        [1]=2   [2]=54  [3]=153         [4]=34  [5]=4   [6]=1


y data
248 data points
avg=2.919355,   std.dev=13.128150,      var=172.348328
distribution
        [0]=14  [1]=134         [2]=71  [3]=16  [6]=2   [7]=4   [8]=4
  [37]=1  [140]=1         [151]=1

 no x stats


l data
724 data points
avg=86.664368,  std.dev=110.906487,     var=12300.249023
distribution
        [0]=112         [1]=222         [2]=48  [3]=10  [4]=1   [5]=1
  [6]=2   [7]=2   [8]=1   [10]=1  [13]=1  [16]=1  [18]=1  [21]=2 [23]=1
[26]=2 [27]=1 [28]=2 [29]=1 [30]=2 [31]=1 [32]=2 [34]=3 [35]=1 [36]=1
[37]=2 [38]=1 [39]=2 [40]=1 [41]=1 [42]=2 [43]=1 [44]=2 [45]=1 [46]=1
[47]=2 [48]=1 [49]=1 [50]=2 [51]=1 [52]=3 [54]=1 [55]=1 [56]=1 [58]=1
[60]=1 [62]=1 [64]=1 [66]=1 [68]=1 [70]=1 [72]=1 [74]=1 [76]=1 [78]=1
[80]=1 [82]=1 [84]=1 [86]=1 [88]=1 [90]=1 [92]=1 [94]=1 [96]=1 [98]=1
[100]=1 [102]=1 [104]=1 [106]=1 [108]=1 [110]=1 [112]=1 [114]=1 [116]=1
[118]=1 [120]=1 [122]=1 [124]=1 [126]=1 [128]=1 [130]=1 [132]=1 [134]=1
[136]=1 [138]=1 [140]=1 [142]=1 [144]=1 [146]=1 [148]=1 [150]=1 [152]=1
[154]=1 [156]=1 [158]=1 [160]=1 [162]=1 [164]=1 [166]=1 [168]=1 [170]=1
[172]=1 [174]=1 [176]=1 [178]=1 [180]=1 [182]=1 [184]=1 [186]=1 [188]=1
[190]=1 [192]=1 [194]=1 [196]=1 [198]=1 [200]=1 [202]=1 [204]=1 [206]=1
[208]=1 [210]=1 [212]=1 [214]=1 [216]=1 [218]=1 [220]=1 [222]=1 [224]=1
[226]=1 [228]=1 [230]=1 [232]=1 [234]=1 [236]=1 [238]=1 [240]=1 [242]=1
[244]=1 [246]=1 [248]=1 [250]=1 [252]=1 [254]=1 [256]=176


D data
65536 data points
avg=0.957413,   std.dev=0.905534,       var=0.819993
distribution
        [0]=26880       [1]=14914       [2]=23671       [3]=11  [4]=15
 [5]=2   [6]=14  [7]=3   [8]=2   [9]=5   [10]=10         [11]=1 [12]=4
[13]=2 [16]=1 [19]=1
```

106.f201.list

```
area stats
number of points 248
mean=253.003983  variance=8008588.500000,       std.dev.=2829.945068

vertex time 941.162476 (13.712728%): segment time 470.177429 (6.850480%):
pixel time 5313.083496 (77.411568%): poly overhead 139.000000 (2.025229%)

avg poly time 6863.423340: total scene time 1702129.000000


 end of stats
```

C.4 SPLITTER--16 PROCESSOR (SLOWEST)

The following statistics are from the slowest processor of a 16-processor splitter configuration. Micro number 1 was the slowest.

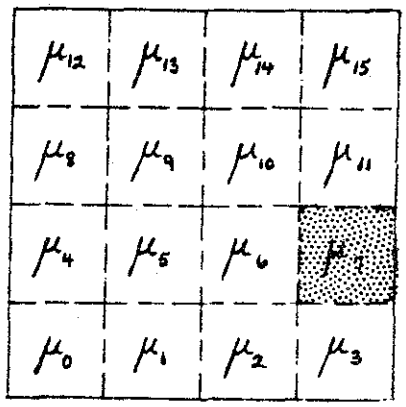


106.p410.list

```
v data
2 data points
avg=4.000000,   std.dev=0.000000,       var=0.000000
distribution
        [4]=2


y data
2 data points
avg=128.000000, std.dev=0.000000,       var=0.000000
distribution
        [128]=2

 no x stats


1 data
256 data points
avg=128.000000, std.dev=0.000000,       var=0.000000
distribution
        [128]=256


D data
16384 data points
avg=2.000000,   std.dev=0.000000,       var=0.000000
distribution
        [2]=16384

area stats
number of points 2
mean=16384.000000        variance=0.000000,     std.dev.=0.000000

vertex time 1277.199951 (0.349244%): segment time 20224.000000 (5.530153%):
pixel time 344064.000000 (94.082603%): poly overhead 139.000000 (0.038009%)

avg poly time 365704.187500: total scene time 731408.375000


 end of stats
```

C.5 SPLITTER--16 PROCESSOR (SHUTTLE PROCESSOR)

The following statistics are from micro number 7 in a 16-processor splitter configuration.



106.p431.list

```
v data
51 data points
avg=2.745098,   std.dev=0.588888,       var=0.346790
distribution
        [2]=17  [3]=30  [4]=4


y data
51 data points
avg=6.117647,   std.dev=19.533428,      var=381.554810
distribution
        [1]=20  [2]=18  [3]=10  [49]=2  [128]=1

 no x stats


l data
312 data points
avg=67.211540,  std.dev=55.997704,      var=3135.742676
distribution
        [1]=38  [2]=8   [3]=10  [4]=15  [5]=8   [6]=2   [7]=3   [8]=3
  [9]=2   [10]=4  [11]=5  [12]=3  [13]=3  [14]=1  [15]=2  [16]=1 [17]=2
[18]=1 [19]=1 [20]=2 [21]=1 [22]=2 [23]=1 [24]=1 [25]=2 [26]=1 [27]=1
[28]=2 [29]=1 [30]=2 [31]=1 [32]=1 [33]=2 [34]=1 [35]=1 [36]=2 [37]=1
[38]=2 [39]=1 [40]=1 [41]=2 [42]=1 [43]=2 [44]=1 [45]=1 [46]=2 [47]=1
[48]=1 [49]=2 [51]=1 [54]=1 [57]=1 [59]=1 [62]=1 [64]=1 [67]=1 [70]=1
[72]=1 [75]=1 [78]=1 [80]=1 [83]=1 [85]=1 [88]=1 [91]=1 [93]=1 [96]=1
[99]=2 [101]=2 [103]=1 [104]=1 [105]=1 [106]=1 [107]=1 [109]=2 [112]=1
[128]=128


D data
16384 data points
avg=1.279907,   std.dev=0.558281,       var=0.311678
distribution
        [1]=12103       [2]=4193        [3]=17  [4]=27  [5]=2   [6]=20
 [7]=5   [8]=5   [9]=6   [10]=4  [11]=1  [12]=1


area stats
number of points 51
mean=411.176514  variance=5306968.500000,       std.dev.=2303.685791


vertex time 876.509827 (8.255873%): segment time 966.588257 (9.104323%):
pixel time 8634.707031 (81.330566%): poly overhead 139.000000 (1.309245%)
```

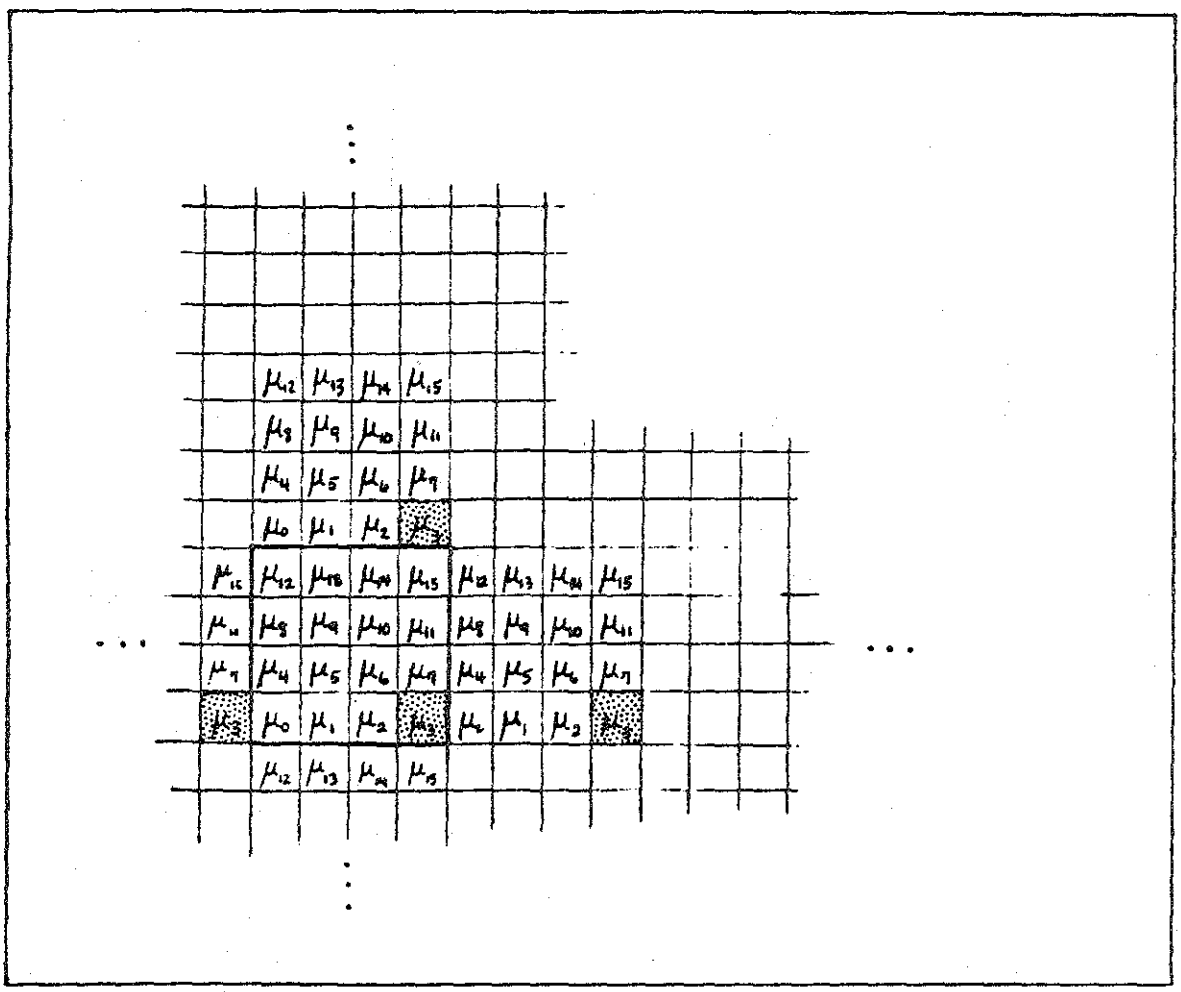
106.p431.list

avg poly time 10616.804688: total scene time 541457.062500

end of stats

C.6 INTERLACE--16 PROCESSOR (SLOWEST)

The following statistics are from the slowest micro in a 16-processor interlace configuration. Micro number 3 is the slowest.



106.f430.list

```
v data
248 data points
avg=2.947581,   std.dev=0.696531,       var=0.485155
distribution
        [1]=2   [2]=54  [3]=153         [4]=34  [5]=4   [6]=1


y data
248 data points
avg=1.439516,   std.dev=6.588447,       var=43.407635
distribution
        [0]=85  [1]=137         [2]=13  [3]=10  [18]=1  [70]=1  [76]=1

 no x stats



l data
357 data points
avg=44.439777,  std.dev=55.717445,      var=3104.433594
distribution
        [0]=84  [1]=100         [2]=2   [3]=6   [4]=3   [6]=1   [9]=1
  [12]=1  [14]=1  [15]=2  [16]=1  [17]=2  [18]=1  [19]=2  [20]=1 [21]=1
[22]=2 [23]=1 [24]=1 [25]=2 [26]=1 [27]=2 [29]=1 [31]=1 [33]=1 [35]=1
[37]=1 [39]=1 [41]=1 [43]=1 [45]=1 [47]=1 [49]=1 [51]=1 [53]=1 [55]=1
[57]=1 [59]=1 [61]=1 [63]=1 [65]=1 [67]=1 [69]=1 [71]=1 [73]=1 [75]=1
[77]=1 [79]=1 [81]=1 [83]=1 [85]=1 [87]=1 [89]=1 [91]=1 [93]=1 [95]=1
[97]=1 [99]=1 [101]=1 [103]=1 [105]=1 [107]=1 [109]=1 [111]=1 [113]=1
[115]=1 [117]=1 [119]=1 [121]=1 [123]=1 [125]=1 [127]=1 [128]=89


D data
16384 data points
avg=0.968323,   std.dev=0.915125,       var=0.837454
distribution
        [0]=6656        [1]=3714        [2]=5993        [5]=2   [6]=7
  [7]=2   [8]=1   [9]=3   [10]=4  [11]=1  [13]=1


area stats
number of points 248
mean=63.971771   variance=509367.187500,       std.dev.=713.699646


vertex time 941.162476 (34.791298%): segment time 281.596771 (10.40959%):
pixel time 1343.407227 (49.660797%): poly overhead 139.000000 (5.138316%)
```

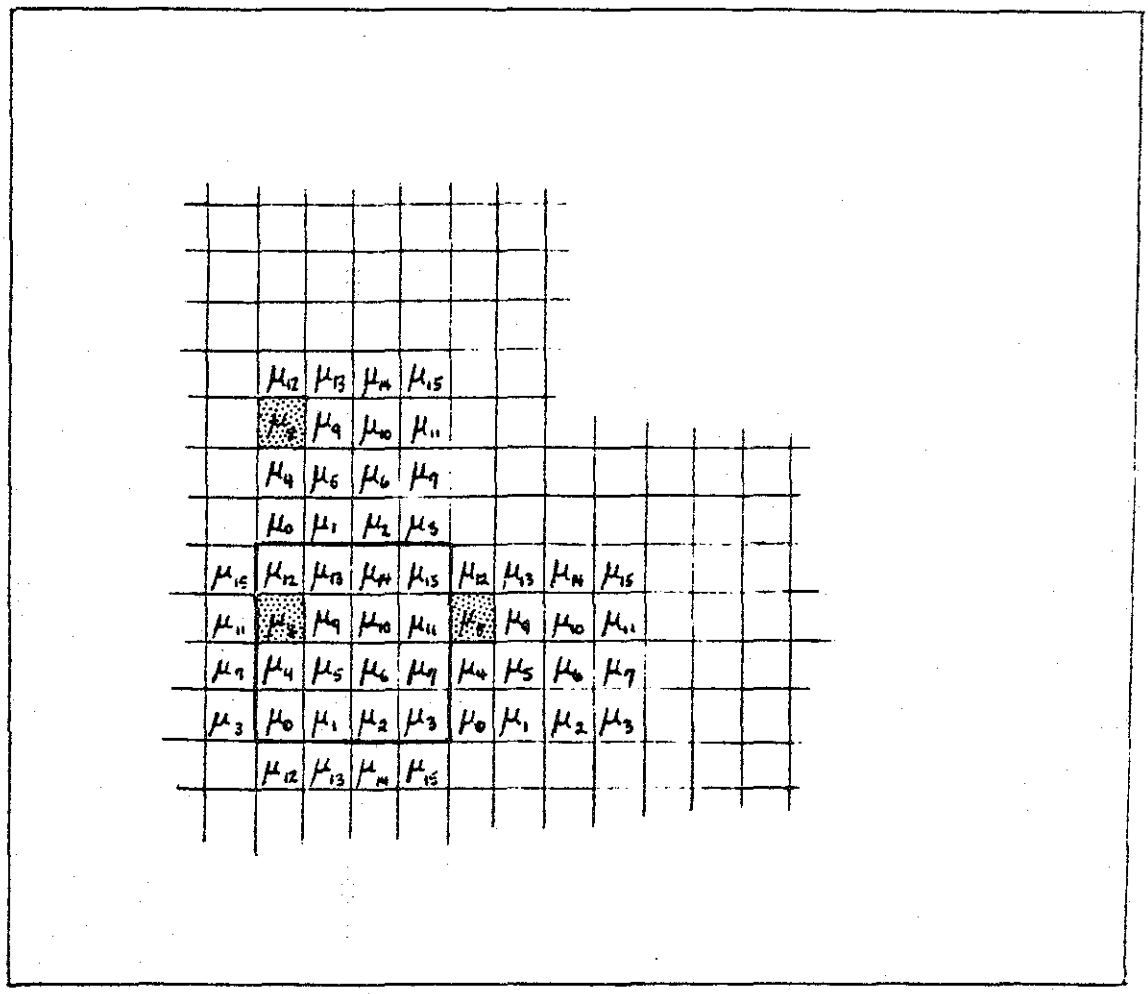
106.f430.list

avg poly time 2705.166504: total scene time 670881.312500

end of stats

C.7 INTERLACE--16 PROCESSOR (FASTEST)

The following statistics are from the fastest micro in a 16-processor interlace configuration. Micro number 8 is the fastest.



106.f402.list

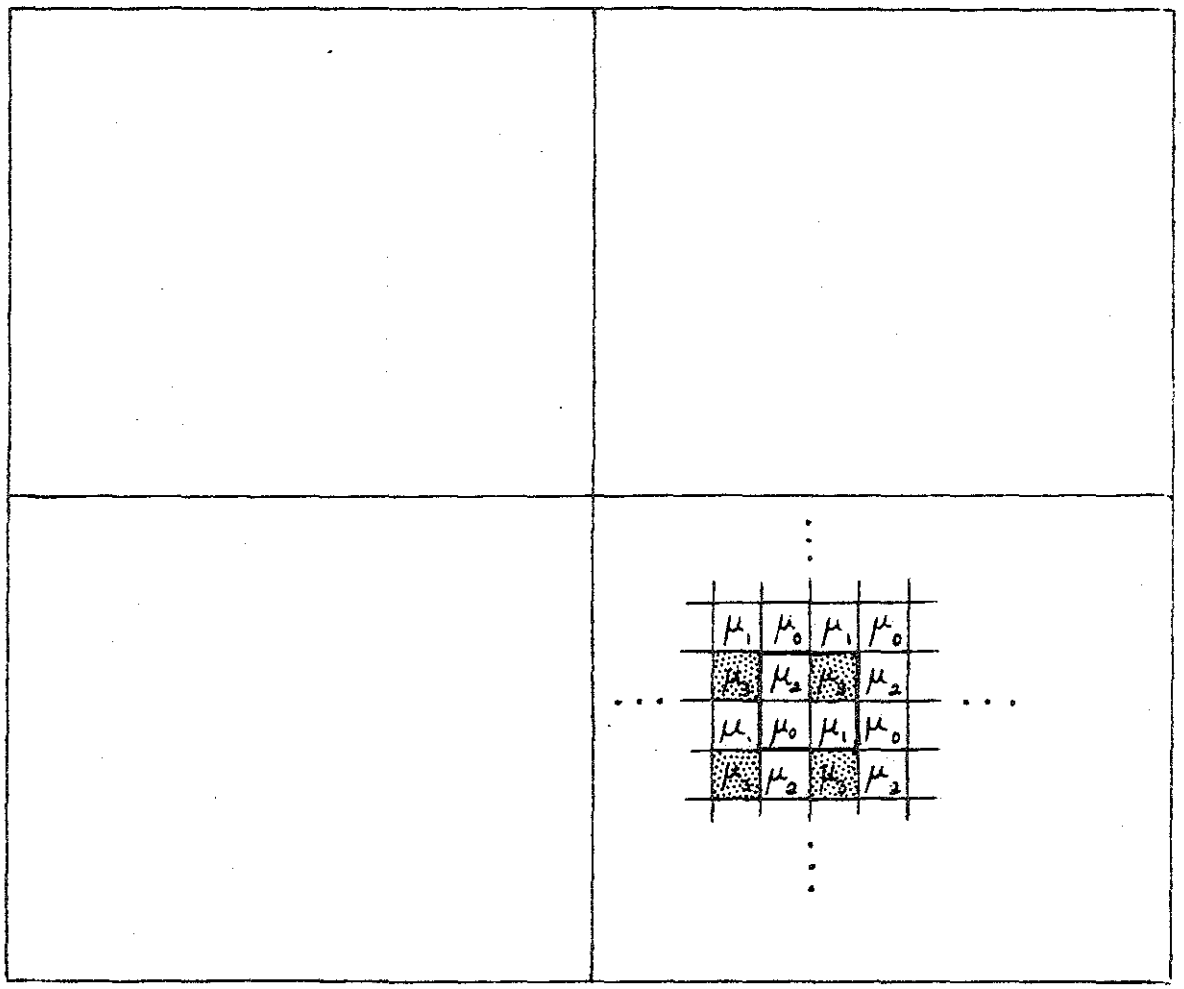
```
v data
248 data points
avg=2.947581,   std.dev=0.696531,       var=0.485155
distribution
        [1]=2   [2]=54  [3]=153         [4]=34  [5]=4   [6]=1


y data
248 data points
avg=1.330645,   std.dev=6.561400,       var=43.051971
distribution
        [0]=104         [1]=131         [2]=2   [4]=8   [18]=1  [70]=1
 [75]=1

 no x stats


l data
330 data points
avg=47.236362,  std.dev=56.242733,      var=3163.245117
distribution
        [0]=79  [1]=80  [2]=4   [3]=4   [4]=1   [6]=1   [9]=1   [11]=1
 [12]=1  [13]=2  [14]=2  [15]=1  [16]=1  [17]=2  [18]=1  [19]=2 [20]=1
[21]=1 [22]=2 [23]=1 [24]=1 [25]=2 [26]=1 [27]=1 [29]=1 [31]=1 [33]=1
[35]=1 [37]=1 [39]=1 [41]=1 [43]=1 [45]=1 [47]=1 [49]=1 [51]=1 [53]=1
[55]=1 [57]=1 [59]=1 [61]=1 [63]=1 [65]=1 [67]=1 [69]=1 [71]=1 [73]=1
[75]=1 [77]=1 [79]=1 [81]=1 [83]=1 [85]=1 [87]=1 [89]=1 [91]=1 [93]=1
[95]=1 [97]=1 [99]=1 [101]=1 [103]=1 [105]=1 [107]=1 [109]=1 [111]=1
[113]=1 [115]=1 [117]=1 [119]=1 [121]=1 [123]=1 [125]=1 [127]=1 [128]=87


D data
16384 data points
avg=0.951416,   std.dev=0.901809,       var=0.813259
distribution
        [0]=6784        [1]=3702        [2]=5878        [3]=1   [4]=4
  [5]=1   [6]=6   [7]=2   [8]=2   [10]=4


area stats
number of points 248
mean=62.854847   variance=494255.312500,        std.dev.=703.032959


vertex time 941.162476 (35.162426%): segment time 276.50000 (10.330215%):
pixel time 1319.951782 (49.314236%): poly. overhead 139.000000 (5.193128%)
```

106.f402.list

```
avg poly time 2676.614258: total scene time 663800.312500

end of stats
```

C.8 HYBRID--16 PROCESSOR (SLOWEST)

The following statistics are from the slowest micro in a 16-processor hybrid configuration. The slowest micro is micro number 3 in the lower right quadrant.



106.h1011.list

```
v data
72 data points
avg=2.263889,   std.dev=0.985915,       var=0.972029
distribution
        [1]=21  [2]=17  [3]=29  [4]=4   [5]=1


y data
72 data points
avg=4.972222,   std.dev=20.986088,      var=440.415894
distribution
        [0]=2   [1]=57  [2]=10  [25]=1  [128]=2


 no x stats



l data
358 data points
avg=79.036316,  std.dev=52.553192,      var=2761.837891
distribution
        [0]=23  [1]=31  [2]=12  [3]=8   [4]=1   [5]=2   [6]=3   [9]=1
  [11]=1  [14]=1  [17]=1  [19]=1  [22]=1  [25]=2  [26]=1  [27]=2 [28]=1
[29]=1 [30]=2 [31]=1 [32]=2 [33]=1 [34]=1 [35]=2 [36]=1 [37]=1 [38]=2
[39]=1 [40]=2 [41]=1 [42]=1 [43]=2 [44]=1 [45]=1 [46]=2 [47]=1 [48]=2
[49]=1 [50]=2 [51]=2 [52]=2 [53]=2 [54]=2 [55]=1 [56]=2 [57]=1 [58]=1
[59]=1 [60]=1 [61]=1 [62]=1 [63]=1 [64]=1 [65]=1 [66]=1 [67]=1 [68]=1
[69]=1 [70]=1 [71]=1 [72]=1 [73]=1 [74]=1 [75]=1 [76]=1 [77]=1 [78]=1
[79]=1 [80]=1 [81]=1 [82]=1 [83]=1 [84]=1 [85]=1 [86]=1 [87]=1 [88]=1
[89]=1 [90]=1 [91]=1 [92]=1 [93]=1 [94]=1 [95]=1 [96]=1 [97]=1 [98]=1
[99]=1 [100]=1 [101]=1 [102]=1 [103]=1 [104]=1 [105]=1 [106]=1 [107]=1
[108]=1 [109]=1 [110]=1 [111]=1 [112]=1 [113]=1 [114]=1 [115]=1 [116]=1
[117]=1 [118]=1 [119]=1 [120]=1 [121]=1 [122]=1 [123]=1 [124]=1 [125]=1
[126]=1 [127]=1 [128]=153


D data
16384 data points
avg=1.726990,   std.dev=0.487709,       var=0.237860
distribution
        [1]=4573        [2]=11782       [3]=6   [4]=11  [5]=2   [6]=2
  [7]=2   [8]=1   [9]=2   [11]=2  [12]=1


area stats
number of points 72
mean=392.986267  variance=5271475.000000,       std.dev.=2295.969238
```

106.hl011.list

```
vertex time 722.859680 (7.298244%): segment time 790.000000 (7.976115%):
pixel time 8252.711914 (83.322250%): poly overhead 139.000000 (1.403392%)

avg poly time 9904.571289: total scene time 713129.125000


end of stats
```