ANIMAC:
A Multiprocessor Architecture for Real-Time Computer Animation

Thesis by

Daniel S. Whelan

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

1985

(Submitted May 23, 1985)

Acknowledgments

Many individuals influenced me during my graduate years at Caltech. I am particularly indebted to Jim Kajiya, Al Barr, Jim Blinn and Carver Mead. Jim Kajiya has served as both an adviser and a friend. I'm grateful to him for starting me in this work. Al Barr motivated me to investigate shadowing. Both Jim and Al encouraged me to continue my research when I was near to giving up and I thank them for doing so.

For two years, I had the good fortune to work with Jim Blinn. During that time, I developed a special appreciation for the amount of effort required both to make outstanding animation and to pass that knowledge along to one's students.

Carver Mead served as my advisor during my early years. I'd like to thank him for encouraging me during those formative years and for supporting my decision to pursue computer graphics. Caltech is fortunate to have such a dedicated individual.

There always seem to be some people who help you through when things come down to the wire. Mike Ullner and Cal Jackson reviewed this text; I am grateful for their many helpful suggestions. Jeff Goldsmith and Sylvie Rueff helped me record my animation sequences on video tape; I wouldn't even like to think about how bad things would have been without their help.

I owe special thanks to my fiancée, Sharon, without whose love and support, this work would never have come to fruition. I am fortunate to have parents who have raised their children with love and fairness and who have unselfishly offered us the finest educations available.

Lastly, I would like to acknowledge the financial support of the Defense Advanced Research Projects Agency, the Silicon Structures Project, and the Rockwell Corporation. Their financial support has allowed me the freedom to pursue academic enlightenment.

Abstract

Advances in integrated circuit technology have been largely responsible for the growth of the computer graphics industry. This technology promises additional growth through the remainder of the century. This dissertation addresses how this future technology can be harnessed and used to construct very high performance real-time computer graphics systems.

This thesis proposes a new architecture for real-time animation engines. The ANIMAC architecture achieves high performance by utilizing a two-dimensional array of processors that determine visible surfaces in parallel. An array of sixteen processors with only nearest neighbor interprocessor communications can produce real-time shadowed images of scenes containing 100,000 triangles.

The ANIMAC architecture is based upon analysis and simulations of various parallelization techniques. These simulations suggest that the viewing space be spatially subdivided and that each processor produce a visible surface image for several viewing space subvolumes. Simple assignments of viewing space subvolumes to processors are shown to offer high parallel efficiencies.

Simulations of parallel algorithms were driven with data derived from real scenes since analysis of scene composition suggested that using simplistic models of scene composition might lead to incorrect results.

The ANIMAC architecture required the development of a shadowing algorithm which was tailored to its parallel environment. This algorithm separates shadowing into local and foreign effects. Its implementation allows individual processors to compute shadowing effects for their image regions utilizing only very local information.

The design of the ANIMAC processors makes extensive use of new VLSI architectures. A formerly proposed processor per object architecture is used to determine visible surfaces while new processor per object and processor per pixel architectures are used to determine shadowing effects.

It is estimated that the ANIMAC architecture can be realized in the early 1990's. Realizing this architecture will require considerable amounts of hardware and capital yet its cost will not be out of line when compared with today's real-time computer graphics systems.

Table of Contents

1

Introduction

During the first half of this decade the computer graphics industry has grown at a rapid rate and it is likely to continue to do so for the remainder of the decade. This growth has evidenced itself in many ways, including the spectacular computer-generated special effects found in entertainment and educational films, the widespread acceptance of engineering workstations, and of course, the now ubiquitous personal computer.

The driving force behind all of this growth has been the semiconductor industry. Advances in VLSI technology have enabled a huge amount of storage to be fabricated on a single chip, while drastically reducing the cost per bit of this storage. At the same time, computer architects have been able to utilize this new technology to produce microcomputers that outperform the mainframes of the past decade.

The availability of this increased computational power has opened the doors to new algorithms. Computation has become so relatively inexpensive that many of the algorithms that were once considered too costly are now commonly used. Newer algorithms, which require even more computational resources, have replaced them. A new interest in realism has motivated the development of these algorithms. Generating state-of-the-art computer graphics still remains out of the grasp of the computing proletariat. State-of-the-art computer graphics images are often computed on the highest performance super-computers available. Real-time simulation engines are themselves very powerful and very specialized computers capable of producing realistic images of an environment that reacts instantaneously to a user's actions.

Just as advances in semiconductor technology ushered in this age of computer graphics, newer advances in the same technology promise to offer vastly improved computer graphics technology at a fraction of today's costs. These new advances are appearing in many areas within the computer graphics field. Specialized processing and storage elements have been designed to assist in the rendering of visible surfaces [FUCHS81] [COHEN80] [WEINBE82], the rendering of graphic primitives [WHELAN82] [CLARKE80] [DEMETR83], and in performing constructive solid geometry operations [KEDEM84].

These new architectures are not without their failings, but they offer substantial promise for the future. Their successors should be able to provide performance well in excess of what today's real-time simulation engines provide and at a fraction of the cost.

Future VLSI architectures will apply massive amounts of computational power to computer graphics problems. This thesis studies how this computational power can be focused into a next generation architecture for animation and real-time simulation needs.

This thesis presents an architecture for an animation machine which I have named the ANIMAC. The targeted performance of this architecture is for it to display in real-time scenes composed of 100,000 polygons. This performance goal represents an improvement of almost two orders of magnitude over today's computer graphics systems.

The ANIMAC architecture is more than a blueprint for building a computer system. It is also a new parallel algorithm for producing images of scenes with shadowing effects. This is believed to be the first shadowing algorithm developed for a multiprocessor.

This dissertation is divided into five major chapters. Each of these chapters addresses an important issue in the development of the ANIMAC architecture.

Chapter 2 deals with analysis of synthetic scene composition. Design decisions that affected the ANIMAC architecture were made based upon models and simulations of realistic work environments. Analysis results indicated that the scene composition is not adequately modeled with simplistic assumptions. An important observation indicates that scenes are spatially non-uniform and that the spatial non-uniformity of scenes can be characterized by an asymptotically determined value. This observation strongly suggests that when a scene in spatially divided into more and more distinct regions, the distribution of objects to these regions starts to look similar.

Chapter 3 addresses the problem of how best to use parallelism to construct a high performance visible surface rendering system. The main thrust of this chapter is to investigate how parallelism can be used to achieve the

performance required for the ANIMAC architecture. Simulations were performed of architectures proposed by others and of new ones proposed by the author. The results of these simulations suggest that a two-dimensional array of physical processors together with a simple mapping of image regions onto processors can provide the needed performance with relatively high efficiency.

Chapter 4 develops an algorithm for producing shadowed images on a multiprocessor which spatially subdivides the scene among its processors. This algorithm simplifies shadow determination by dividing the determination of shadowing effects into two processes. A local shadowing process acts only upon information local to the processor while a foreign shadowing process utilizes information residing in other processors. This algorithm is useful for software implementations as well as for hardware implementations since it allows the image computation task to be subdivided down to a manageable size.

Chapter 5 discusses a software implementation and simulation of the ANIMAC shadowing algorithm. Two different shadowing algorithms were implemented for use as local shadowing algorithms. One of these algorithms was found to suffer from a serious problem. This problem and solutions to this problem are discussed. Other difficulties were noticed in the implementation of the foreign shadowing algorithm. The causes behind these difficulties are examined and a solution is offered. The software implementation was used to generate animation sequences which indicated that the algorithm is appropriate for use in animation.

Chapter 6 discusses using the ANIMAC architecture to implement systems capable of real-time performance. Two possible implementations are examined. The ANIMAC-1 system uses ANIMAC parallelization techniques to produce visible surface images without shadowing effects. The ANIMAC-2 system builds upon the ANIMAC-1 system architecture to produce images with shadowing effects. Both of these architectures make extensive use of VLSI architectures, some which have been previously proposed, and some which are new. It is predicted that implementing the ANIMAC architecture will be both technologically and economically feasible in the early 1990's.

Chapter 7 discusses conclusions drawn from the work described in the previous chapters and suggests areas for future work. An appendix contains the results from the simulations discussed in Chapter 3.

2

Analysis of Synthetic Scene Composition

Systems are designed to work within a constrained environment. In computer animation, this environment consists of the model, the view, and the lighting environment. In order to design systems that are capable of working properly within an environment, it is necessary to parameterize the environment in a meaningful manner. Surprisingly, there is little or no meaningful data in the literature. Manufacturers are likely to have developed their own models but must consider them proprietary. Academicians either haven't had the interest in analyzing scene composition or haven't had the tools to do so.

This chapter presents studies of certain properties of synthetic scenes. All of these properties have a bearing on design decisions which must be made in the course of implementing an architecture. Hopefully, the information provided by these studies will allow us to make intelligent tradeoffs between overall system performance and cost.

Analysis was performed on data derived from synthetic scenes. These scenes were selected because of their diversity and availability. By no means are these scenes representative of all images that might be produced by a computer graphics system. Still, these selected scenes are useful. In mathematics, it is common to disprove a conjecture by showing that it does not hold for a specific case. Likewise, designers of computer graphics systems hold certain beliefs. Hopefully, the analysis results will help designers change their beliefs by showing that some of them are not necessarily well founded.

It is usually possible to derive special cases that illustrate the worst case behavior of an algorithm. The scenes selected here have not been intentionally

selected to do this. They are meant to be representative of commonly occurring scenes and hopefully are diverse enough to be representative of more than a single design niche. Figure 2.1 illustrates the six scenes that were selected for analysis. These six scenes range in complexity from 178 polygons to 65,030 polygons.

Each of these scenes was modeled in a language similar to one used by Blinn [BLINN82]. This modeling language has become known as *Blinn Like Format* by the Caltech and the Art Center College of Design community. It is commonly referred to as *BLF*. The language provides support for all of the common modeling transformations, object definition, and object instancing. Scenes are typically modeled as hierarchies of object instances.

These scenes were generated by a graphics package called *render* which was written by the author. *Render* produces images of *BLF* models using a depth buffer algorithm [CATMUL74] to produce shaded visible surface images. Various shaders can be used during the rendering of an object. Gouraud [GOURAU71] and Phong [BUI-TU75] illumination models are most commonly selected by designers.

An additional shader has been incorporated into *render*. This shader collects statistical information about each polygon that the polygon tiler receives. A concise description of each polygon is eventually output to a file. The *statistics* shader tiles polygons differently than other shaders. Instead of updating pixel depth and color information, the *statistics* shader maintains a count of the number of objects that intersect each pixel. This information can be output into another file. For purposes of scene analysis, each of the selected scenes was rendered using the *statistics* shader. The resulting files were used to drive various analysis programs.

Scene analysis has been undertaken to elucidate three particular concerns. First, certain authors have performed performance analysis on multiprocessor computer graphics systems and have based some of their analysis on the premise of having a uniform spatial distribution of polygons [PARKE80]. This seems to be an all too convenient assumption since it makes the performance of these architectures look extremely attractive. Analysis of the spatial distribution of polygons will indicate whether their assumption was well founded or whether these architectures should be re-examined in light of the findings.

The second concern centers around being able to predict the cost for tiling polygons. This is a fundamental step in many visible surface algorithms so it is natural to want to understand the processes that contribute to tiling costs.

The third concern centers around being able to understand the behavior of visible surface algorithms. Visible surface algorithms must sort potentially visible polygons by depth to determine which polygons are visible at a pixel.

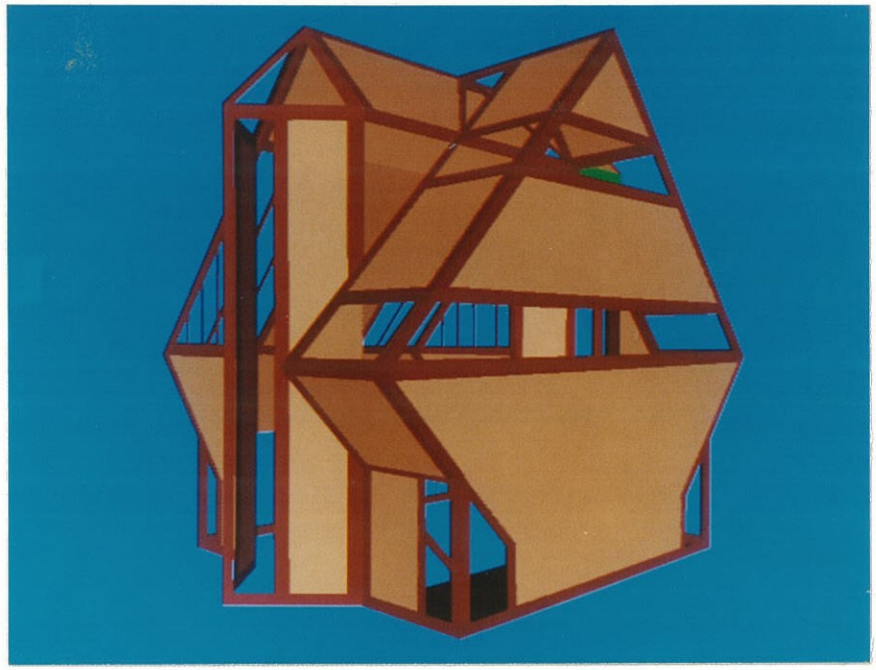


Figure 2.1a: *House*—A simple scene, composed of 178 polygons, modeled by Carol Koffel, Brian Von Herzen, and Jim Hunter.

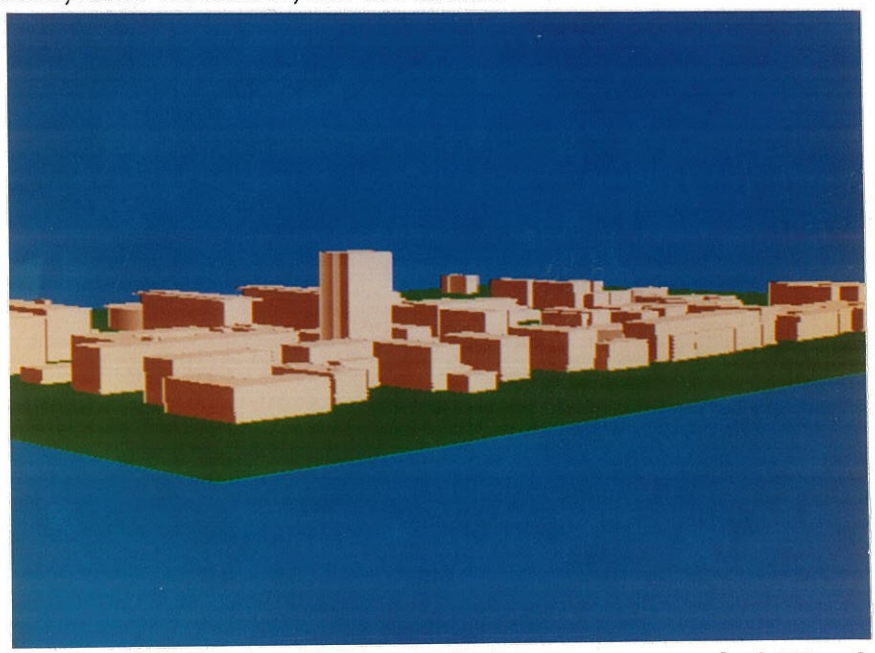


Figure 2.1b: *Caltech*—A moderate complexity scene, composed of 990 polygons, modeled by John Biedenharn and John Platt.

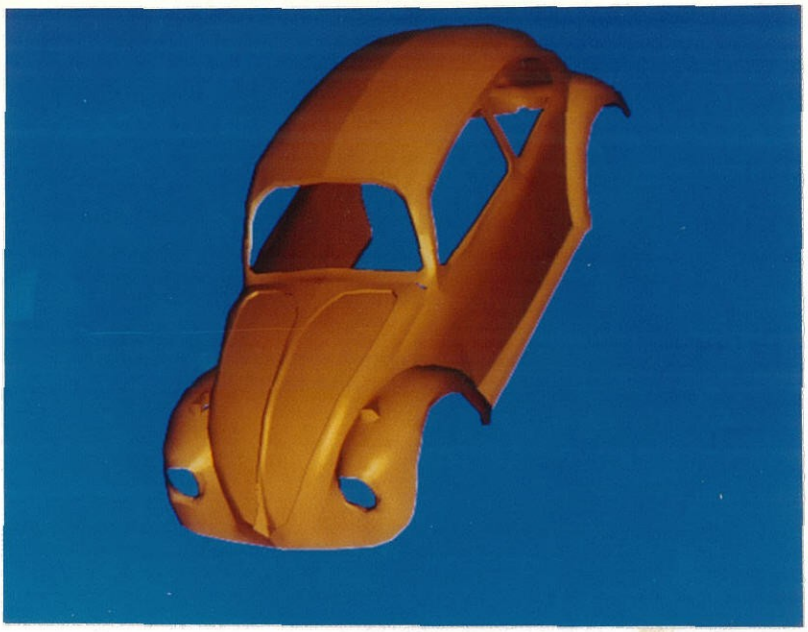


Figure 2.1c: *VW*—A moderate complexity scene, composed of 1072 polygons, modeled by Ivan Sutherland and his students at the University of Utah.

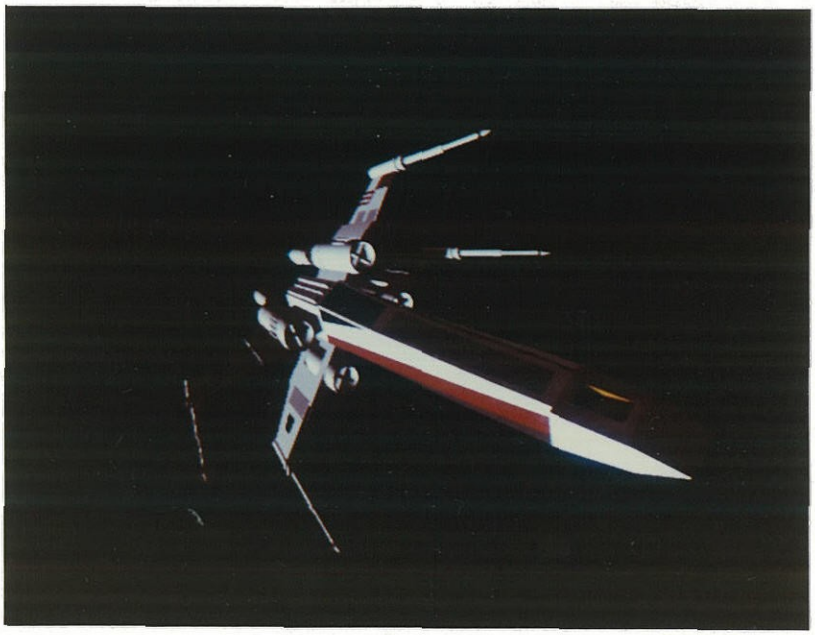


Figure 2.1d: *X-Wing*—A moderate complexity scene, composed of 1407 polygons, modeled by Chuck Esrock.

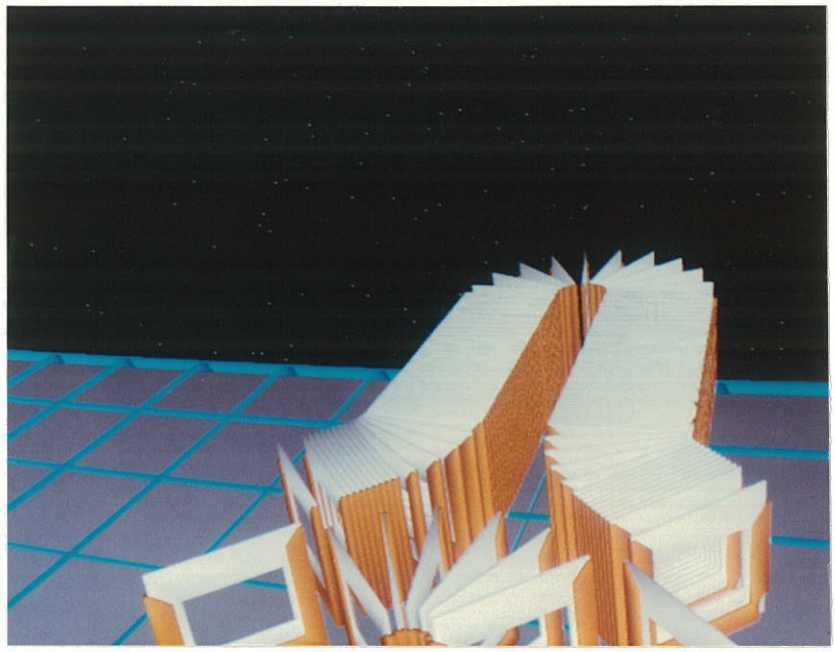


Figure 2.1e: *Frame1100*—A relatively complex scene, composed of 8089 polygons, modeled by the author.

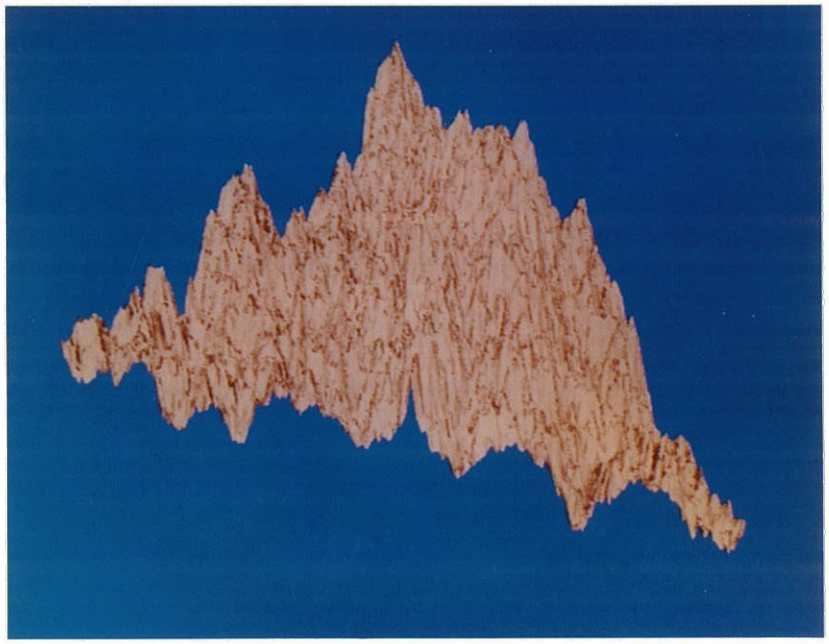


Figure 2.1f: *Fractal64*—A complex scene, composed of 65030 polygons, modeled by the author.

It would be interesting to have a concrete idea as to how many polygons are potentially visible at a pixel.

2.1 Spatial Distribution of Objects

The selected scenes, illustrated in Figure 2.1, do not appear to have a uniform spatial distribution of polygons. Clearly, there are regions of these images that have more polygons than other regions. Most interesting images seem to be spatially non-uniform.

One can't correctly estimate spatial uniformity by viewing an image since this doesn't really indicate how polygons are distributed. Many polygons aren't visible and some polygons meld into others because of the illumination models that were used to generate the image.

To understand how polygons are spatially distributed, an analysis program was written that subdivided the screen space into rectangular regions of equal area and distributed the image's polygons to each of these regions. After all of the polygons had been distributed, each region contained a count of the number of objects that intersected the region. The standard deviation, σ, was computed for the distribution and recorded.

Standard deviations were computed for various numbers of regions ranging from 4 to 256 and for all six test images. So that the results for the various scenes could be compared, the coefficient of variance, V, was computed as:

$$V = \frac{100\sigma}{\bar{x}}$$

where $\bar{x}$ is the mean of the counts for the various regions. A uniform spatial distribution would result in $\sigma = 0$ and $V = 0$. $V > 0$ indicates how far the distribution is from being uniform.

Figure 2.2 presents a plot of V versus the number of regions. Each curve represents one of the six test scenes. All scenes appear to be spatially non-uniform since $V > 100\%$. In general, V increases to some asymptotic value as the number of regions increases. This behavior is expected since in the limit a region will have either zero polygons or some number of polygons which represents the depth complexity for a point.

These results are interesting in that they suggest that the distribution of polygons to regions starts to look similar when the number of regions increases. Distributions for most of the scenes seem to become similar for partitionings into 75 or more regions.

While these results suggest that the distribution starts to look similar as the number of regions increases, they do not suggest that the number distribution of objects to regions becomes uniform. In fact, they suggest that there is inherent non-uniformity to these scenes.

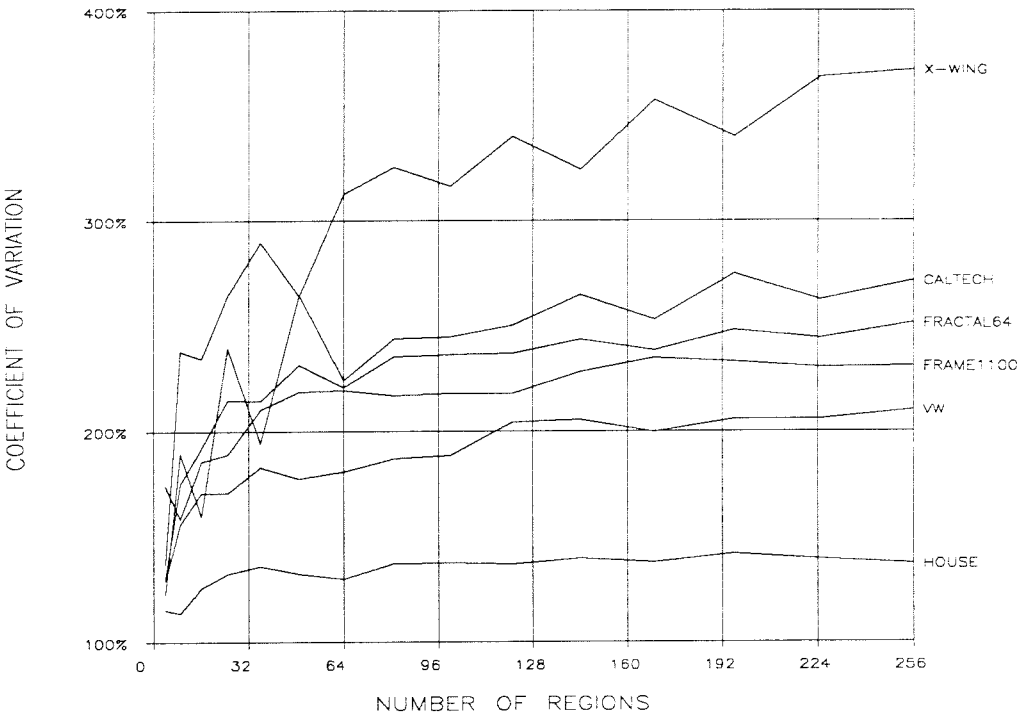


Figure 2.2: A plot of the coefficient of variance, V, versus the number of regions that scenes were divided into for the six test scenes.

2.2 Polygon Tiling

Polygon tiling is at the center of most visible surface determination algorithms. Polygon tiling determines which pixels are within a polygon's borders and what color to render them. Depth buffer algorithms use polygon tiling exclusively to implement the visible surface algorithm at each pixel.

Since the ultimate action of any visible surface algorithm is to tile a polygon, or a portion thereof, it makes good sense to study the various subcosts that contribute to the total cost of tiling a polygon. It also makes good sense to study how polygon tiling costs are affected by increasing scene complexity.

Polygon tiling can be broken down into several readily identifiable subcosts. Parke [PARKE80] suggests that if polygon scan conversion is performed by a single processor, the total time T is:

$$T = N_p T_p + N_s T_s + N_e T_e$$

where:

N_p	the number of pixels computed
N_s	the number of scan line segments
N_e	the number of polygon edges

and:

T_p	the computation time per pixel
T_s	the computation time per segment
T_e	the computation time per edge

Parke's analysis suggests that the times associated with processing a segment and processing edges are about the same, and about seven times greater than the per pixel time T_p. This suggests that for very small polygons the computation time may be entirely dominated by the edge and segment terms. In his paper, Parke does not adequately describe what costs the T_p term includes. From his numbers, it would seem that it includes the time to color a single pixel with a very inexpensive illumination model. Experience with the author's *render* program suggests that more realistic illumination models, such as the one described by Bui-Tuong, easily cause the N_pT_p term to dominate even for very small polygons.

The term N_pT_p may dominate, so N_p, the area of a polygon as it is projected onto the screen, represents an important metric. A program was written to analyze the distribution of polygon areas. This program computes a histogram which represents the distribution of polygons by projected area $\sqrt{N_p}$. The square root of the areas was sampled in order to compress the data into a meaningful range. Polygon areas were sampled and binned into 512 bins.

Figure 2.3 presents the results in a form that allows the histograms to be easily compared. The abscissa represents the square root of polygon area and ranges from 1 to 512. The ordinate is a percentage which has been computed as:

$$y(n) = \sum_{i=1}^{n} x_i \Big/ \sum_{i=1}^{512} x_i$$

where x_i is the bin count. This measure is useful since it allows us to determine that polygons with $\sqrt{N_p} \leq n$ constitute a certain percentage of the polygon

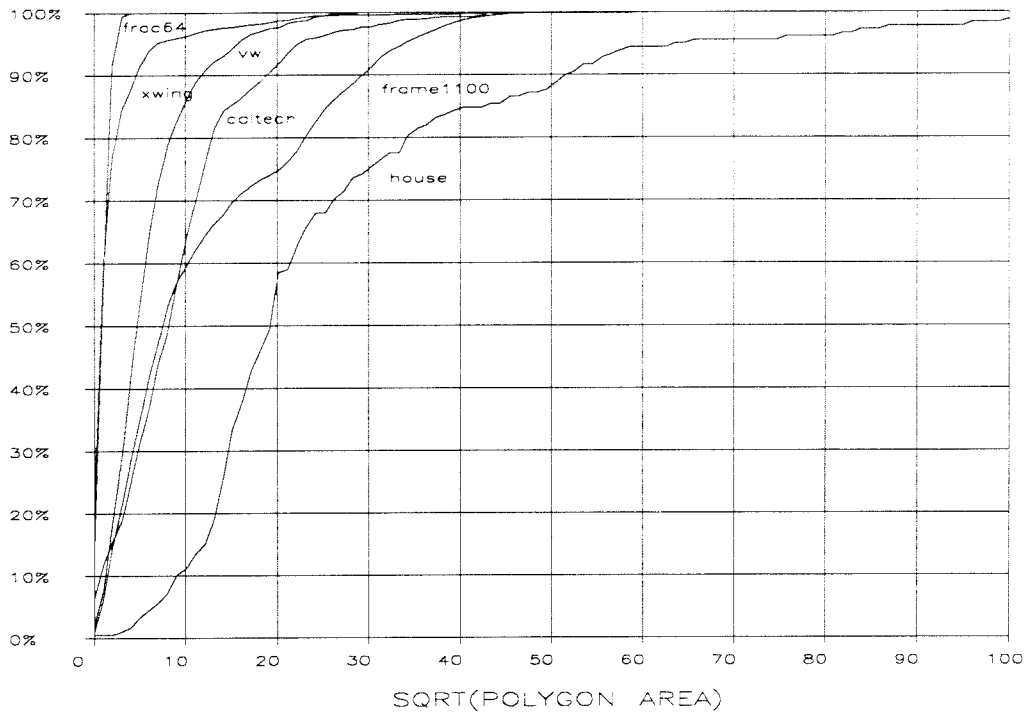


Figure 2.3: A plot of the distribution of polygons by area as a percentage of all polygons in a scene.

population. The plot in Figure 2.3 has been scaled to show $\sqrt{N_p} \leq 100$ since most of the curves saturate by this point.

Figure 2.3 indicates that the vast majority of all polygons are small. We also notice that the median polygon area tends to decrease as scenes become more complex. This behavior is expected.

Another analysis of polygon area distribution is also important. Depth buffer algorithms require time proportional to the total number of pixels, N_d, that must be tiled for a scene. N_d is often called the drawn area and is simply computed as:

$$N_d = \sum_{i=1}^{p} N_p$$

where p is the number of polygons in a scene. A plot similar to Figure 2.3 can be produced that corresponds to the distribution of polygon areas as a

percentage of drawn area. In this case, the ordinate of a point with abscissa n was computed as:

$$y(n) = \sum_{i=1}^{n} x_i i^2 \Big/ N_d$$

The results of this analysis are presented in Figure 2.4. This plot indicates that small objects contribute to the majority of the drawn area, N_d, but that for some scenes a few large polygons also make significant contributions to N_d.

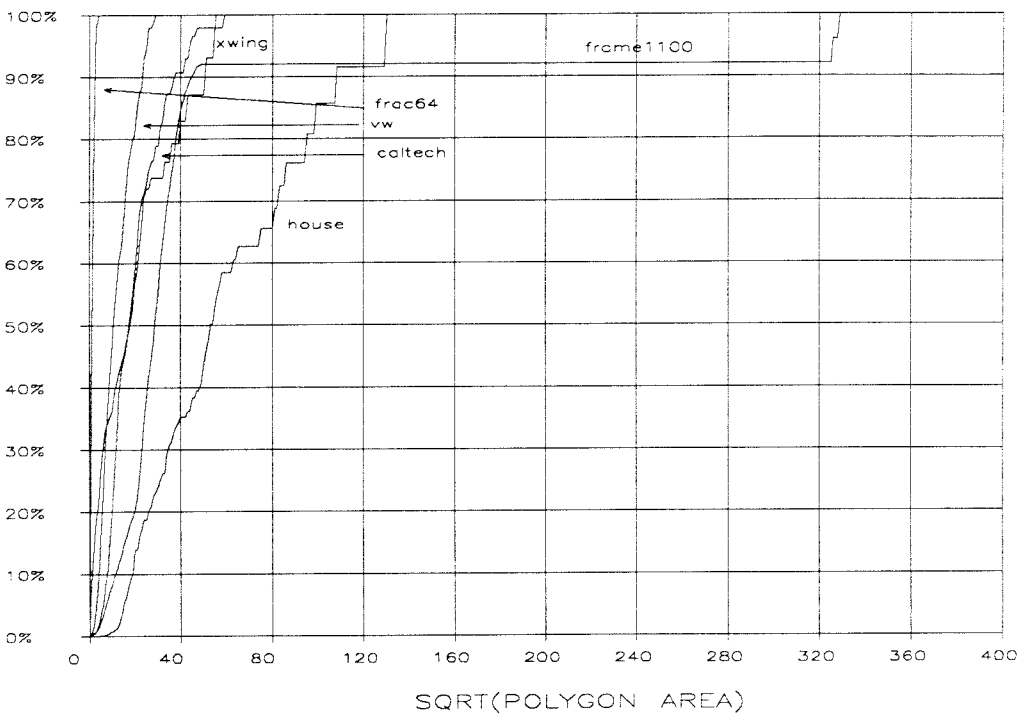


Figure 2.4: A plot of the distribution of polygons by area as a percentage of drawn area, N_d, for a scene.

The number of polygonal edges, N_e, also affects tiling cost through the $N_e T_e$ term. A number of proposals have favored limiting polygons to having at most a fixed number of edges, usually three or four. Limiting the number of polygonal edges allows certain resources to be statically allocated, generally reducing hardware costs. In addition, there may be another good reason to limit the number of polygonal edges. Most of the illumination models rely on interpolation of colors or surface normal vectors. This works best if the

polygon is convex. The results are usually so undesirable when the polygon is concave that limiting the number of polygonal edges to three is not uncommon.

Figure 2.5 illustrates the distribution of polygonal edges for the selected scenes. The results have been plotted in a fashion similar to Figure 2.3. In this case, polygons have been binned by their number of edges. The horizontal axis of Figure 2.5 represents the number of edges in a polygon. The ordinate has been computed as:

$$y(n) = \sum_{i=1}^{n} x_i \Big/ \sum_{i=1}^{512} x_i$$

such that it represents the percentage of polygons with the same or fewer edges. The figure indicates that, for all six scenes, the vast majority of polygons have four or fewer edges. This is not surprising given the environment in which the scenes were modeled.

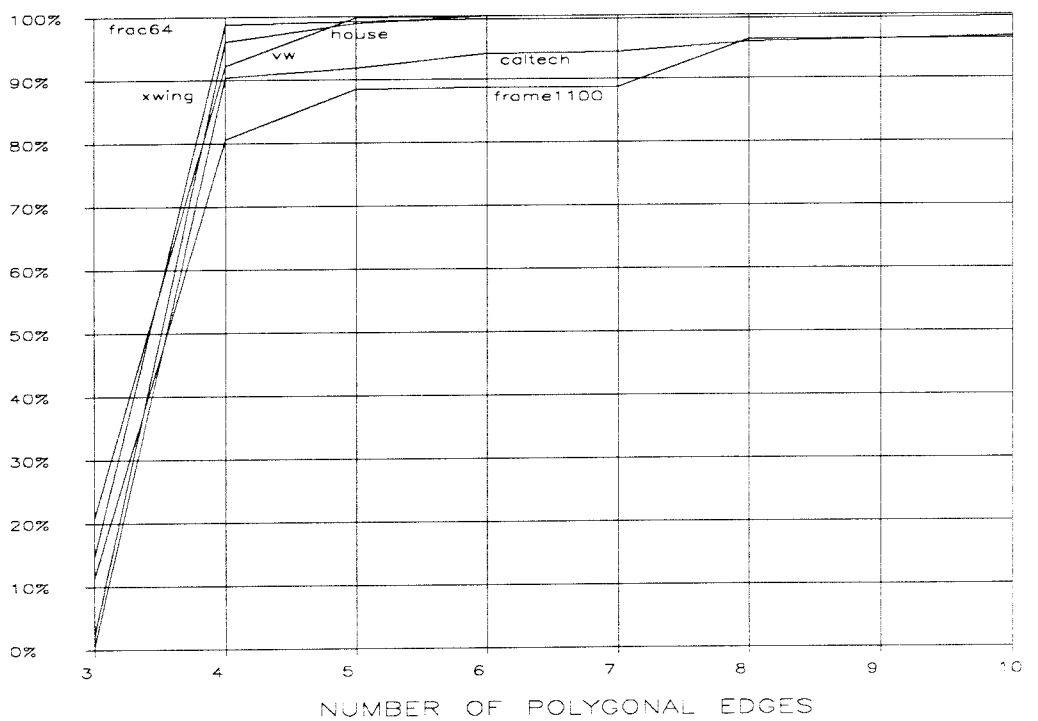


Figure 2.5: A plot of the distribution of polygons by number of edges.

The number of scan lines that a polygon intersects, N_s, is another factor that affects tiling costs. If we were to plot the distribution of N_s for these test scenes, the resultant figure would appear very much like Figure 2.3. This would be the case since we can model N_s as a function of N_p as:

$$N_s = A\sqrt{N_p}$$

where A is an aspect ratio correction factor. The aspect ratio of a polygon can be described by the ratio of a polygon's width to its height. Square polygons will have an aspect correction factor $A = 1$, while non-square polygons will have $A > 1$ or $A < 1$.

Plotting the distribution of N_s will not offer much information about the distribution of aspect ratios. Instead of plotting distributions of N_s, the distribution of aspect ratios was examined. Each scene's polygons were binned by aspect ratio, the results are presented in Figure 2.6. Separate plots are shown for each scene since the plots for multiple scenes appeared very cluttered and were impossible to decipher. Each scene's histogram has been plotted as a polar plot occupying one quadrant. When drawn this way, the angle of each bin's walls correspond to the range of aspect ratios that fall into the bin. Only one quadrant is utilized since aspect ratios are computed from a polygon's width and height which are unsigned quantities.

Figure 2.6 indicates that the distribution of polygon aspect ratios is not uniform for these six scenes. There often appears to be a single preferred aspect ratio but several of the scenes have two or more preferred aspect ratios.

2.3 Depth Complexity

Depth complexity at a pixel is a measure of how many polygons completely or partially cover that pixel. Depth complexities are important for two reasons. First, since many visible surface algorithms explicitly sort data by depth for each pixel, a measure of depth complexity will indicate how well these algorithms will perform on a scene.

Depth complexity is also important for another reason. High pixel depth complexities indicate that many surfaces interplay at a pixel and suggest that the pixel is likely to experience aliasing problems. High depth complexities do not mandate that a pixel will experience aliasing problems since a polygon may cover the entire pixel region hiding all other polygons from view.

At least one modern depth buffer algorithm, the A-Buffer developed at Lucasfilm [CARPEN84], maintains a list of all polygons that intersect the pixel in the buffer so that transparency and antialiasing effects can be calculated. One hardware architecture also proposes accumulating a list of objects that

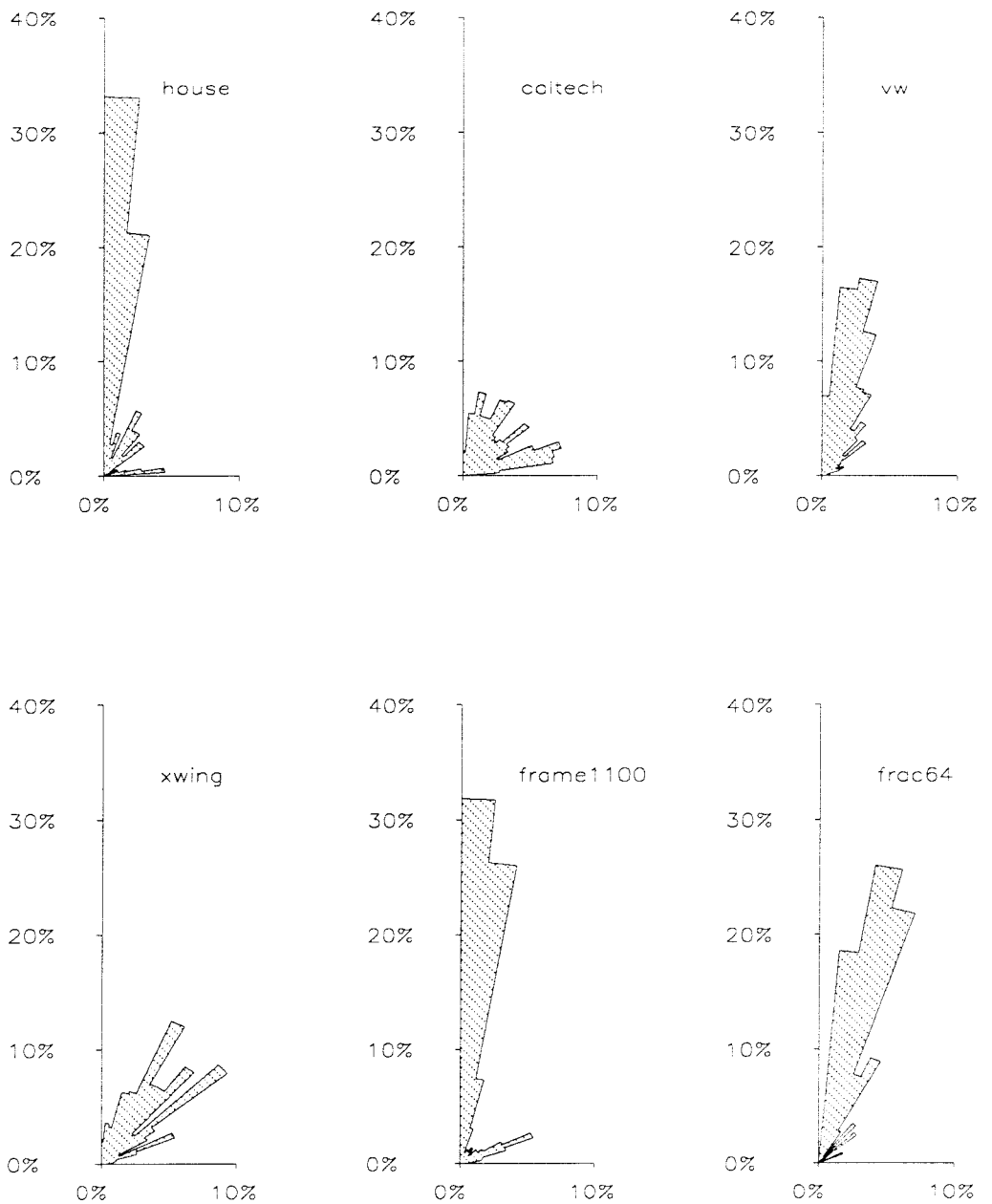


Figure 2.6: These six plots illustrate the distribution of polygon aspect ratios for the six scenes. The most vertical bin represents polygons that are tall and narrow while the most horizontal bin represents polygons that are short and wide.

intersect each pixel region for antialiasing purposes [WEINBE82]. It seems to be an evolving trend to keep a list of objects that intersect pixel regions. Often these lists can be pruned so that the length of a list never approaches the pixel depth complexity. Nevertheless, it is interesting to examine the distribution of pixel depth complexities since depth complexities suggest a worst case behavior for these algorithms.

Figure 2.7 illustrates pixel depth complexity distributions for the selected scenes. All six test scenes have median pixel depth complexities less than or equal to five. Ninety percent of the pixels in four of the scenes have depth complexities less than nine. The two most complex images, *Frame1100* and *Fractal64*, have much higher depth complexities. These images also experience the most severe aliasing problems which supports the suggestion that aliasing problems are correlated to high pixel depth complexities.

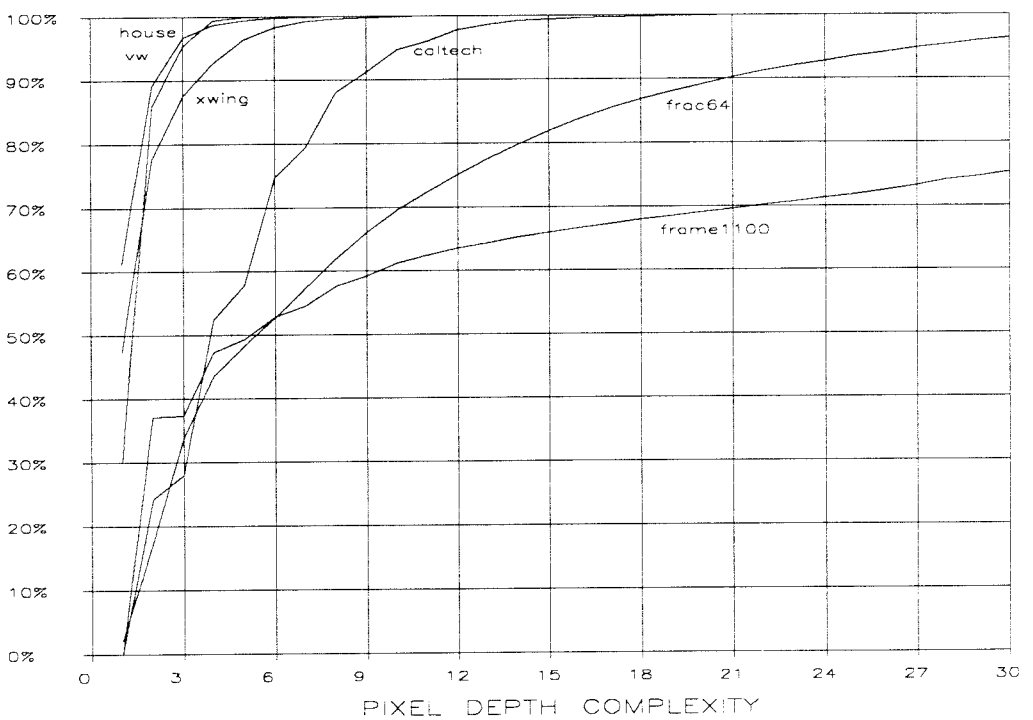


Figure 2.7: Distribution of pixel depth complexities for the six test scenes. The ordinate value indicates the percentage of the pixels that have depth complexities less than or equal to the abscissa value.

2.4 Conclusions

This chapter analyzed several interesting characteristics of scene composition. The data presented here was gleaned from a small sample of six scenes and the results should be considered in light of this. Nonetheless, several interesting points can be made about scene composition. These include:

1. Scenes are not composed of a spatial uniform distribution of polygons and assuming otherwise will invariably lead to incorrect assumptions.
2. It appears that the spatial non-uniformity of scenes can be characterized by an asymptotic value. This value is computed as the coefficient of variance of the number of polygons in a region as the number of regions goes to infinity.
3. The majority of polygons are small and their size decreases as scene complexity increases. In certain scenes, significant amounts of the total drawn area can be attributed to a few large polygons.
4. Most polygons have few edges. Requiring polygons to be triangles probably will only double the number of polygons in a scene model.
5. Polygon aspect ratios do not appear to be distributed uniformly. Scenes appear to have their own preferred aspect ratio but some scenes may have more than one preferred aspect ratio.
6. Most pixels seem to have small depth complexities although it is easy to create a model and view of that model with very high pixel depth complexities. Aliasing difficulties seem to be correlated to high average pixel depth complexities.

The analysis presented in this chapter only touches the tip of an iceberg. There are many other interesting parameters that affect scene composition that haven't been addressed. Continued work in this area could lead to the development of a statistical model of scene composition that could be used for statistically analyzing algorithms or for driving simulators.

Since scenes are very complex and an adequate statistical model does not currently exist, the performance of algorithms is best determined by simulating them with realistic work loads that are derived from real scenes.

3

Utilizing Parallelism in Computer Graphics Systems

Computer graphics systems are used in animation and simulation tasks to generate sequences of images. Animated images may be very complex, and computing a single frame often requires many minutes or even hours of processing time. Real-time simulation requires that images must be generated within a frame time. Because of this, real-time simulation systems have been limited to displaying relatively low complexity images rendered with simplified lighting models.

Parallelism can be utilized to significantly reduce the time needed to generate images. Reducing image generation time will allow more complex scenes to be rendered in real-time simulation environments and will enable animators either to reduce production costs or to enhance their imagery. This chapter explores several ways of utilizing parallelism to this end.

To a certain degree, parallelism has always been employed in computer graphics systems, since most are not capable of generating data at video rates without relying upon parallelism to make their memory devices appear to be much faster than they really are. This chapter will not address such fine grain uses of parallelism, but will instead address parallelism on a much coarser level. This work is targeted at applying parallelism at the subsystem or functional-unit level in order to produce completed images faster than conventional sequential architectures would.

This chapter addresses the problem of how to make efficient use of parallelism in order to dramatically reduce the time needed to compute sequences

of images for use both in animation and simulation environments. Acceptable solutions will improve both system throughput and latency.

Many people have pointed out that throughput can be increased rather arbitrarily if the latency, the time required to compute a single frame, remains constant. A simple implementation improves throughput by a factor of N by employing N identical computer systems, each of which computes $1/N$ of the frames.

Decreasing latency is an important goal for two reasons. First, by decreasing latency we improve the state-of-the-art in real time simulation by allowing more complex images to be computed in the same fixed amount of time. Future generation simulation engines will produce images of today's animation complexity in real-time. The work, presented in this thesis, attempts to lay the groundwork for the transition between the architectures of today's simulation machines and those yet to come.

Second, decreasing latency decreases turn-around time. Animation work may be divided into two phases: design and production. Systems that deliver higher throughput may reduce production time. Systems that provide lower latency reduce design time by providing the designers with faster feedback.

Inefficient uses of parallelism are not practical solutions. While some parallel architectures might provide both higher throughputs and lower latencies, the cost to run these machines may relegate them to the junk pile if they make inefficient use of their resources. Therefore, we are necessarily constrained by the market place to produce cost-competitive solutions. Doing so requires making use of parallelism in a very efficient manner.

In this chapter, I will present an overview of previous work done in applying parallelism to computer graphics. Afterwards, I will present various architectures for consideration and will simulate them with data derived from real scenes. Results will be compared and an architecture will be selected as a framework for the ANIMAC systems.

3.1 Prior Work

Much work has been expended developing various architectures for computer graphics that exploit parallelism in one way or another. Some of these architectures are merely proposals, or paper studies, while some have been built and used successfully.

I have separated the various architectures into three distinct categories: (1) Spatial Subdivision Architectures, (2) Processor per Object Architectures, and (3) Processor per Pixel Architectures. The rest of this section will discuss these three classes of architectures. The architectures belonging to each category will be presented in chronological order. The applicability of each

architecture to the problem at hand will be discussed and an overall recapping will be made for each subsection.

3.1.1 Spatial Subdivision Architectures

Spatial subdivision architectures use parallelism by dividing a space into subspaces and assigning processors to these subspaces. The space to be subdivided may be any of the spaces common to computer graphics, typically either the image space or the modeling space is used.

Spatial subdivision architectures appear to be much more prevalent than either of the other two types. They take two different approaches to improving the performance of scene rendering.

The first approach utilizes spatial subdivision to associate objects with processors that perform a general visible surface algorithm to tile a portion of the image. Justification for this approach usually relies on Sutherland's observation that visible surface algorithms behave as sorting processes [SUTHER74]. As such, the time required for visible surface determination is determined largely by the number of objects in the scene. Since sorting, and thus visible surface determination, requires a minimum time proportional to the number of objects, N, and commonly time proportional to $N \log N$, decreasing the number of objects dramatically improves performance.

The second approach found in the literature makes use of spatial subdivision to speed up the tiling of objects. Overall performance is improved if tiling of objects consumes a great proportion of the total time.

There has been at least one proposal for a hybrid solution which makes use of spatial subdivision in both of the ways just outlined [PARKE80].

3.1.1.1 Kaplan and Greenberg

Kaplan and Greenberg [KAPLAN79] echo Sutherland's observation that visible surface determination is a sorting process. They note that effective algorithms make use of coherence to reduce sorting times. They suggest that parallel processing is an attractive approach to designing real-time visible surface engines.

They studied the performance, in a parallel environment, of two visible surface algorithms: a Watkins-like scan-line algorithm [WATKIN70] and a Warnock-like spatial subdivision algorithm [WARNOC69]. Figure 3.1 illustrates the two techniques that Kaplan and Greenberg used for subdividing the screen space. Parallelism was utilized by dividing the screen space into: (1) groups

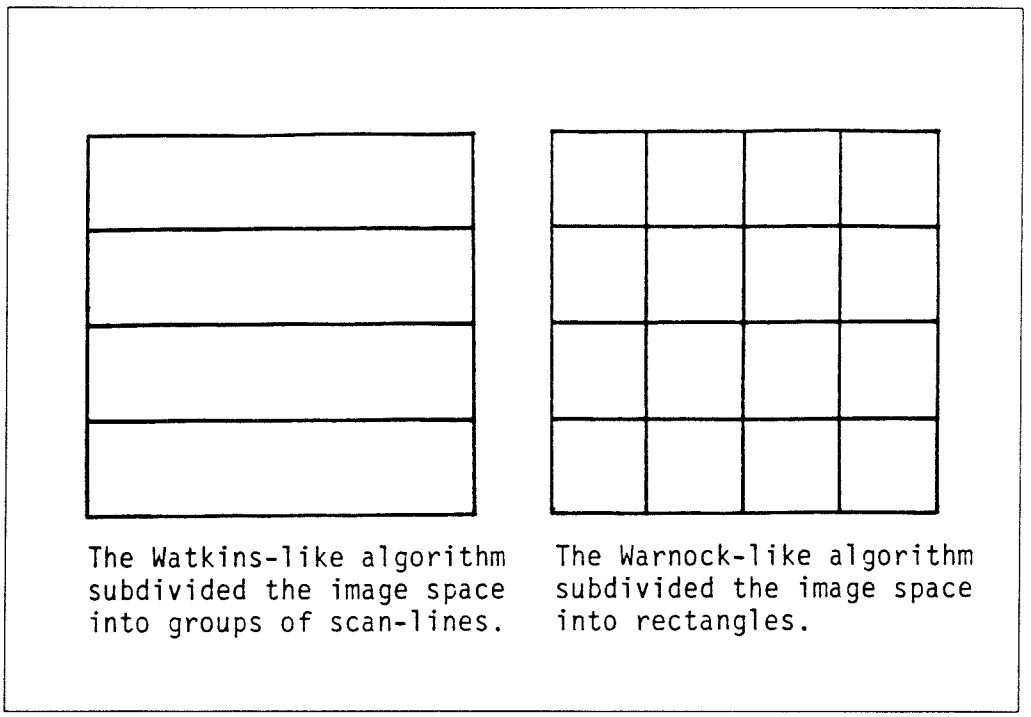


Figure 3.1: Subdivision techniques used by Kaplan and Greenberg.

of adjacent scan-lines for the scan-line algorithm and (2) rectangles for the Warnock algorithm.

The significant observations based on the scan-line algorithm were that: (1) processing time decreased as more processors were utilized, but at a less than linear rate, (2) the number of memory references and depth comparisons for the first scan-line associated with a processor increased, and (3) the processing time required for a particular screen area was highly variable. The first two of these observations were explained by a loss of coherence in the scan-line algorithm. The last observation was explained by the non-uniformity of the object environment.

Observations based on the Warnock-like algorithm were that: (1) total processing time decreased as more processors were added and (2) the number of polygons per processor and the total polygon edge length per processor showed a strong correlation to the processing time required by that processor. Again, processing time was highly variable across the set of processors.

The authors claim that in both cases low intercommunication was required between processors, and they stress the importance of having an easily

calculated value that can serve as a predictor of total processing time. They suggest that one can obtain maximum efficiency from a parallel processor by heuristically scheduling tasks using a predictor function to estimate costs.

3.1.1.2 Fuchs and Johnson

Fuchs and Johnson [FUCHS79] present a multiprocessor depth buffer architecture that uses spatial subdivision to improve tiling performance. Figure 3.2*a* illustrates the overall tiling engine architecture.

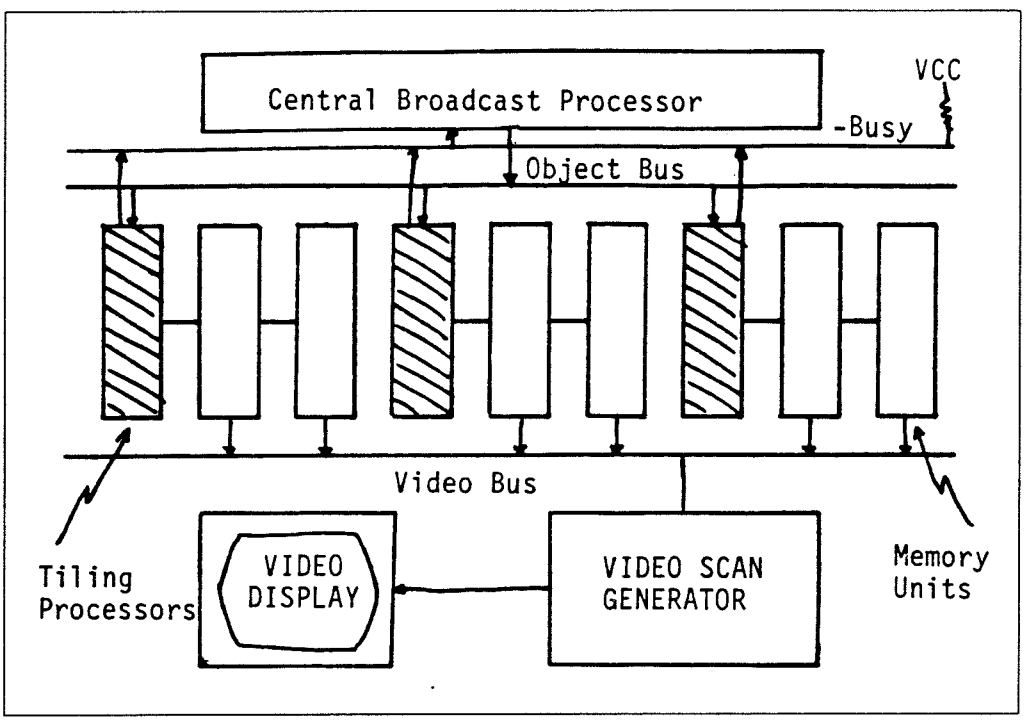


Figure 3.2a: Distributed depth buffer architecture of Fuchs and Johnson.

Many tiling processors operate in parallel and they communicate over two global busses: the *object bus* and the *video bus*. Each processor implements a depth buffer algorithm for some fixed subset of pixels on the screen. Processors have two memories, one for storing pixel colors and the other for storing the associated depths.

Polygons to be tiled are handed to the *Central Broadcast Processor*, which in turn broadcasts a description of the polygon to all of the tiling engines via

the *object bus*. After receiving the polygon description, a tiling processor acknowledges receipt by asserting the *-busy* signal and then colors those of its pixels that lie within the polygon and are potentially visible. When a tiling processor finishes, it lets the *-busy* signal float. The *Central Broadcast Processor* detects that all tiling processors have finished when *-busy* is deasserted. It then may proceed with the next polygon.

The *video bus* is used by the *Video Scan Generator* to retrieve pixel values from the tiling processors. The *Video Scan Generator* assembles a completed image in a frame buffer memory, updating the frame buffer memory whenever a new frame is available.

The novelty of this design lies in the method for partitioning the screen space among the processors. Figure 3.2*b* illustrates how a modulo arithmetic scheme can be used to map adjacent pixels to different processors in a regular fashion. The authors describe this as a tesselation and claim that it allows all processors to work in parallel during the scan-conversion of a polygon. The illustrated tesselation is only one of many possible pixel to processor assignments that the authors consider.

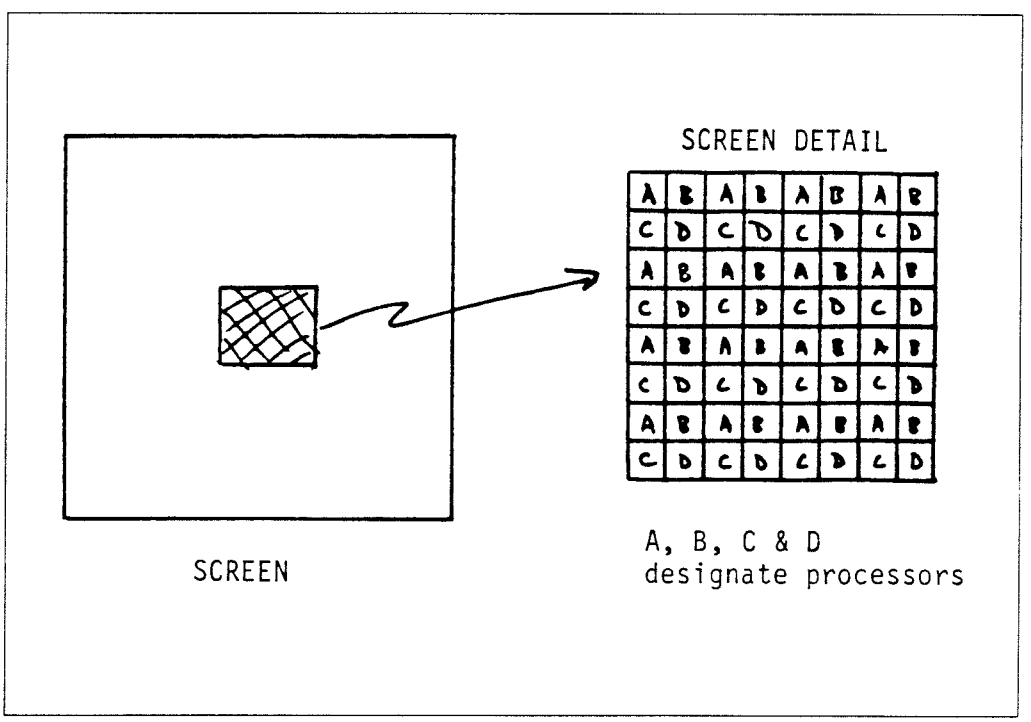


Figure 3.2b: Assignment of pixels to processors in Fuchs' and Johnson's architecture.

3.1.1.3 Parke

Parke [PARKE80] presented an analysis of three different multiprocessor depth buffer architectures. He compared the previous architecture proposed by Fuchs and Johnson with his own architecture, which he calls a Splitter Tree. He also proposed a hybrid of these two architectures.

Parke's Splitter Tree architecture is illustrated in Figure 3.3. It consists of a tree of clipping engines that redistribute polygons to tiling processors. Each engine divides the incoming polygons into two sets according to the side of a clipping plane on which the polygon lies. Polygons straddling the clipping plane are split into two polygons about the clipping plane. A tree of these clipping engines effectively distributes a stream of polygons to a number of processors, each associated with a distinct screen space region. These new processors proceed in parallel to perform a depth buffer scan-conversion process.

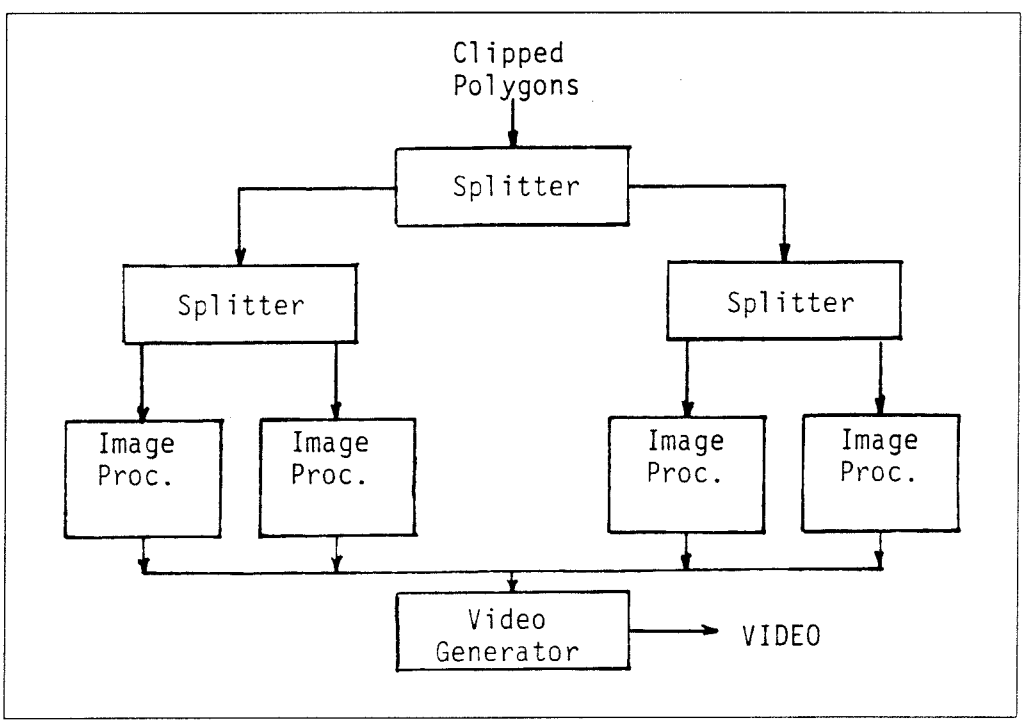


Figure 3.3: Parke's Splitter Tree Architecture

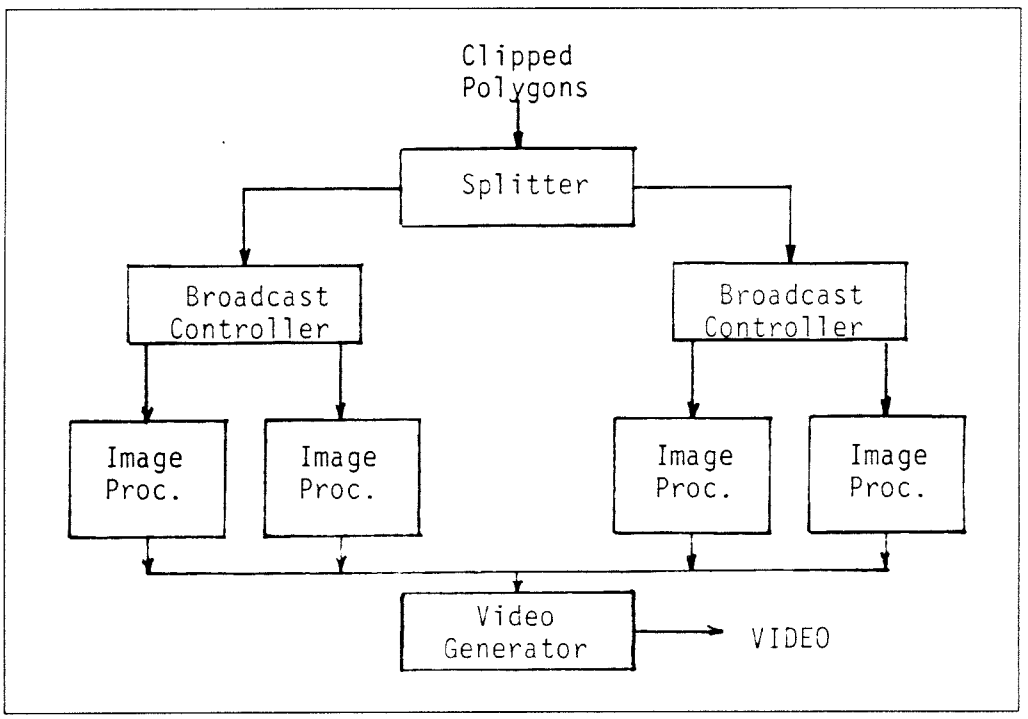


Figure 3.4: Parke's Hybrid Architecture

Parke's Hybrid architecture uses a splitter tree to partition polygons according to screen position, then instead of using a conventional depth buffer engine to scan convert them, it uses a multiprocessor similar to that of Fuchs and Johnson to tile the polygons in each subregion. This hybrid architecture is illustrated in Figure 3.4.

The simulation of Parke's splitter tree architecture shows impressive results when compared with Fuchs' machine and the Hybrid engine. However, Parke points out that the splitter tree architecture depends on having a uniform spatial distribution of polygons. He suggests that the hybrid architecture might be a better approach since it offers some immunity to a nonuniform distribution of polygons in space.

3.1.1.4 Evans and Sutherland

The CT-5 architecture [SCHUMA80], developed by the Evans and Sutherland Computer Co. (E&S), uses spatial subdivision to reduce the number of objects that must participate in visible surface calculations, thus reducing the time needed to determine the visible surfaces for a region.

The CT-5 architecture includes at least five types of processors. The most interesting processor in the context of this discussion is the Display Processor, which accepts image space object descriptions in visual priority order and produces a composite video image.

The display processor divides the image plane into rectangles of adjacent pixels, which E&S calls "spans." Earlier pipeline stages order objects by visual priority so that closer objects will arrive at the display processor first. Objects are then processed a span at a time for all spans that the object intersects.

Visible surfaces are determined by associating a mask with each span. The mask is the union of all objects that have been presented to the span processor so far. When a new object is presented, two new masks are formed: a new union which becomes the new mask, and a difference, which precisely describes which portions of the new object will be visible. This difference image description is passed to a spatial filter which adds the new object's contribution to all of the affected pixels. The result is an antialiased image, which E&S claims has been properly sampled and filtered.

E&S states that their algorithm requires time proportional to the drawn area of the image and claims that as a scene's complexity increases, the system requires less than a linear increase in computing time.

In their conclusions, E&S states that parallel processing is the ultimate goal for the CT-5 architecture. They do not disclose the degree of parallelism utilized in the currently marketed CT-5 systems, although they do indicate that future systems will be heavily dependent upon custom VLSI chips to realize parallel implementations of the CT-5 architecture.

3.1.1.5 Clarke and Hanna

Clarke and Hanna [CLARKE80] proposed an architecture that utilizes spatial subdivision of the screen space to achieve higher tiling rates by increasing the effective bandwidth to the image memory. Instead of multiplexing image memory address and data busses, as is the common practice, they associate custom VLSI memory controllers with small collections of memory chips. These VLSI chips can be configured as either Column Image Memory Processors (C-IMPs) or as Row Image Memory Processors (R-IMPs). Figure 3.5 illustrates a configuration of C-IMPs and R-IMPs.

C-IMPs accept commands from the parent processor and issue commands to the R-IMPs. These commands allow for line, character, or polygon rendering, and for raster merging. In parallel, the C-IMPs subdivide the rendering tasks and redistribute the work to their R-IMPs.

R-IMPs are connected directly to the image memory chips and are responsible for scan-conversion. This architecture uses a tesselation scheme much like

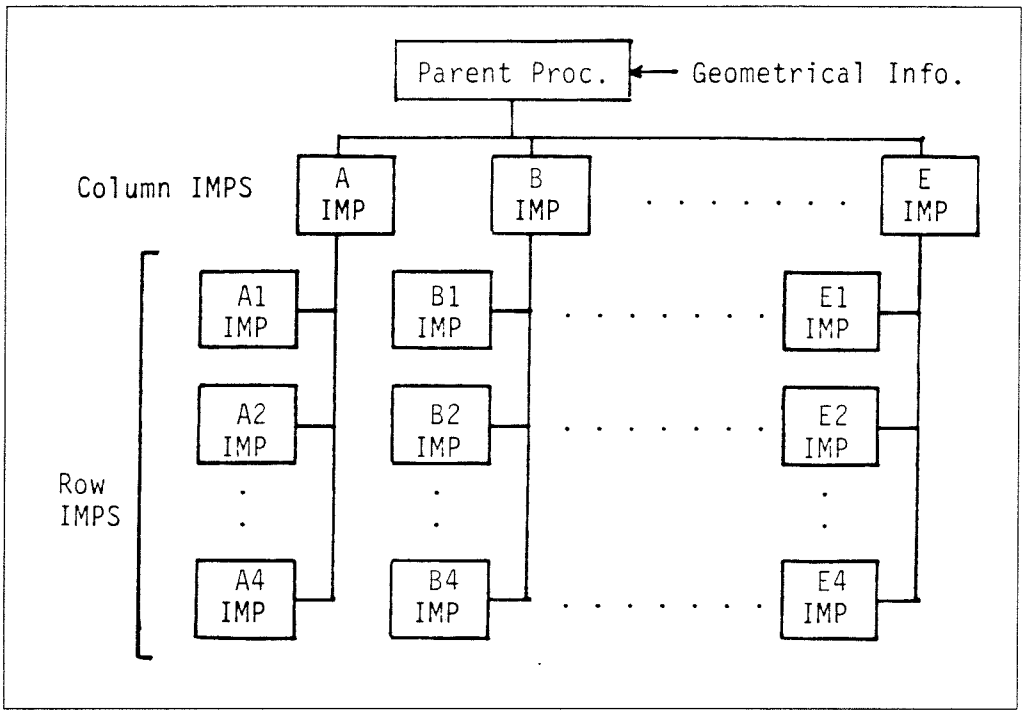


Figure 3.5: Image Memory Processor Architecture

that of Fuchs and Johnson to associate pixels with the R-IMPs. Thus, an adjacent group of pixels are effectively scattered across the collection of R-IMP processors.

This architecture is able to achieve very high scan-conversion rates for large polygons. Performance for small polygons degrades when all of the R-IMPs are unable to share the task. Worst case performance would be that of a combined C-IMP and R-IMP.

The Image Memory Processor architecture can be extended to implement a parallel depth buffer algorithm. The authors suggest that such an approach would provide a very cost effective hidden surface rendering system.

3.1.1.6 Ullner

Ullner [ULLNER83] presented several ray tracing machines in his dissertation. One of these, which he called the "Ray Tracing Array," makes use of spatial subdivision of the modeling space.

Ullner's Array is a two-dimensional array of microprocessors with specialized ray intersection and communications hardware. Although the processors

form a two-dimensional array, the modeling space is subdivided along all three axis. Contiguous subvolumes along a doubly skewed path are mapped onto a single physical processor. This mapping exhibits the property that the neighbor of any subvolume resides in a neighbor of the processor to which the subvolume is mapped. Furthermore, the mapping spreads a family of rays onto an equal number of physical processors. This last property is important to the load balancing of Ullner's machine.

The Array ray traces an image by first associating objects with physical processors. This is done in the manner previously described. Once the objects have been loaded into the processors, many different views may be computed without reloading. To create a view, rays are cast from the viewpoint and fall fairly uniformly on the physical processors. Each processor accepts a ray and checks whether any objects within its subvolume intersect the ray. Several things may happen, all of which generate ray messages. First, the ray may intersect an object, in which case child rays are cast. Second, the ray may not intersect any objects, in which case the ray is passed onto a neighboring subvolume's processor. Third, a ray may meet a termination condition, in which case a result ray message must be created and sent back to the originating processor.

Ullner discusses extensions to his architecture that enable it to handle more general objects than polygons. He also addresses ways to virtualize the algorithm so that subvolumes and ray messages can both be swapped onto disk.

3.1.1.7 Dippé and Swensen

Dippé and Swensen [DIPPÉ84] presented another architecture for a ray tracing machine that utilizes subdivision of the modeling space. Their ray tracing algorithm is essentially similar to that of Ullner. They presented analysis which shows that a message passing algorithm that checks only for ray intersections in the processors along the path of the ray can provide substantial speedups over naïve algorithms that check for intersections in all processors.

Of much more interest to this discussion, they propose using a three-dimensional subdivision of the modeling space in which each subvolume is to be associated with a processor. Adjacent processors can communicate by sending and receiving messages. In order to achieve load balancing, they propose allowing the mapping of subvolumes to be both dynamic and adaptive. They propose using the product of the number of objects within a region and the number of rays in that region as a cost metric to be assessed by a global algorithm that directs the redistribution of resources.

The authors have simulated their algorithm on conventional computers and present a pleasing but by no means state-of-the-art image. They suggest

the adaptive nature of their algorithm is important and indicate that the use of adaptive subdivision algorithms be applied to other types of visible surface algorithms.

3.1.1.8 Summary of Spatial Subdivision Architectures

The six spatial subdivision architectures just reviewed use parallelism in two different ways. Two architectures used parallelism to improve tiling performance. The other four architectures used parallelism to reduce the number of objects that any processor would have to deal with when deciding which surfaces are visible.

The distributed depth buffer architecture proposed by Fuchs and Johnson and the Image Memory Processor architecture proposed by Clarke and Hanna both use parallelism to speed up the tiling of graphical objects. While these architectures are different in how they distribute tasks among processors, they are very similar in how they actually apply parallelism to object tiling.

Both architectures use multiple processors to tesselate the screen plane as in Figure 3.2*b*. Given such an architecture, the time, T, required by these architectures to tile a polygon can be written as:

$$T = T_S + T_P\lceil N_P/P \rceil$$

where:

T_S	per polygon setup time
T_P	pixel modification time
N_P	projected area of the polygon
P	the number of processors

As noted in Chapter 2, the average polygon size decreases as a scene becomes more complex. If the area of a polygon projected on the screen, N_P, decreases to the point where $N_P < P$, some of the P processors do not contribute to the tiling of the polygon. In such a case, polygon tiling time is described by:

$$T \overset{N_P \to P}{=} T_S + T_P$$

We see that as scene complexity increases, T asymptotically approaches a value independent of P, the number of processors. Once this condition has been met, adding processors has no affect on system performance. In fact, P can be decreased without affecting performance. When scenes meet the Kajiya

criterion of complexity†, objects are typically smaller than a single pixel. In this case, we see that we can set $P = 1$ and not suffer a performance loss.

The asymptotic value of T, may very well be dominated by the T_S term. Such architectures are best run well away from their asymptotic knee where:

$$T_P\lceil N_P/P \rceil \gg T_S$$

Under these conditions, system performance is enhanced through the use of parallelism and can be improved by adding more processing elements.

The second class of spatial subdivision architectures use parallelism to redistribute objects to processors that perform a visible surface determination algorithm for the region associated with each processor. The four architectures in this category used different techniques to redistribute objects. These included regular subdivisions along one, two, or all three axes, as well as irregularly spaced subdivisions. The architects hope that their redistribution strategy balances the load on the array of processors so that each processor ends up with the same number of objects.

The behavior of these architectures depends upon two variables: how effectively objects are redistributed and the time complexity of the visible surface algorithm that each processor uses to render an image of its region. A few definitions will be useful for the following discussion of performance.

P_i	Processor i
N_i	Number of objects in P_i
T_i	Time required by P_i to handle N_i objects

Two time metrics are of interest. Generally when an image is computed one waits for the entire image to be computed before viewing it. This is akin to stating that all processors must finish their tasks and can be written as:

$$T = \max(T_1, \ldots, T_N)$$

Another time metric is useful in circumstances when individual processors may proceed without waiting for other processors to finish. This behavior may occur in a system that writes each processor's pixel data to disk and eventually collects all of the data together to create a composite image. Such a system would be useful in an animation environment for production runs. In this

† Kajiya suggests that a scene is complex when it consists of more graphical objects than pixels.

case, behavior depends upon the amount of buffering available in the system. If arbitrary buffering is available, an image can be computed in time:

$$T = \frac{1}{N}\sum_{i=1}^{N} T_i$$

The time, T_i required to scan-convert a screen region depends upon the number of objects in that region, such that:

$$T_i = f(N_i)$$

The function f depends upon the nature of the visible surface determination algorithm employed by the individual processors. Depth buffer algorithms require time related to the number of objects $O(N_i)$, while other algorithms commonly require time $O(N_i \log N_i)$. Thus T_i can be rewritten as:

$$T_i = O(N_i) \qquad \text{or} \qquad T_i = O(N_i \log N_i)$$

Performance improvement, ρ, can be viewed as the time required by a uniprocessor divided by the time required for a multiprocessor. For these two visible surface algorithm behaviors, and for ideal distributions of objects to processors, we find:

$$\rho = O\left(\frac{1}{P}\right)$$

and:

$$\rho = O\left(\frac{P \log N}{\log \frac{N}{P}}\right)$$

These spatial subdivision architectures exhibit performance improvements that depend upon the number of processors. Adding more processors can improve overall performance if objects are evenly distributed among the processors. Unlike the tiling architectures, these spatial subdivision architectures would seem to perform best for highly complex scenes. Simple scenes contain few objects and are more likely to result in uneven distribution of objects to processors. The law of large numbers would seem to suggest that it would be easier to evenly distribute objects from more complex scenes.

3.1.2 Processor per Object Architectures

Processor per object architectures associate a processor with each graphical object in a scene. Collectively, the processors determine which of the objects is visible at each pixel.

Some of the earliest real-time graphics systems, such as the GE NASA system [SCHACH83], used processor per object architectures. Over time, these systems have evolved into systems that do not statically allocate a processing element for each object. This evolution has been greatly motivated by a need to build economical and robust systems. Economic motivation resulted from the increased level of integration and faster switching times of integrated circuits. Classical processor per object architectures do not exhibit a high degree of robustness. A system may be configured to handle a specific number of objects and no more. When such a system is presented with too complicated a task, it fails completely instead of degrading nicely. Thus, alternate architectures were sought out.

Interestingly, the "silicon foundry" revolution has led to a renewal of interest in processor per object architectures. The design philosophy promoted by Mead and Conway [MEAD80] stresses that high performance can be achieved in MOS systems by parallelizing the computation instead of serializing it. Furthermore, replication of functional units decreases design and development time. Because of this new design philosophy, we find three modern day proposals for processor per object architectures. These three architectures will be reviewed in the following sections.

3.1.2.1 Cohen and Demetrescu

Cohen and Demetrescu [COHEN80] presented one of the first interesting VLSI architectures targeted at real-time visible surface computer graphics. The Cohen and Demetrescu algorithm may be divided into three distinct phases. These phases—preparation, loading, and compute—operate sequentially, but double buffering may be used to overlap their execution.

During the preparation stage, the scene description is transformed into screen space, clipped, and fractured into triangles. Figure 3.6 illustrates that this architecture consists of a pipeline of processors. During the load phase, geometric and coloring information is passed down the pipeline and loaded into the processors. After the pipeline has been properly loaded, the compute phase may begin.

Each triangle processor contains a row and column counter, together with logic for determining whether the pixel addressed by the row and column counter is inside the triangle and for determining the depth at which the pixel intersects the triangle. During the compute phase, the pixel address is incremented on each clock tick. During each clock period, the processor determines whether its triangle is potentially visible at the current pixel. If the triangle is potentially visible at the current pixel, and the depth of the triangle at the current pixel is closer to the viewer than the depth asserted on the input bus, the triangle's depth and color information are asserted on the output bus;

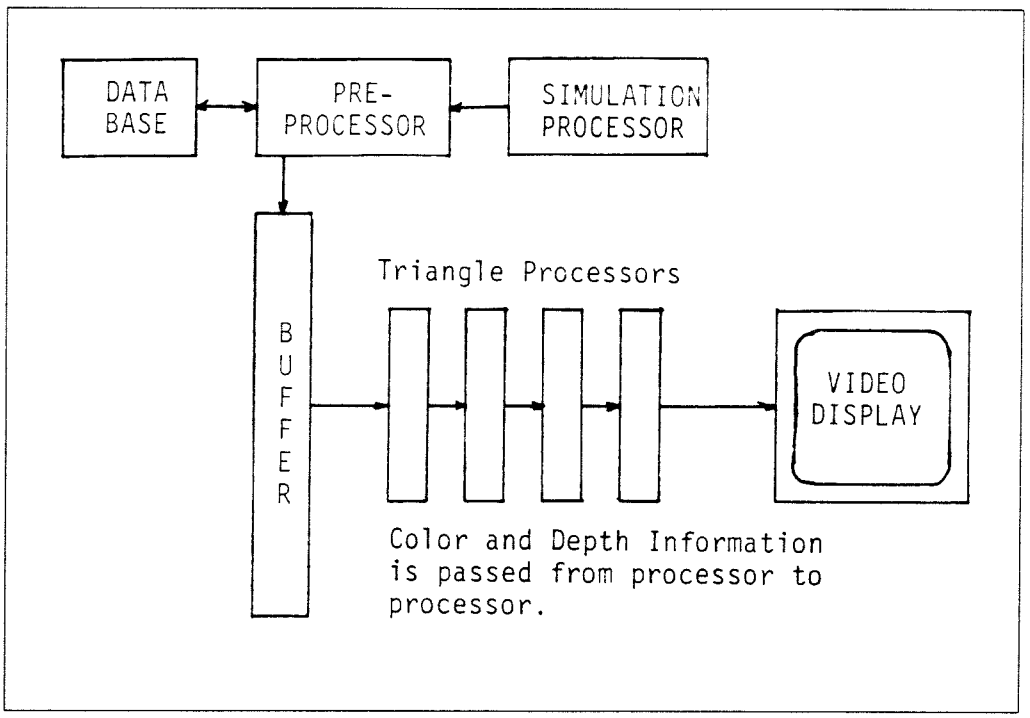


Figure 3.6: Pipelined Processor per Object Architecture proposed by Cohen and Demetrescu.

otherwise depth and color information from the input bus are simply passed through the processor onto the output bus.

The output from the pipeline is a stream of pixel colors and depths for each pixel in scan-line order. These pixel colors can be written into a frame buffer for display or may be displayed directly.

3.1.2.2 Weinberg

Weinberg's work [WEINBE82] builds upon that of Cohen and Demetrescu. Weinberg's goal is to enhance the processor per object paradigm to produce properly antialiased images. Antialiasing computations generally require information about a locality surrounding a pixel center to be present at the time the pixel's color is computed. Depth buffer algorithms, of which Cohen and Demetrescu's is a hardware implementation, are attractive because they perform radix sorts and require only computational time proportional to the number of objects. Unfortunately, this type of sorting prevents the required information about a locality from being easily extractable.

Weinberg's solution requires three major enhancements to the Cohen and Demetrescu architecture. First, he modifies the processing elements so that they do not merely decide whether the center of the pixel lies within a triangle. Weinberg's processors decide whether a region around the pixel center intersects a trapezoid and whether the intersection partially or fully covers the pixel region.

Weinberg also modifies the architecture so that it no longer passes only the closest object's information down the pipeline, but now passes a list of objects that intersect the pixel region. Each processor inserts its object's identifier into the list in such a way that the list is in a visual priority sorted order by the time it emerges from the pipeline. Processors whose objects are completely behind objects already in the list need not insert their objects, and processors whose objects completely obscure the entire list insert their identifier and remove those that are hidden. This behavior prunes the list so that the pixel tiling engine need consider only a few objects per pixel.

Weinberg's third modification is to add a pixel tiling unit which tiles a pixel by computing each trapezoid's contribution on a fine subpixel grid. These contributions are weighted by a filter function and added together to determine the pixel's final color.

Weinberg suggests that his architecture is capable of generating images of 6000 triangles every sixtieth of a second, for a display of 625 by 700 pixels. He would implement this with ECL gate arrays. Each processor requires roughly 1500 ECL gate sites. Today's ECL gate array technology would require several thousand gate arrays to implement the object processor pipeline.

3.1.2.3 Ullner

Ullner's architecture [ULLNER83] is much different from the previous two. The two previous architects strived to keep their processors as simple as possible so that each processor's cost would be minimized. Ullner takes a different tack by associating larger processors with each object.

Ullner's architecture logically consists of a binary tree of processing elements. The leaf processors are loaded with an object's world space coordinates and a viewing transformation. Each processor transforms its object into screen space, and in lock step with other processors, scan converts its object. The scan conversion is unique in that each processor determines which region of the current scan line its object intersects. This line segment is passed up the tree to a combine processor which merges the line segments together. The root processor emits a sequence of line segments that tile the scan line.

One of the features of Ullner's architecture is that it does not have to dedicate a large portion of its time to loading object descriptions into processors. Object descriptions are loaded just once. New views are generated by

changing the viewing transformation. This works best if the model is static. Even if it isn't, it is possible to only update the dynamic portions of the model. This selective updating of the scene model requires stronger ties between the modeling system and the viewing system.

3.1.2.4 Summary of Processor per Object Architectures

The three processor per object architectures presented here run along a common thread. Cohen and Demetrescu provided the seminal work in this area. They showed that VLSI could be put to good use to produce real-time visible surface graphics. Their architecture was very simple and had a few problems but it opened the door. The most notable of the problems were: (1) lack of support for antialiasing, and (2) a large overhead was required for initializing the state of the processor pipeline.

Weinberg's architecture directly addressed the antialiasing problem and provided a good solution. He was still plagued with having to allocate a considerable amount of time to initializing the processors.

Ullner's contribution seeks to reduce the amount of time spent loading the processors. He suggests using more powerful processors and distributing the model across the processors. The processors are responsible for transforming and clipping the model. Ullner's work doesn't really address handling dynamic models. There is room for future work in this area.

All three of these systems consist of pipelines of processors. The architectures proposed by Cohen and Demetrescu and by Weinberg make use of fairly linear pipelines while Ullner's makes use of a binary tree form. The time required to compute a scene can be written similarly for all three architectures as:

$$T_{frame} = T_L N + T_D N + T_P(Height * Width)$$

where:

T_L	time to load a processor
T_D	data propagation time
T_P	average pixel generation time
N	number of processors and objects
$Height$	number of pixels in the Y direction
$Width$	number of pixels in the X direction

We see that as we add more processors and more objects, more time is required to load the object data into the processors and more time is required for the first pixel's data to propagate through the pipeline. Both the loading time, $T_L N$ and the propagation delay $T_D N$ can be reduced in ways suggested by Ullner. Nevertheless, these terms tend to dictate overall performance in

real-time applications. Increasing the number of objects increases the overhead time which requires the average pixel generation time, T_P, to be reduced in order to produce the image in a frame time.

3.1.3 Processor per Pixel Architectures

Another result of the recent access to VLSI design and fabrication facilities is a class of architectures that have come to be known as processor per pixel architectures. These architectures associate a processing element with each pixel. These pixel processors work on a task in parallel and have the potential to achieve very high tiling rates.

3.1.3.1 Fuchs

In 1981, Fuchs devised the Pixel-Planes architecture [FUCHS81] and, in collaboration with others, has been developing it ever since. Pixel-Planes represents another member in the class of parallel depth buffer algorithms. This architecture utilizes a simple processor for each pixel in the image buffer.

The array of processors scan converts convex polygons in time proportional to the number of polygon edges. Each processor is able to evaluate the plane equation ($D = aX + bY + c$) and uses the resulting sum in different ways depending upon the different phases of the scan conversion process. The scan conversion process is divided into three phases: visibility determination, depth determination, and shading.

During the visibility determination phase, the state of all pixel processors is initialized to indicate potential visibility. The line equations of each polygonal edge are then presented to the pixel processors. The processors evaluate the line equation in parallel, each calculating D, which is interpreted as the distance of the pixel from the line. All processors to one side of the line (outside the polygon) disable themselves from all further calculations. By the time all of the polygons edges have been presented, only the pixel processors in the interior of the polygon are still marked as potentially visible.

In the next phase, processors calculate the depth at each pixel. The distance from each pixel to the viewer is encoded as ($Z = aX + bY + c$), and this equation is presented to the array of pixel processors. Only processors that are still marked as being potentially visible compare the computed depth with their stored depth. If the computed depth is closer than the stored depth, the stored depth is updated with the new value. Obscured pixels disable themselves from all further calculations.

The final phase calculates the intensity at each pixel. This is done by encoding each of the primary color intensities as: ($I = aX + bY + c$). Thus, three more equations are presented to the array of processors, and each of the pixel processors, still in the visible state, stores the result in their color

registers. This completes the polygon tiling process and the next polygon is started.

Fuchs's processors are relatively simple due to a clever partitioning of the logic needed to evaluate the line equation. Each chip consists of an array of processors surrounded on two sides by some logic. This logic serially calculates $aX + c_1$ for each column in the array. Similarly, the other logic tree calculates $bY + c_2$. In order to calculate D, each processor need only serially add the numbers presented on its row and column lines.

Fuchs estimates that a Pixel-Plane machine can achieve a throughput of about 1,000 polygons per frame time. This machine would perform as well as most of today's flight simulator engines. Drawbacks include its simple shading model and most notably the lack of antialiasing, which is a common drawback with depth buffer algorithms.

3.1.3.2 Whelan

This author also developed a processor per pixel architecture [WHELAN82]. The architecture was originally not intended for use in three-dimensional hidden surface graphics applications, but rather for use in two-dimensional VLSI CAD applications. The architecture is capable of scan converting an arbitrary-sized axis-aligned rectangle in constant time.

This rectangle tiling architecture probably represents the simplest processor per pixel architecture achievable. Its processing element is simply a RAM cell with two extra transistors that allow the RAM cell to be written when both row and column selects are asserted. Like Fuchs, Whelan partitioned the logic so most of it is shared and resides on the periphery of the RAM array in the form of specialized row and column decoders. Unlike ordinary RAM decoders, which assert only one output, these decoders assert a band of outputs. In use, the row decoder is loaded with the lower and upper Y values of the rectangle, and the column decoder is loaded with the corresponding X values. The RAM cells in the region defined by the intersection of the asserted row and column selects are within the rectangle and participate in the write cycle.

An adaptation of this algorithm would serve as a fairly decent tiling engine. Instead of tiling arbitrary rectangles, this new architecture would tile scan-line segments and could be used with many visible surface algorithms which first determine visible scan-line segments and then tile them. The scan-line architecture is illustrated in Figure 3.7 and requires no modification to the design of the conventional RAM cell. It requires modifying only the column decoder, which again asserts a band of outputs corresponding to the affected pixels. Gouraud shading can easily be implemented by organizing the chips such that a full color bank of pixels is resident on a single chip. Logic similar

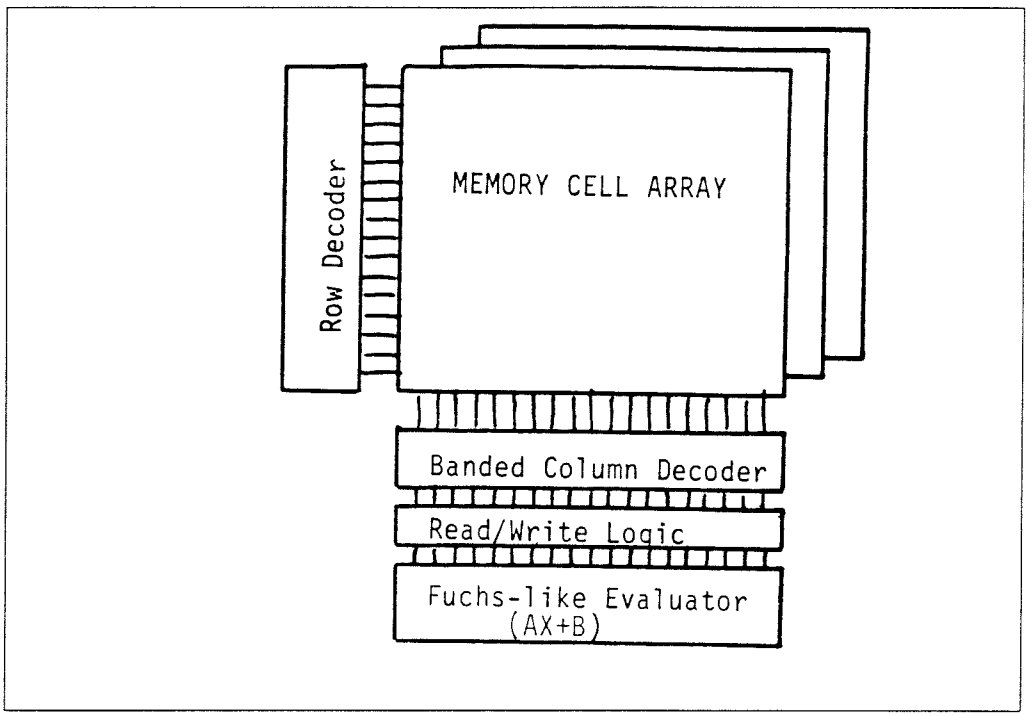


Figure 3.7: Whelan's processor per column tiling architecture.

to Fuchs's can be used to calculate $I = aX + c$, and the result can be stored in the memory cells.

3.1.3.3 Summary of Processor per Pixel Architectures

Fuchs's Pixel-Planes architecture addresses the entire visible surface determination problem by utilizing a distributed depth buffer algorithm. The processing elements required to implement this algorithm are fairly complex when compared to a RAM cell.

Whelan's architectures rely on very small processing elements. His second architecture is in effect a processor per column architecture, and therefore incurs no per pixel overhead.

These two approaches are at different ends of a spectrum. Fuchs finds himself working hard to reduce the size of his processing elements so that many more can be placed on a single die. His architecture will become economically feasible when a Pixel-Planes system can be implemented with a reasonable number of integrated circuits. Whelan's architectures may be economically feasible today but they cannot provide the same performance improvements as

Fuchs's. An architecture somewhere between the two will likely provide good performance at reasonable costs.

With processor per pixel architectures, performance depends directly on the number of objects in the scene. For Pixel-Planes, the time required to tile a polygon, T_{poly}, is:

$$T_{poly} = T_S + T_P(E + 4)$$

where:

T_S	per polygon setup time
T_P	basic calculation time
E	number of polygonal edges

Interestingly, performance does not depend on the number of processors. Likewise, performance is not dependent upon the projected area of a polygon.

Clearly, processor per pixel architectures achieve very high tiling rates for large polygons. Tiling rates, measured in pixels/second, decrease as polygons become smaller suggesting that these architectures are best avoided for overly complex scenes.

While much work has been done on processor per pixel architectures, major issues still need to be resolved. First and foremost, neither architecture provides any support for antialiasing. Without antialiasing, these architectures cannot provide the image quality that is expected of such systems. Secondly, these architectures provide for only the simplest of illumination models and provide no support for higher order effects like shadowing and reflection.

Two approaches might be taken in the further development of these architectures. First, they might be improved to overcome the drawbacks inherent in current designs. Second, applications might be found that aren't hindered by these drawbacks.

3.1.4 Conclusions

Three different uses of parallelism have been studied. Each use increases system performance. All three exhibit different characteristics that seem to indicate they will find a certain niche in which they excel. Table 3.1 compares the performance of these different uses of parallelism. Performance has been written both as a function of the number of processors, N_{proc}, and as a function of the number of objects, N_{obj}, in the scene.

The table indicates that all of the architectures require more time for scenes that contain more objects. This dependency is linear for the three architectures listed because they all utilize depth buffer algorithms for visible surface determination.

The table also indicates that performance improves as more processors are added to spatial subdivision architectures. Performance worsens as more

Architecture	Processors	Objects
Spatial Subdivision	k/N_{proc}	kN_{obj}
Proc. per Object	$k_1 N_{proc} + k_2$	$k_1 N_{obj} + k_2$
Proc. per Pixel	k	kN_{obj}

Table 3.1: Performance comparisons for various parallel architectures.

processors are added for processor per object architectures. This arises from pipeline loading and propagation times. The processor per pixel architectures show no performance dependency on the number of processors.

Both processor per pixel and processor per object architectures are claimed to offer performances of approaching 10,000 objects per frame time. This performance would be approximately an order of magnitude better than today's flight simulation engines.

Both of these architectures could be extended to provide higher performance. Both architectures are similar in that they utilize a depth buffer algorithm. Depth buffer algorithms allow objects to be tiled independently. This independence can be utilized to increase the performance of any depth buffer algorithm.

Given a depth buffer engine capable of performance ρ, performance 2ρ can be obtained by using two engines. Half of the objects are given to each processor. Each processor creates a resultant image. The resultant images are easily merged by selecting the pixel with the lowest depth value.

This technique works but has a severe problem. Aliasing will result when two images are merged together. Aliasing may or may not be worse than the aliasing present in the original images depending upon what antialiasing measures were taken. The current state-of-the-art in real-time simulation and animation requires that effective antialiasing measures be taken.

Spatial subdivision architectures provide a way of improving performance by adding more processors. All objects that cover a region are redistributed to a processor. This provides the needed information about a locality enabling effective antialiasing measures to be taken.

Spatial subdivision techniques improve system performance by changing the magnitude of the visible surface determination problem that must be solved by each processor. The other techniques improved their performance by increasing the amount of work accomplished per clock tick. Spatial subdivision architectures provide a way of obtaining performance gains that result from decreasing algorithmic complexity rather than from reducing gate propagation delays.

3.2 Simulations of Spatial Subdivision Architectures

Architectures that use spatial subdivision to reduce the total number of objects per processor seem to offer the greatest gains. By reducing the number of objects each processor must handle, these architectures provide a mechanism for drastically reducing the amount of time required by visible surface determination algorithms. In addition, the smaller number of objects in each processor reduces the demands on that processor's tiling engine.

The potential performance improvements that can be attained by using these spatial subdivision architectures can be easily estimated. If we assume that each processor implements a visible surface algorithm that requires time proportional to the number of objects distributed to each processor, then spatially subdividing the computation of a scene among N processors can improve performance by at most a factor of N.

Any distribution of objects to processors is likely to be non-uniform and would lower the performance gains accordingly. Even if we could achieve ideal performance gains, a performance improvement of two orders of magnitude would require one hundred processors.

These processors are not simple devices. If built today, they certainly would require at least one printed circuit board per processor. In the future we might expect this to reduce to several VLSI circuits. Few extremely large multiprocessors have actually been implemented. Considering the size of current multiprocessor implementations and other physical constraints on system size, we could expect spatial subdivision architectures to be limited to a few hundred processors. These several hundred processors would likely yield a performance improvement of two orders of magnitude.

Spatial subdivision architectures have one key performance advantage over other parallel architectures. Spatial subdivision architectures prescribe only a method of distributing objects to visible surface determination processors. How these visible surface determination processors implement their algorithms has not been discussed. Many of the previous parallel architectures address exactly this problem. Performance improvements of a parallel visible surface determination algorithm can be compounded with those provided by spatial subdivision to provide much higher performance improvements.

For example, if we were to couple a spatial subdivision architecture with the Pixel-Planes architecture, we might expect to achieve performance improvements of about four orders of magnitude over conventional architectures. In more concrete terms, Pixel-Planes architectures seem to be capable of offering slightly more performance than commercially available real-time simulation systems. We could then expect our compound system to achieve a performance gain of two orders of magnitude over today's real-time systems.

This ability of spatial subdivision architectures to compound their performance improvements with performance improvements of parallel visible surface determination architectures makes spatial subdivision architectures particularly attractive for use in very high performance graphics systems. Spatial subdivision architectures that do not employ parallel tiling architectures do not provide substantive performance gains over parallel tiling architectures, although they may provide other benefits.

The remainder of this section attempts to justify the performance gains I have cited for spatial subdivision architectures. This has been done by simulating the performance of various spatial subdivision strategies under differing load conditions. The following discussions present experimental methods, results, and conclusions.

3.2.1 Method

The performance gains offered by spatial subdivision architectures can be measured by parallel efficiency, the amount of parallelism actually delivered from a parallel architecture. In the case of spatial subdivision architectures, the parallel efficiency is controlled by two factors: processor utilization and object fragmentation. Both of these factors are highly dependent both on the method being used to spatial subdivide the image space and on the image that is being rendered.

Processor utilization is a measure of the percentage of time a processor is doing constructive work. In spatial subdivision architectures, it is likely that the distribution of objects to processors will not be uniform. Processors will have to wait for an individual processor to finish before proceeding with the computation of the next frame. Load balancing occurs when all processors require the same amount of computational time. Spatial subdivision methods that achieve processor load balancing provide higher processor utilization and contribute to higher parallel efficiencies.

Object fragmentation is a measure of the increase in the number of objects that must be handled by the visible surface determination processors. Object fragmentation occurs when an object spans processor boundaries. These objects must be either subdivided into smaller objects or replicated in adjacent processors. In either case, more objects are created. Spatial subdivision methods differ in the amount of object fragmentation they produce.

Optimizing parallel efficiency involves optimizing both processor utilization and object fragmentation. Certain spatial subdivision methods may compromise one of these factors in favor of the other. To understand more about the behavior of different spatial subdivision methods, the performance of certain spatial subdivision methods was simulated and certain factors were metered.

It was previously mentioned that the processor utilization is affected by both the spatial subdivision method and the scene being rendered. The importance of simulating the subdivision methods under realistic conditions was realized early on. As the previous chapter illustrated, realistic scenes are complex in nature and cannot be adequately simulated by simple means. For example, analysis of performance under uniform loading provides few useful insights that can be extended to real loading.

For this reason, the simulator was driven by data derived from views of real models. Computer generated images of models created by the author and others, including graphic art designers and computer scientists, were rendered with the author's *render* program. In addition to producing images, *render* also produces trace information that describes each polygon which its polygon tiler was requested to tile. This information is not an exact polygon description but adequately describes the bounding box of each tiled polygon.

A general purpose simulator for spatial subdivision architectures was written. This simulator views a multiprocessor architecture as a collection of rectangular image space regions. It then reads a *render* statistics file and clips each polygon's bounding box against each processor's image space region. A hit is recorded whenever an polygon's bounding box intersects a processor's image space region. This method properly simulates fragmentation since a polygon may fall into many processor's image space regions and the simulator will record a hit for each affected processor.

Simulation results are tallied at the end of each simulation pass. Five separate cost metrics were collected. Four of these cost metrics measure the amount of time required for the simulated computation. Each metric is normalized with respect to the time required by a conventional uniprocessor.

The first cost metric is the fraction of polygons handled by the processor responsible for the greatest number of polygons. This metric is justified by two factors. First, the faster algorithms require time proportional to the number of objects, and second, when computing a frame, one usually must wait until all processors have finished computing their subregions before being able to view the resulting image.

The second cost metric measures the average number of polygons distributed to the processors and divides this number by the total number of polygons. This metric is appropriate when one doesn't have to wait for all processors to finish before proceeding with the computation of the next frame.

These two metrics have an analog in pipelined systems. The first metric measures the total time required for the computation while the second metric measures the computation rate. In pipelined systems both measures are important and the computation rate often greatly exceeds the rate implied by the total computational time.

In a graphics environment, the first cost metric measures the time to compute a single frame, which was referred to as latency in this chapter's introduction. The second cost metric measures the average frame computation time when computing a sequence of frames. This corresponds to throughput.

Two additional cost metrics were calculated but proved to be of little use. These are similar to the first two metrics, but instead of assuming a visible surface determination algorithm with time cost proportional to the number of objects, these two metrics implement a $O(N \log N)$ cost metric. These metrics depend greatly on the value of N and as such it is not useful to compare results from the simulations of scenes of differing complexities. Because the first two metrics provided this capability, they were preferred over the latter two.

The fifth cost metric was a measure of average object fragmentation. The number of objects residing in all of the processors was added up and divided by the number of initial objects. This metric provides interesting insights into the behavior of the different subdivision methods.

3.2.1.1 Spatial Subdivision Methods

The multiprocessor architectures that were simulated fall into two classes. The first class of architectures make use of one-to-one mappings of object space regions onto processors. The second class of architectures make use of many-to-one mappings of object space regions onto processors. These can be thought of as virtual processors.

The non-virtual architectures were the most apparent and the literature reflects this. Practically all proposed architectures have been non-virtual, with the exception of [KAPLAN79]. I, too, approached this task with non-virtual architectures in mind and sought to achieve load balancing among processors via non-equal partitioning of the image space. The results will show that this is an approach that may deserve more attention.

More recently, it occurred to me that a better approach to load balancing might be to have each processor sequentially perform many small tasks rather than one large task. This assumption was based on a belief that it is easier to distribute many small tasks equitably among a group of processors than it is to distribute fewer large tasks. Further thought resulted in the notion of virtual processors, and the resulting architectures which fall into two categories: those that make use of topological information to perform virtual to physical processor assignments and those that make no use of topological information. Section 3.2.3 of the experimental results describes these architectures.

3.2.1.2 Scene Models

The decision of which scene models should drive the simulator was an important one. Considerations included: (1) having a selection of scenes that was

representative of typical scenes, (2) having a selection of scenes that varied in complexity between 1,000 and 100,000 polygons, and (3) having a selection of scenes that was free of bias.

All of these factors had to be compromised somewhat. Scenes that had been previously modeled at Caltech were neither representative of all of the scenes we would expect in state-of-the-art computer graphics animation nor were these scenes terribly complex. Most of these scenes consisted of between 1,000 and 10,000 polygons.

Bias presents itself in many forms. Foremost, I feared that my choice of scenes or my creations of models might be self-serving. It is clear that one can create scenes for which spatial subdivision works well and scenes for which it doesn't work at all. My fear was that I might pick scenes for which a certain type of subdivision worked better than others. To avoid this, I have tried to insulate myself from the design of the basic models.

Bias also exists in a more subtle form. Since models were created in a particular modeling language and scenes generated by a particular rendering program, it is unavoidable that the model will both take advantage of and be hindered by the limitations of the modeling and rendering environments. There was little I could do to avoid this type of bias. It is bias that is bound to be found in any similar experiment and bias that has to be tolerated yet understood.

My original simulations were run on a handful of scenes created by several people. The results were inconclusive due mainly to the fact that most of these scenes were rather simple. Overly simple scenes are best computed with a single processor. The work inherent in distributing a simple scene to many processors makes the proposition unworkable. Furthermore, I was not interested in the rendering of simple scenes, but in the rendering of highly complex scenes containing some 100,000 polygons.

Since it takes considerable work to generate scenes of 100,000 polygons, and since many such scenes would be needed, a new strategy evolved. One of the goals of the simulations was to study how increased scene complexity affected the performance of spatial subdivision architectures. I desired to readdress this task in a systematic fashion. This required forming a model of scene complexity.

Scenes complexity varies in many ways. There are two important ways of increasing a scene's complexity. One method is to add new objects to the scene. This has the effect of increasing the drawn area and may increase the average depth complexity of the scene. Certain attributes such as the distribution of polygon areas are likely to remain the same.

Another method involves the addition of detail to existing objects within a scene. This usually has the effect of keeping the drawn area relatively constant and not affecting the average depth complexity much. Most notably, the distribution of polygon areas changes greatly.

Together, these two paths to increased scene complexity provide a fairly rich way to describe image complexity. One can think of these two paths as basis functions in a two dimensional vector space. Points on the plane that can be represented by a linear combination of these two basis functions describe reachable image complexities.

Thus, in order that we might be able to draw some conclusions about how changes in scene complexities affect the performance of various spatial subdivision architectures, two scene models were chosen to drive the simulation.

The first scene model represents increasing image complexity by adding new objects to the scene. I started with the model of a X-Wing fighter which is illustrated in Figure 3.8*a*. This model was created by Chuck Esrock who was then a student at the Art Center College of Design. A program assembled a model consisting of a number of X-Wing fighters by placing X-Wing fighters in randomly selected cells in a 9 by 9 array. The random selection was used to avoid bias that might otherwise have been introduced by the author. A sequence of ten scenes was created. These range from around 1,000 polygons to more that 96,000 polygons.

The second scene model represents increasing image complexity by adding detail. The most expedient way of generating such a model seemed to be to generate views of a fractal landscape. Seven fractal landscape models were generated. They range from 700 polygons to more than 96,000 polygons. A view of one of these fractal landscapes is shown in Figure 3.8*b*.

These seventeen scenes involving more than 800,000 polygons were used to drive the simulations. The following sections report on the simulation results. Many days of VAX CPU time were required to compute these simulation results.

3.2.2 Non-virtual Multiprocessor Architectures

The non-virtual multiprocessor architectures can be divided into two categories. The first category makes use of regular subdivisions of the image space while the second category does not.

Three regular spatial subdivision algorithms were simulated and are illustrated in Figure 3.9. The first algorithm is referred to as subdivision into *columns* and is implemented as a one-dimensional subdivision of the horizontal axis. The scene may also be subdivided along the vertical axis into *rows* or may be subdivided along both axes into *rectangles*. Other regular subdivisions exist but were not simulated because they do not align with the pixel grid.

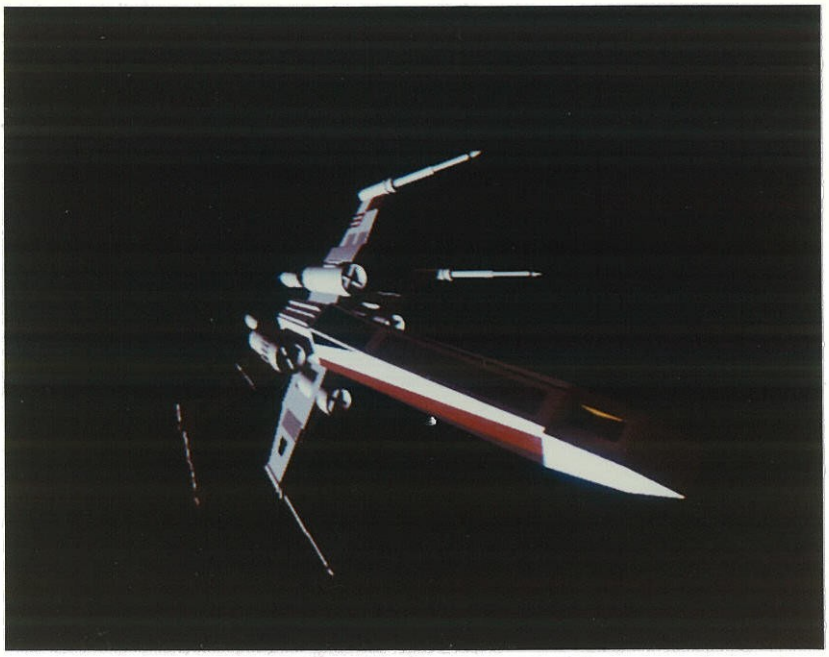


Figure 3.8a: X-Wing Fighter Model

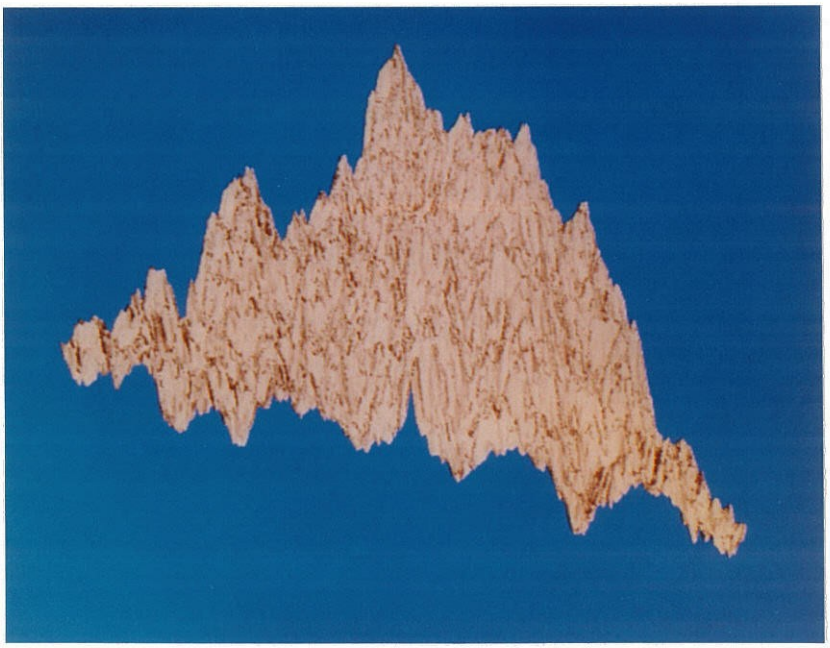


Figure 3.8b: Fractal Landscape Model

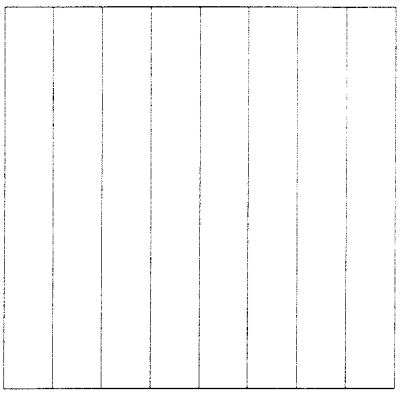


Figure 3.9a: *Columns*—Regular subdivision along the horizontal axis.

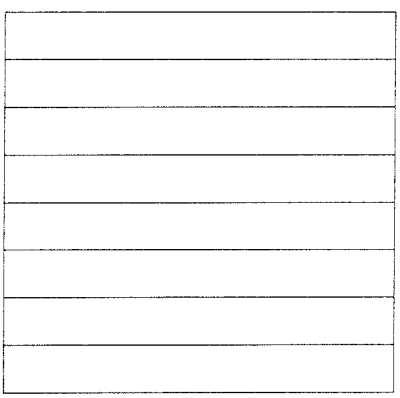


Figure 3.9b: *Rows*—Regular subdivision along the vertical axis.

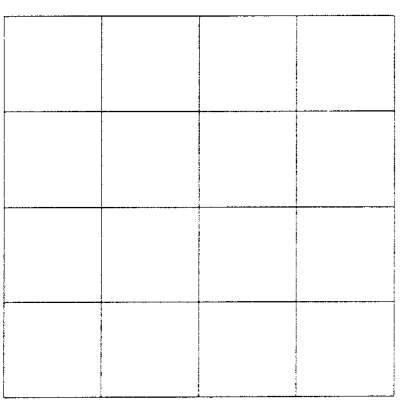


Figure 3.9c: *Rectangles*—Regular subdivision along both axis.

One irregular two-dimensional subdivision of the image space was simulated. The algorithm used to irregularly subdivide the image space attempted to achieve load balancing through a top down subdivide and conquer strategy. Figure 3.10 illustrates how this irregular two-dimensional subdivision technique recursively partitions the image space. At each recursion level, a median point is determined from the centroids of all objects within a region and the region is divided at that median point along one of the axis.

This algorithm, referred to as the *Median Cut* algorithm, attempts to put half of the objects into each subregion but doesn't properly achieve this goal since fragmentation occurs when objects cross the cutting plane. The Median Cut algorithm does not produce optimal partitionings of the image space but was selected as a representative member of this class of algorithms and should be thought of as providing a lower bound on the performance that is achievable with adaptive algorithms.

Each of these four spatial subdivision algorithms was simulated with varying numbers of processors for each of the seventeen test images. Simulations were performed over the range of 1 to 256 processors. The different algorithms required simulations to be performed at different sample points since the number of processors required by the different algorithms is related to the dimensionality of their subdivision technique. The one-dimensional algorithms can easily partition the image space N ways, whereas the two-dimensional algorithms can only partition the image space $\sqrt{N}$ ways and the median cut algorithm $\log N$ ways.

The simulation results are presented in Appendix A. Figures $A.1$–$A.17$, illustrate the simulation results for the ten X-Wing fighter images and the seven fractal landscape scenes. Each of these figures is presented in the same format to allow for easy comparison of the results. Each of these seventeen figures consists of four subfigures.

Subfigure (a) is a photograph of the scene for which the simulation results correspond. Subfigure (b) illustrates the four different spatial subdivision methods for which simulations were run. Each of these methods is illustrated for sixteen processors. Notice the figure which corresponds to the Median Cut algorithm since this is the only spatial subdivision which changes depending upon the scene content.

Subfigures (c) and (d) illustrate simulation results. Subfigure (c) plots the cost metric which corresponds to latency while subfigure (d) corresponds to the average frame computation time. These two figures are plotted similarly. The vertical axis corresponds to the cost metric and is accordingly labeled *Relative Time*. The horizontal axis corresponds to the number of processors and is so labeled. Four curves are drawn on each graph, one curve representing each of the four spatial subdivision techniques. These curves are identified by

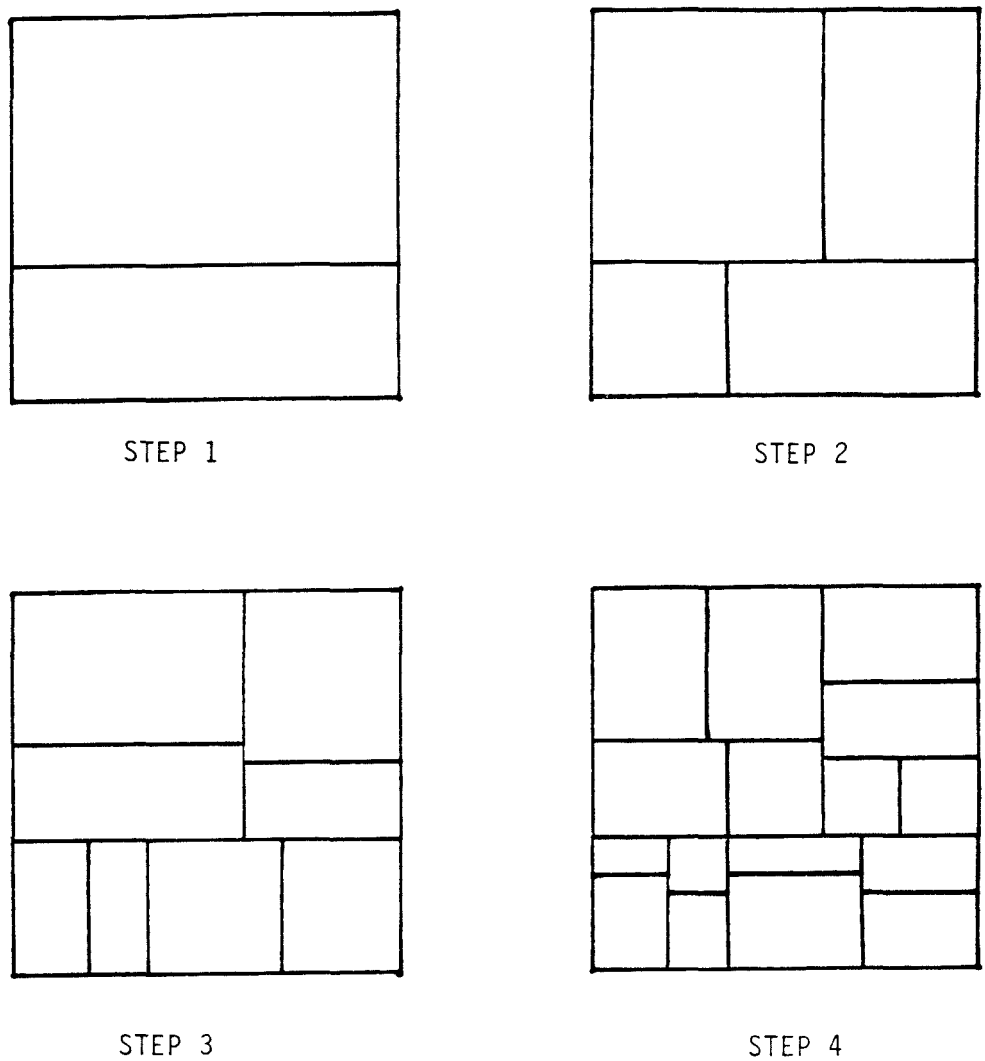


Figure 3.10: The Median Cut algorithm recursively divides the image space depending upon scene complexity.

textual labels on the right and by distinct symbols which are used to mark each simulation point.

A diagonal line is also drawn on each graph. This line represents the ideal performance curve. The distance from the ideal curve to the simulation curves indicates how good or bad the results are. Besides indicating how well a spatial subdivision technique performs for a given number of processors, these graphs indicate how performance is affected by increasing the number of processors.

Notice how performance indicated by *Relative Time* changes as the number of processors changes. Also notice how well the different spatial subdivision techniques track the ideal performance curve.

By comparing results from the different images one can easily see how performance changes with scene complexity. Notice the different behaviors for the X-Wing series and the fractal landscape series. Remember that these two series represent two different approaches to changing scene complexity.

Figure 3.11 consists of a plot of data extracted from the simulation results associated with the X-Wing fighter series (Figures $A.1$–$A.10$). This figure indicates how performance, this time measured as efficiency, changes with scene complexity. Efficiency has been calculated from the simulated *relative times* which represent scene computation time, or latency. The data presented is for processor configurations of sixteen processors.

The figure illustrates that the Median Cut technique provided the highest efficiencies across the entire spectrum of scene complexities. The Median Cut algorithm provided efficiencies well over 90% while the other methods struggled to reach 50%. We do notice a general upward trend in the efficiency curves, indicating that efficiency tends to increase as scene complexity increases due to the addition of new objects.

Figure 3.12 consists of a plot similar to the previous one. This one presents data extracted from the fractal landscape simulation results (Figures $A.11$–$A.17$). Once again the Median Cut technique dominates the picture, easily providing efficiencies in excess of 70%. The other methods never exceed 35%. Of these three methods, subdivision into columns appears best, while the rectangular subdivision appears worst. The *columns* method may work better than the other techniques since the fractals were generated by stretching polygons in the z direction which happens to correspond to the vertical axis in the views used to drive the simulator.

The relative flatness of the curves in Figure 3.12 is more interesting than their performance. This flatness suggests that increasing scene complexity by adding detail does not have much affect upon parallel efficiency. This is somewhat expected since the spatial distribution of objects is somewhat invariant when scene complexity is increased this way.

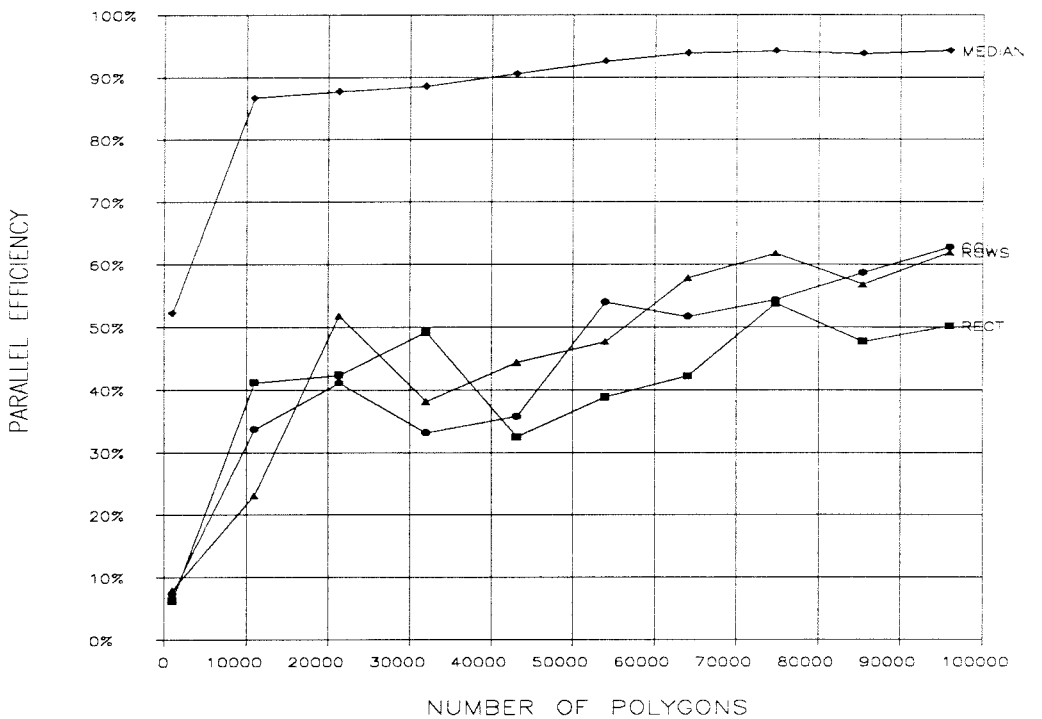


Figure 3.11: A plot of parallel efficiency versus scene complexity for the four non-virtual spatial subdivision strategies. This plot represents data from the simulations of 16 processors scan-converting the ten X-Wing images.

These results would suggest that computer graphics systems ought to employ the Median Cut algorithm to implement parallelism through spatial subdivision. Unfortunately, the Median Cut algorithm has several serious drawbacks. The most serious problem with this algorithm is that determining a partitioning involves performing many sorts on large amounts of data and probably requires as much computation as computing the image of the scene.

Another problem with irregular spatial subdivision is that it requires that the clipping processors be capable of dynamically changing their clipping boundaries. This isn't difficult to implement yet it will cost more to implement than static clipping boundaries.

A third problem associated with irregular spatial subdivision is that processors end up being responsible for screen regions of various sizes. One processor may end up tiling nearly all of the image's pixels while others tile very few pixels. This behavior requires that each processor must be designed to handle the worst case where it must tile the entire screen at the video update rate. The regular spatial subdivision techniques allow processors to tile pixels

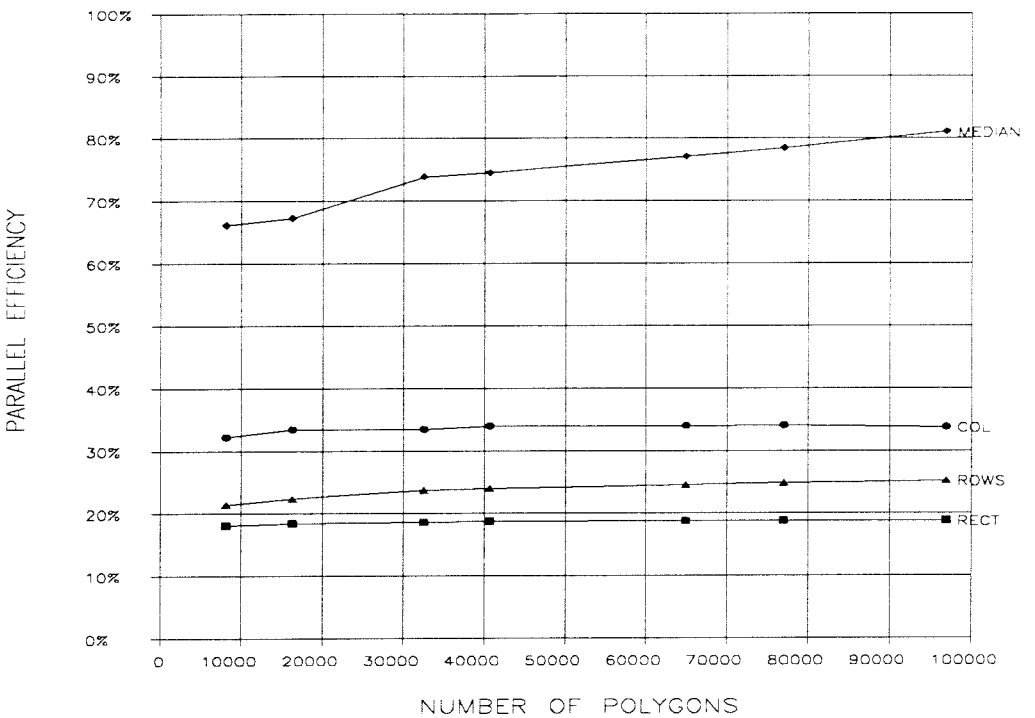


Figure 3.12: A plot of parallel efficiency versus scene complexity for the four non-virtual spatial subdivision strategies. This plot represents data from the simulations of 16 processors scan-converting the seven fractal landscape images.

at much slower speeds since they only tile a fixed fraction of the image's pixels. As an example, a processor in a sixteen processor Median Cut system may have to be designed to tile pixels sixteen times faster than the tiling engines used by the regular spatial subdivision methods. The regular spatial subdivision methods have more time to tile each pixel and can afford to do a better job of it.

Irregular subdivision techniques like the Median Cut algorithm can use feedback techniques to adapt to a scene. An initial guess can be made as to how to subdivide the image space. After the objects have been distributed, a processor may determine how uniform the distribution of objects is and based upon this information it may select the next image space partitioning. Such techniques have been proposed [DIPPÉ84] and deserve further investigation.

Because of the high costs associated with irregular subdivision strategies, alternative techniques were looked into as a way of improving the performance of the regular spatial subdivision strategies. The following section reports on this work.

3.2.3 Virtual MultiProcessor Architectures

The virtual multiprocessor architectures that were simulated all make use of static partitionings of the image space. These partitions are also statically mapped onto physical processors. These restrictions were applied for two reasons. First, virtual architectures were studied to see how much improvement they can provide over the non-virtual architectures. Thus, the partitions must exhibit a strong correspondence to the partitions employed by the non-virtual architectures. Second, static partitionings of the image space are less expensive than dynamic partitionings and require no *a priori* knowledge about how the image space should be partitioned.

Static assignments of image space regions to physical processors are also less expensive than dynamic techniques because the system does not have to provide for hardware that is capable of dynamic changes and no decision has to be made about how to associate image space regions with processors.

It seems clear that dynamic partitionings and dynamic mappings of image space regions onto physical processors may provide better performance. It should be pointed out though that there may be considerable advantages to having adjacent image space regions reside in the same or adjacent physical processor. In fact, the shadowing algorithm presented in the next chapter relies heavily upon this.

Six virtual spatial subdivision methods will be studied. These are best described as one- and two-dimensional subdivisions of the image space and as tessellations of various degrees of the image space.

Figure 3.13 illustrates the regular one- and two-dimensional subdivisions of the image space. Since these partitionings are similar to those simulated for non-virtual processors, we would expect their performance to be very similar to the prior case. More precisely, we expect to achieve a certain amount of parallelism which can be predicted from the results of the non-virtual simulations. In addition, we expect to have to pay a certain amount for having to simulate a parallel machine on a sequential machine. This increased cost corresponds directly to the object fragmentation occurring in the virtual processors that map onto a physical processor.

The one- and two-dimensional tessellations appear to be much more interesting. They are illustrated in Figure 3.14. The interesting aspect of these organizations is that they map topological regions that are far apart onto the same processor. There is hope that this property has a load balancing effect on the system. Similar schemes have been suggested to improve tiling performance [FUCHS79] [CLARK80].

All seventeen scenes were once again simulated. These simulations were performed for various numbers of physical processors at various ratios of virtual

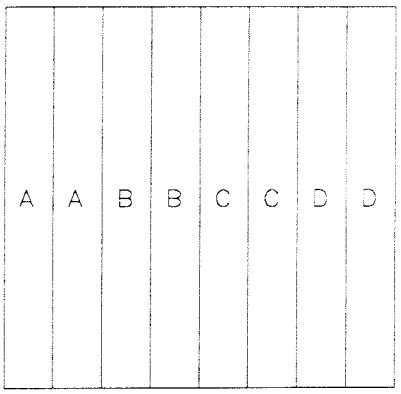


Figure 3.13a: A regular virtual horizontal subdivision. Eight virtual processors are shown mapped onto four physical processors.

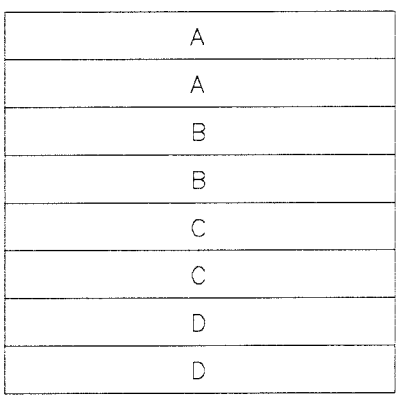


Figure 3.13b: A regular virtual vertical subdivision. Eight virtual processors are shown mapped onto four physical processors.

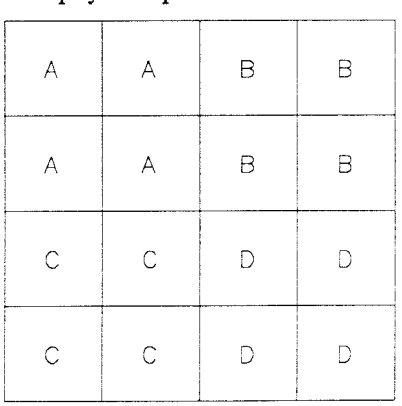


Figure 3.13c: A regular virtual rectangular subdivision. Sixteen virtual processors are shown mapped onto four physical processors.

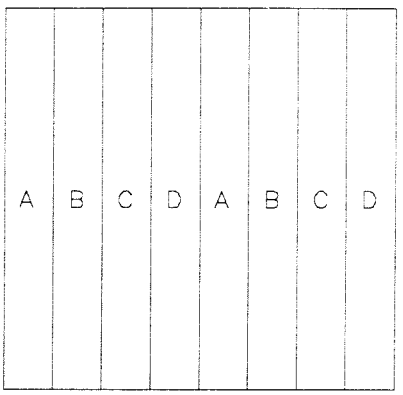


Figure 3.14a: A tesselated virtual horizontal subdivision. Eight virtual processors are shown mapped onto four physical processors.

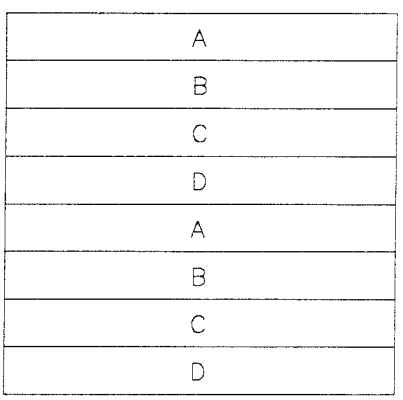


Figure 3.14b: A tesselated virtual vertical subdivision. Eight virtual processors are shown mapped onto four physical processors.

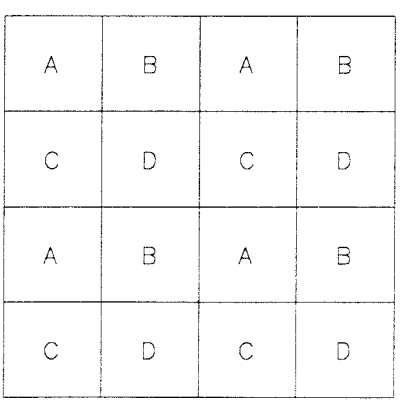


Figure 3.14c: A tesselated virtual rectangular subdivision. Sixteen virtual processors are shown mapped onto four physical processors.

to physical processors. The simulations yielded families of curves for each virtual spatial subdivision method.

The simulation results are illustrated in Appendix A. Figures *A*.18 through *A*.34 illustrate the simulation results for the ten X-Wing fighter images and the seven fractal landscape scenes. Only the data that represents image computation time (latency) has been plotted. Each figure is divided into six subfigures. Subfigure (a) represents the results for the Rows method. Subfigure (b) represents the results for the Tessellated Rows method. Subfigure (c) represents the results for the Columns method. Subfigure (d) represents the results for the Tessellated Rows method. Subfigure (e) represents the results for the Rectangle method. Subfigure (f) represents the results for the Tessellated Rectangle method.

Notice that when the ratio of virtual to physical processors is equal to one, the subdivision method is identical to the one used in the non-virtual section and the performance curves are identical. These curves can be used to compare the performance of the virtual techniques with the non-virtual techniques. When curves fall below the non-virtual curve, the virtual technique performs better than the non-virtual technique.

The data from the preceding plots is terribly difficult to capture in a brief reading. Data can be extracted from these simulation results and plotted in a more concise form. Figure 3.15 plots efficiency versus scene complexity for the X-Wing fighter series. It is similar to Figure 3.11 but contains three additional curves which represent the performance achievable with the virtual schemes. Curves were plotted for *virtual rows*, *virtual columns*, and *virtual rectangles*. Each of these curves is drawn for a fixed number of processors, sixteen in the case of Figure 3.15. Each point on a curve was computed by choosing the best result from both tessellated and nontessellated strategies over all the computed ratios of virtual to physical processors. Although both tessellated and nontessellated strategies were considered, the tessellated strategies always performed best.

Figure 3.15 illustrates that the virtual processor technique provides better performance than the non-virtual schemes. We notice a great improvement for rectangular subdivision in a virtual environment. This occurs because the rectangular subdivision technique fractures fewer objects than the other two techniques.

All of the virtual schemes easily made it above the 50% mark and the rectangular subdivision scheme made it into the 70% region. Once again, the curves show an upward trend indicating that efficiency improves as scene complexity increases due to adding more objects.

Figure 3.16 illustrates the performance of virtual and non-virtual spatial subdivision techniques for the fractal landscape series. Again we see a marked

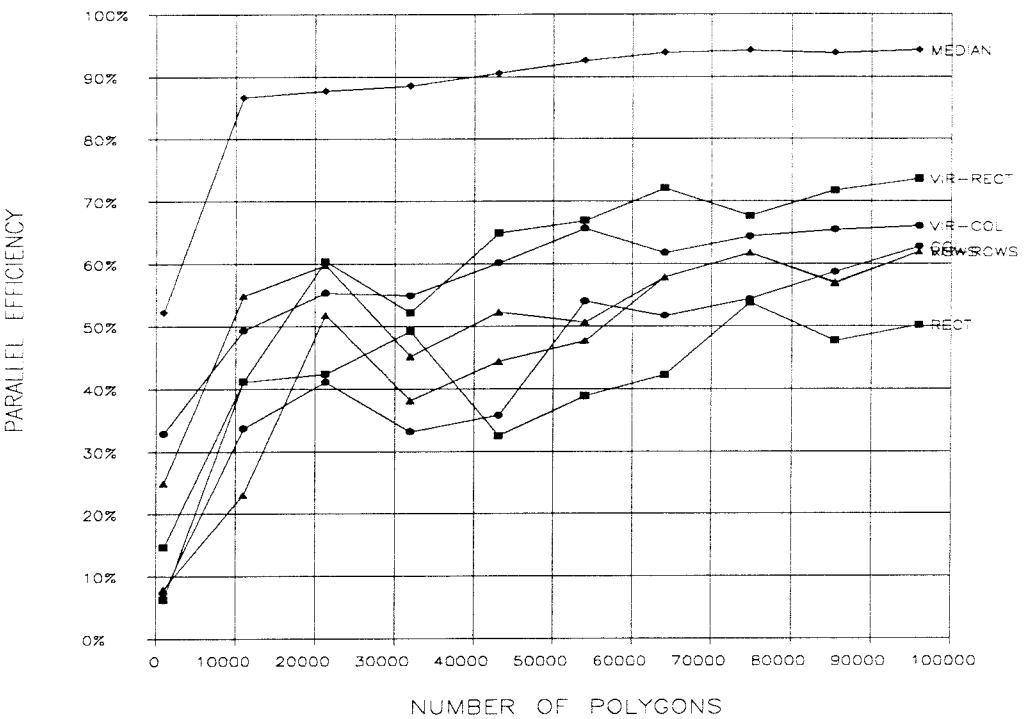


Figure 3.15: A plot of parallel efficiency versus scene complexity for the four non-virtual spatial subdivision strategies and the three tesselated virtual spatial subdivision strategies. This plot represents data from the simulations of 16 processors scan-converting the ten X-Wing images.

improvement for the virtual techniques. Notice that the *virtual columns* technique provided performance comparable to that provided by the Median Cut algorithm. Also note that the *virtual rectangles* technique improved from less than a 20% efficiency to well over 50%. As was the case for the non-virtual curves, the virtual curves are relatively flat. This suggests that efficiency does not change much as scene complexity changes due to changing the level of detail.

We have seen that certain scenes may prefer subdivision into rows while others such as the fractal landscape series may prefer to be subdivided into columns. The rectangular subdivision approach seems to offer some insensitivity to the preferred subdivision axis of the scene. The previous results indicate that virtual rectangular spatial subdivision can offer respectable efficiencies in excess of 50%. Such efficiencies make it practical to build graphics engines that utilize spatial subdivision techniques.

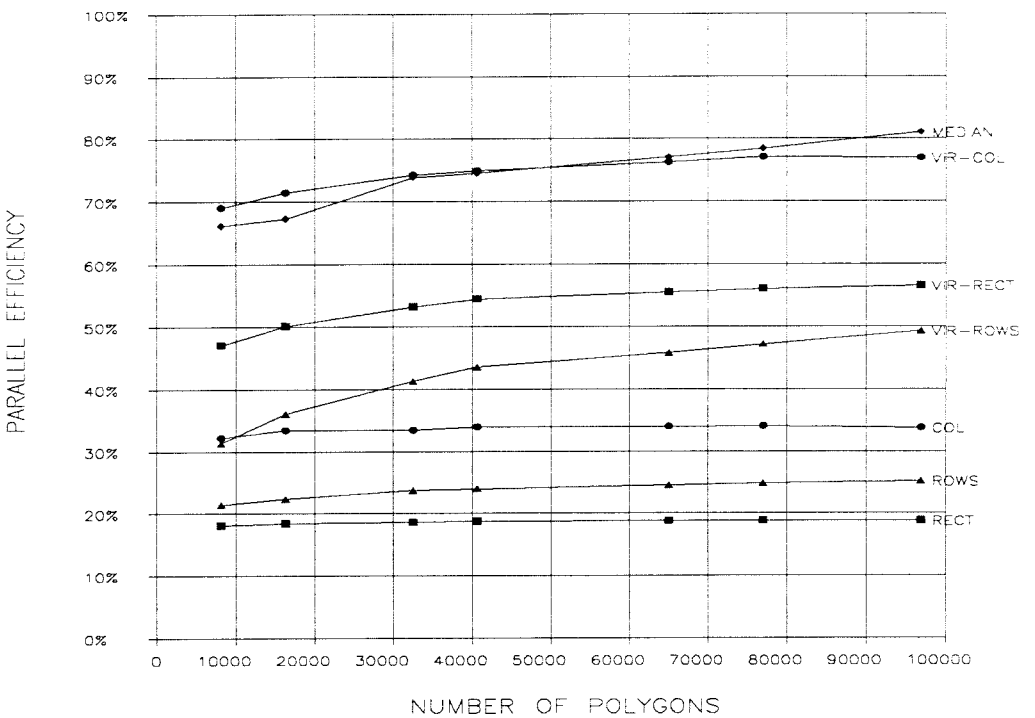


Figure 3.16: A plot of parallel efficiency versus scene complexity for the four non-virtual spatial subdivision strategies and the three tesselated virtual spatial subdivision strategies. This plot represents data from the simulations of 16 processors scan-converting the seven fractal landscape images.

All of the previous plots were illustrated for an array of sixteen processors. This number was chosen for a good reason. Proponents of processor per pixel and processor per object architectures have claimed that their architectures are capable of performance approaching 10,000 polygons per frame time. A goal of this research was to study how to build a machine capable of rendering 100,000 polygons per frame time. With an engine capable of rendering 10,000 polygons per frame time, and with a spatial subdivision architecture that provided decent efficiencies, an array of sixteen processors should be capable of approaching the goal of 100,000 polygons per frame time.

Nevertheless, it is important to understand how the number of processors used in the spatial subdivision strategy affects the performance of the machine. Figures 3.17 and 3.18 plot efficiency versus scene complexity. Each curve represents a different number of processors used in the subdivision strategy. All curves represent virtual rectangular subdivision. Figure 3.17 represents data

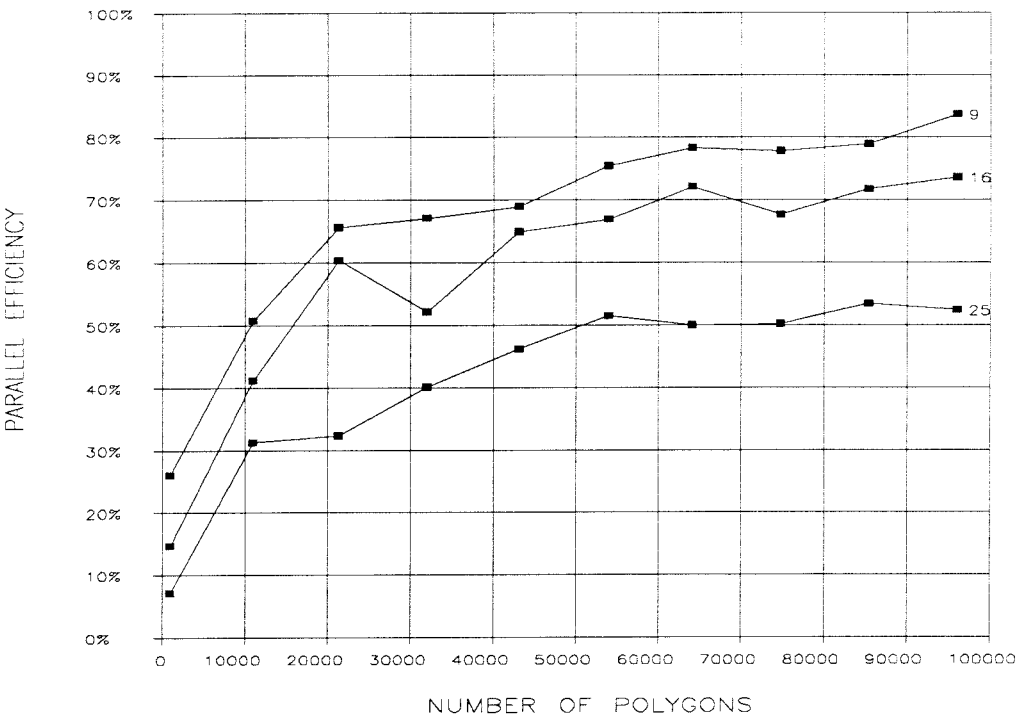


Figure 3.17: A plot of parallel efficiency versus scene complexity using the virtual tesselated rectangular spatial subdivision strategy. The different curves represent different numbers of physical processors. This plot represents data from the simulations of the ten X-Wing images.

from the X-Wing series of images. Figure 3.18 represents the fractal landscape scenes.

These figures illustrate what can easily be seen in the simulation results. As we decrease the number of processors, we improve efficiency. A uniprocessor provides us with 100% efficiency. Increasing the number of processors decreases efficiency. Efficiency decreases more slowly than the number of processors increases.

3.3 Conclusions

This chapter has studied various ways of utilizing parallelism to improve the performance of computer graphics systems. The behavior of processor per object and processor per pixel architectures limit their performance to about 10,000 polygons per frame time. This is consistent with the claims of the proponents of such architectures.

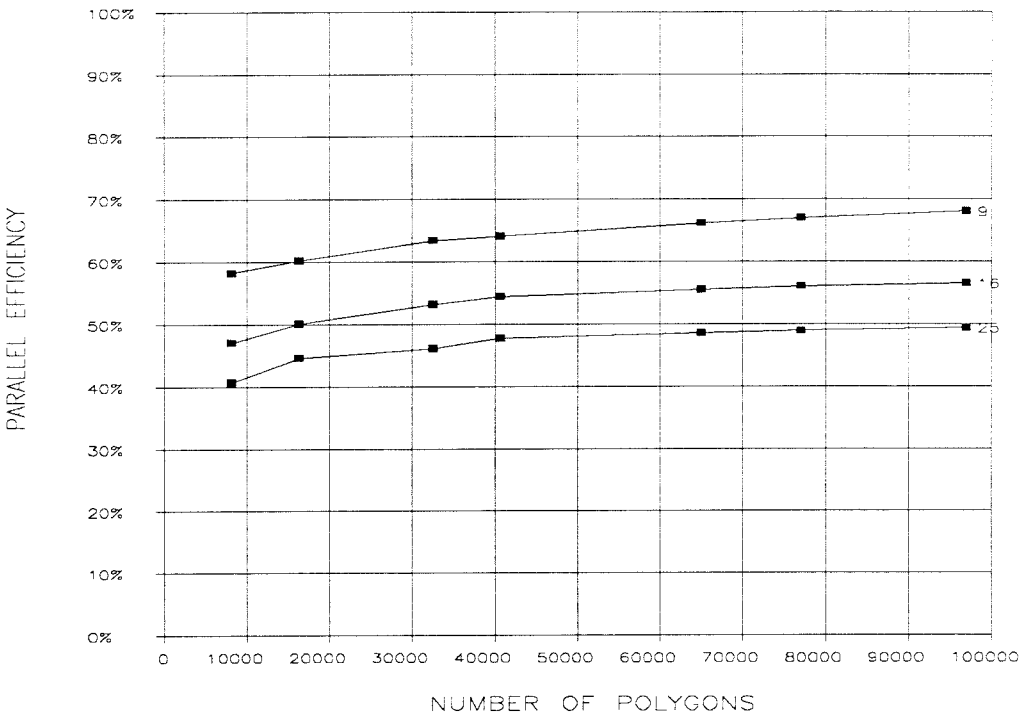


Figure 3.18: A plot of parallel efficiency versus scene complexity using the virtual tesselated rectangular spatial subdivision strategy. The different curves represent different numbers of physical processors. This plot represents data from the simulations of the seven fractal landscape images.

Analysis indicated that spatial subdivision architectures could provide a method to achieving higher system performance figures. Simulation of different spatial subdivision strategies indicated that a virtual rectangular spatial subdivision technique holds much promise.

Virtual rectangular spatial subdivision provides performance that is likely to improve as scene complexity increases. It also provides performance that can be improved by adding more processors. This parallelization technique is the foundation for the ANIMAC architectures. Details of these architectures are discussed in the following chapters.

4

A Partitionable Shadowing Algorithm

Shadows arise in scenes when objects occlude light rays and in doing so prevent other objects from being illuminated. Shadows form an important part of our every day visual environment. Shadows may enrich a scene and at the same time decrease visual discernibility.

Shadows enrich scenes by providing additional information. Shadows can greatly improve depth perception by providing many additional cues as to which objects lie in front of others. Information about relative distances between objects may often be inferred from their shadows.

Shadows can also mask objects making them less discernible, if visible at all. Crow [CROW77A] points out that this aspect of shadowing can have important implications upon task training and performance. Certain tasks, such as space craft manipulations, may not be performable because objects may be hidden in cast shadows. Crow's suggestion is that visual simulations of certain tasks need to include accurate shadowing effects in order to determine whether a protocol is feasible. Failure to accomplish a protocol during a space shuttle mission could easily result in the loss of millions of dollars.

Most commonly available computer graphics systems are not capable of generating scenes with shadows. Several methods for producing realistic shadow effects have been available to implementors of computer graphics systems during the past decade. Due to the increased computational requirements of these algorithms, few software implementors seem to have implemented shadowing and there have been no hardware implementations capable of real-time performance.

The previous chapter suggested that multiprocessor architectures might be used to speed up the rendering of visible surfaces. In order to compute images with shadowing effects on a multiprocessor, we must develop a visible surface algorithm that makes use of reasonably local data. Heavy use of non-local data would require complex and expensive communications networks and in the end would limit the usefulness of the algorithm by making it economical for only the smallest networks.

Shadow calculations are by their very nature non-local since an object far away can cast a shadow upon another object. Figure 4.1 illustrates that even objects outside of the field-of-view can cast shadows on visible objects. Thus localizing the transfer of information in a shadowing algorithm is a non-trivial problem and deserves attention.

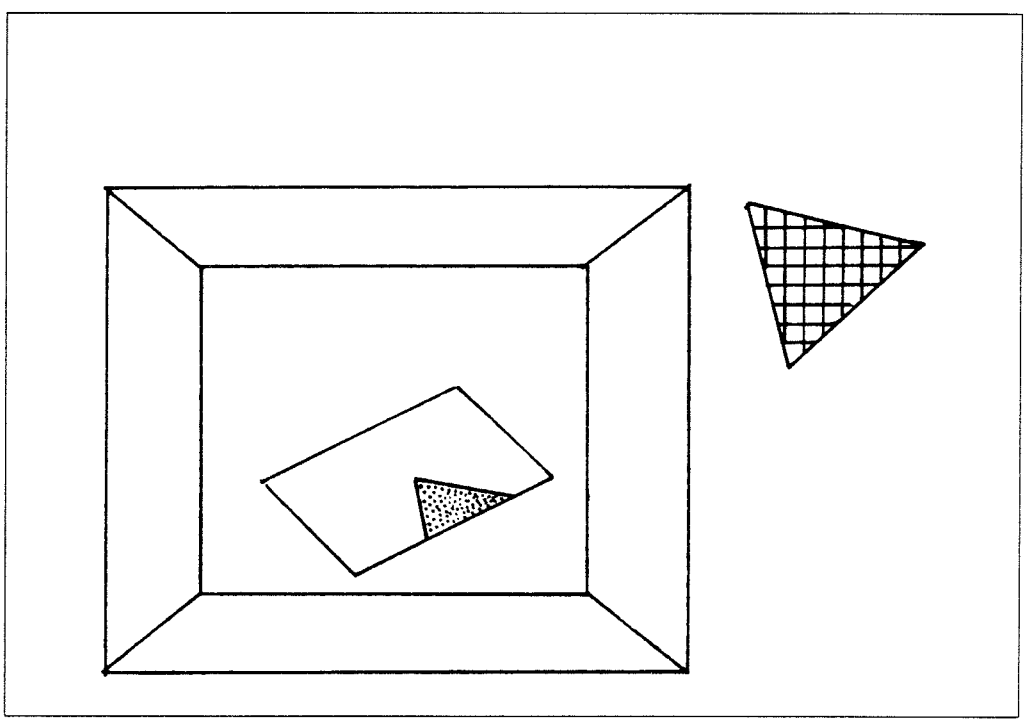


Figure 4.1: Shadowing is a non-local effect. Even objects outside of the observer's field-of-view may cast visible shadows.

This chapter deals with shadowing algorithms by first presenting an overview of previous work in this area. These algorithms are then examined as to how they would behave in a spatial subdivision multiprocessor environment. Finally, a new shadowing algorithm is presented that is tailored to this environment.

4.1 Previous Work

Sutherland *et al.* [SUTHER74], in their seminal taxonomic survey, divided hidden surface algorithms into three main categories: object space algorithms, image space algorithms and list priority algorithms. Object space algorithms attempt to solve the hidden surface problem as accurately as possible and are generally limited by the precision of the computer's numeric representation. On the other hand, image space algorithms make great use of the limited resolution of display devices to reduce the total amount of computation. List priority algorithms operate in both realms.

I have tried to use Sutherland's notation for categorizing shadowing algorithms. Shadowing algorithms generally consist of two phases. The first phase determines the extent of potential shadow boundaries, and the following phase performs an augmented hidden surface algorithm that makes use of the shadow boundary information when calculating pixel intensities. The type of algorithm used for the second phase can be quite independent of the first phase algorithm. For example, either an object or an image space algorithm can be used as a second pass algorithm. Since the first pass algorithm can be relatively independent of the second pass algorithm, I have categorized shadowing algorithms solely on the methods they utilize to determine potential shadow boundaries.

Object space shadow algorithms attempt to solve for shadow boundaries exactly, independent of the resolution of the final image. Image space shadow algorithms may also solve for shadow boundaries in manners independent of the resolution of the final image, but they do so in a discrete manner. Shadow boundaries are calculated only to some predetermined resolution.

4.1.1 Object Space Shadowing Algorithms

The vast majority of shadowing algorithms fall into the object space category. These include algorithms proposed by Appel [APPEL68], Bouknight and Kelly [BOUKNI70], Crow [CROW77A], Atherton *et al.* [ATHERT78], and Brotman and Badler [BROTMA84]. In addition to these algorithms, object space algorithms include the currently in vogue ray tracing algorithm, which in its recent incarnation traces back to Whitted [WHITTE80] and has been employed by many people since then.

4.1.1.1 The Shadow Volume Algorithm

Crow [CROW77A] proposed that projected shadow polygons be added to the environmental description. Shadow polygons are treated like other polygons by the hidden surface algorithm except that they are not visible and serve only to determine which portions of visible surfaces lie in shadow.

The basis for Crow's algorithm rests on the observation that an object casts a shadow over a well defined region of space, which he calls a shadow volume. Figure 4.2 illustrates the shadow volume that an object might cast. Shadow volumes encompass all of the space that the object obscures from the light source's point-of-view. Such a volume is semi-infinite and can be clipped against the viewing volume to yield shadow polygons.

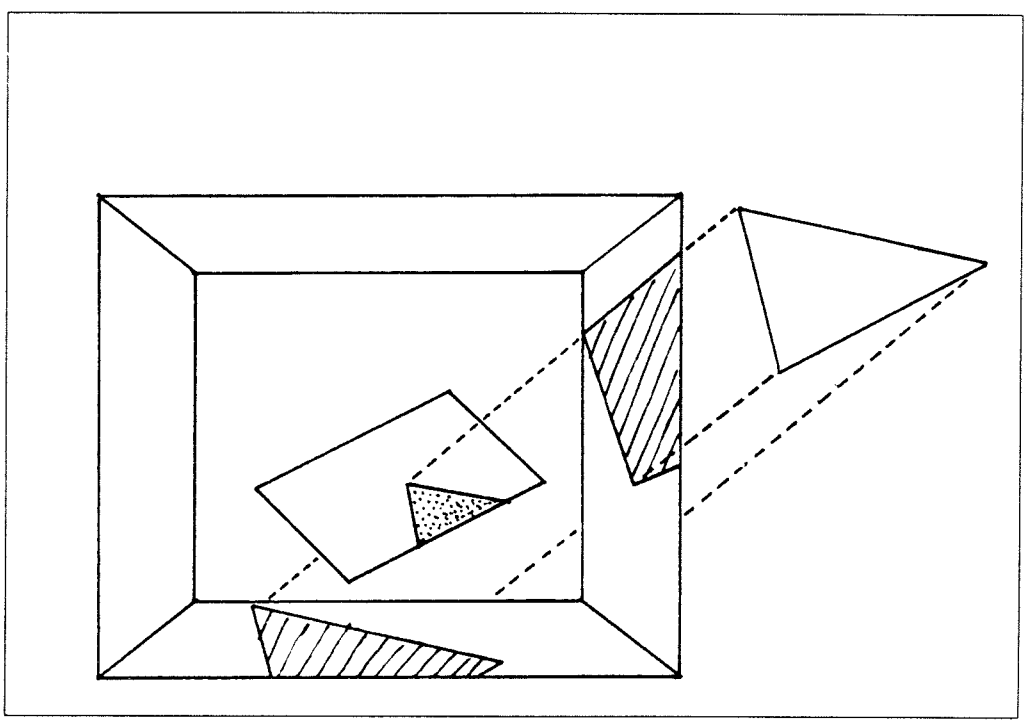


Figure 4.2: An object shadows any object within its shadow volume.

Shadow polygons are used during the hidden surface algorithm to imply the presence of a shadow. If a portion of a visible surface lies behind a frontfacing shadow polygon and in front of a backfacing shadow polygon, that portion of the visible surface lies in shadow.

Since shadow polygons are treated the same as regular objects, they may be clipped against the hither and yon clipping planes. This clipping action may remove the front or back faces of a shadow volume. Crow precisely describes the criteria for determining whether a point lies in a shadow volume by stating that "surfaces are shadowed whenever they lie in front of a backfacing frontmost shadow polygon or the surface depth count is such that more frontfacing than backfacing shadow polygons have been pierced."

One of the advantages of Crow's shadow volume technique is that shadow polygons need only be computed once for a particular environment. If the light source and the objects remain stationary, the environment may be viewed from any orientation and correct shadowing effects will be observed. Thus, it is possible to view the computation of shadow polygons as an additional cost incurred during data base creation for static environments. For dynamic environments, this cost must be incurred whenever the object data base or the lighting environment change. This may happen seldomly or frequently, possibly requiring recomputation for each frame.

Shadow volume computations increase the size of the environmental data base. If all objects are allowed to cast shadows many more polygons may have to be added to the environment's description. A worst case analysis requires an additional shadow polygon for each edge in the data base. This would seem to indicate that the total number of polygons in the data base would grow by at least a factor of four since a triangular polygon would require an additional three shadow polygons.

Fortunately, shadow polygons do not need to be cast for each polygonal edge. If the data base is properly structured, shadow polygons can be cast only for the contour edges. Deriving shadow volumes from object contour edges may greatly reduce the number of shadow polygons that need to be generated. The number of shadow polygons can be reduced further by taking into account that if the shadow volume cast by one object encompasses the shadow volume cast by another object, the latter shadow volume need not be cast since it can have no affect.

While the addition of shadow polygons may increase the size of the environmental data base at a rate that has a less than linear dependence upon the number of objects in the data base, other costs may be incurred. Crow pointed out that the shadow polygons may greatly increase the average depth complexity of the image. He mentioned that Sutherland [SUTHER74] has observed

that "increased depth complexity may well severely hamper the performance of scanning algorithms."

Finally, since Crow's shadow polygon method entails modifications to the environmental data base, the computation of shadow polygons needs to be done prior to distributing objects to processors in a multiprocessor that relies upon spatial subdivision. Parallelism can be utilized in the data base back end processor to rapidly generate these shadow polygons, but nonetheless, this process may be very time consuming for large environmental data bases.

4.1.1.2 The Shadow Polygon Algorithm

The shadow polygon algorithm has been proposed by Atherton, Weiler, and Greenberg [ATHERT78]. Although its name is similar to terminology used to discuss Crow's shadow volume algorithm, it is quite different in nature and should not be confused.

The shadow polygon algorithm employs a two-pass approach to generating shadowed scenes. During the shadowing pass, descriptions of the object and lighting environments are read in and an object environment description that has been augmented to include shadow descriptions is emitted. This output description has precisely the same form as the input description and can be used by a number of hidden surface algorithms to generate shadowed scenes without any modification to the algorithms.

Shadows are generated as detail polygons, which are used only to determine pixel illumination and thus do not play any role in visible surface determination. The authors claim that an important advantage of their approach is that it produces these shadow polygons explicitly. They claim that the output from their algorithm can be used for purposes other than display. For example, shadow polygons might serve as input to various energy analyses that can be useful in architectural engineering and design.

Shadows are determined by creating a hidden surface view of the object environment as viewed from the light source. The polygon area sorting visible surface algorithm of Weiler and Atherton [WEILER78] is used to generate visible polygons. These visible polygons are then added to their source polygons as surface detail.

The authors also illustrate that their technique can be used to model the effects of multiple light sources. With their technique, the data base is simply run through the shadowing program for each light source.

Like Crow's algorithm, this one manipulates the environmental data base. As such, it works best for static object and lighting environments, allowing multiple views to be generated without further data base modification. Unlike Crow's algorithm, this one requires substantial work to make the data base modifications. This additional work does provide an advantage in that

the modified data base does not alter the average depth complexity of the scene. Once the data base has been modified, views of shadowed scenes can be computed much faster with this technique than with Crow's.

4.1.1.3 Ray Tracing Algorithms

In recent years, ray tracing algorithms have been used to create some of the most realistic synthetic images. These images illustrate reflection, refraction, and shadowing effects. The ray tracing algorithm is very simple, but unfortunately it typically requires a great deal of time to compute an image.

Figure 4.3 will be helpful in understanding the ray tracing algorithm. As the figure illustrates, the ray tracing environment exists in a three-dimensional modeling space. Within this space are an observer, a viewing rectangle, a collection of objects, and one or more light sources. Different views of the objects can be selected by repositioning the observer and the viewing rectangle through which the observer views the world.

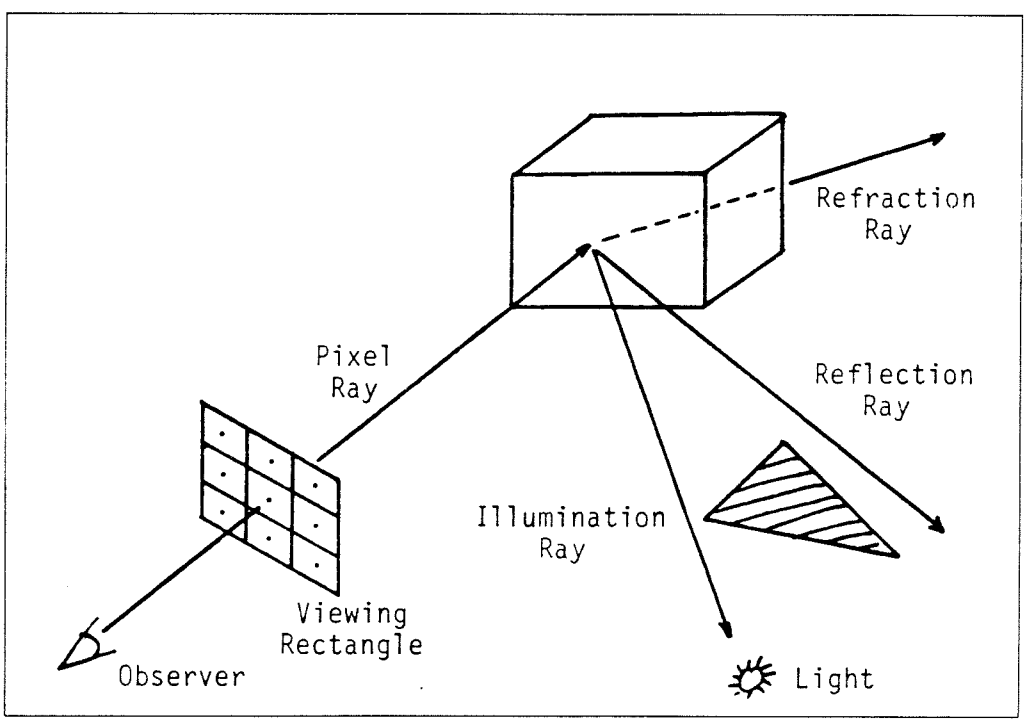


Figure 4.3: A ray tracing environment.

The raster display is mapped onto the viewing rectangle. Pixel centers can be thought of as points in the plane of the viewing rectangle and within the boundaries of the rectangle. The color of a pixel is determined by casting a *pixel ray* from the observer's position through a pixel center. The collection of objects is searched to find if the ray intersects any of the objects and if so, which of these objects lies nearest to the observer. If the ray intersected an object, this object is visible at the pixel and must be colored, otherwise the pixel can be colored according to some background coloring algorithm.

If reflection and refraction effects are not needed, the pixel coloring is simplified. The color of the pixel depends on properties of the object, how the object is illuminated, and how the object is being viewed. Color properties can be associated with objects in a variety of ways, none of which is important in the context of this discussion. Illumination is determined by casting an *illumination ray* from the visible surface towards each light source. Once again all objects are searched to find out whether any object might intersect the illumination ray and prevent the light source from illuminating the visible surface. Once illumination has been determined, the pixel can be colored using information derived from the pixel and illumination rays and object color properties.

If reflection and refraction are desired, a recursive procedure results. From the visible surface, two additional rays are cast. The *reflection ray* is cast in the appropriate reflection direction and the *refraction ray* is cast in the appropriate refraction direction. The same algorithm is applied recursively to associate pixel color contributions with these rays. The depth of recursion is usually limited to some maximum to insure that the process terminates.

While the algorithm appears rather simple, it is time consuming. The basic algorithm outlined here must be repeated for each pixel or subpixel if antialiasing measures are being taken. Interesting scenes usually consist of at least one quarter of a million pixels and often require the casting of millions of pixel rays since antialiasing steps must be taken.

Each pixel ray may result in the generation of 3^d additional rays, where d is the recursion limit. For each ray that is generated, the entire object data base must be searched for potential object-ray intersections. A 512 by 512 image of a scene composed of 1,000 objects often results in billions of object-ray intersection calculations. Ray tracing requires much time to compute single images on the fastest computers.

Several researchers have proposed parallel machine architectures for ray tracing applications. These include Ullner [ULLNER83] and Dippé [DIPPÉ84] both of which were reviewed in the Chapter 3. Both of these architectures speed up the ray tracing algorithm by (1) implementing an object-ray intersection algorithm in hardware and by (2) pruning the list of objects which must be

searched to those objects which are capable of intersecting the ray. Even with hardware support, neither of these authors has suggested that their parallel architectures are capable of real-time performance.

4.1.2 Image Space Shadowing Algorithms

At this time, the only image space shadowing algorithms is the one proposed by Williams.

4.1.2.1 The Shadow Buffer Algorithm

Williams [WILLIA78] suggested an image space shadowing algorithm that may be used for computing shadowed images of scenes containing curved surfaces. Curved surfaces cast curved shadows and thus the interplay between shadow volumes quickly becomes quite complex.

Williams's approach does not explicitly compute shadow volumes or regions, but rather uses an implicit technique fashioned after the depth buffer approach to computing hidden surface images [CATMUL74].

Williams suggested that a shadowed image can be computed by first computing a depth buffer view of the environment as seen from the light source. This image need only consist of depth values; pixel colors need not be computed. This depth buffer image is often referred to as a shadow buffer.

A second depth buffer image is then computed from the viewer's point-of-view. During this process, as each pixel is being tiled, the visible surface's (X, Y, Z) coordinates are transformed into the light sources coordinate space, (X', Y', Z'). The depth value, Z_S, stored in the shadow buffer at (X', Y') is compared compared with Z'. The pixel is determined to lie in shadow if $Z' > Z_S$ since this implies that another surface lies in front of it in the light source's view.

Once shadowing has been determined, the proper color is computed for the pixel in the viewer's image buffer and that pixel is updated. After all pixels in the viewer's image buffer have been tiled in this manner, a shadowed visible surface image resides in the buffer.

Williams suggested the previous algorithm as a correct implementation. He has however modified the algorithm for the sake of speed. His modification consists of calculating the two images, one from the light source's point-of-view and the other from the observer's point-of-view, independently. A post-processing pass transforms the (X, Y, Z) values of the visible pixels into the light source's coordinate space (X', Y', Z'), checks for shadowing and darkens those pixels that are found to lie in shadow.

This modified algorithm has the advantage that it need transform surface coordinates only once for each pixel. The original algorithm possibly required a transformation for each surface at each pixel. Thus this post-processing

pass makes transformation costs proportional to the number of pixels and independent of image complexity. This greatly reduces the time needed to compute images for complex scenes.

Williams's modified algorithm does have one admitted problem. Since it computes pixel intensities independently of shadowing and then simply attenuates the pixel intensity if that pixel is found to be in shadow, highlighted areas are not shadowed correctly. Highlights are due to specular reflection of light and should not occur within shadowed regions since light does not directly shine on these regions. Since the modified algorithm only attenuates pixel intensities, highlights can occur in shadowed regions.

Williams's algorithm requires time proportional to the number of objects in the scene. Creating shadowed images requires roughly twice the computational time as generating a depth buffered image, since two depth buffer images need to be computed. The fixed transformation cost may be small compared to the depth buffer image generation costs.

Since this algorithm is discrete in nature, care must be taken in the handling of certain aspects, and certain tradeoffs have been made. Most importantly, the object environment must be scaled so that the visible portion is also rendered from the light source's point-of-view. The object environment should also be scaled so that it makes maximum use of the available shadow buffer resolution. Objects not visible in the light source's view will not be able to cast shadows.

Williams suggests that if the light source is within the observer's field-of-view, the environment may be sectored as Crow has suggested [CROW77A]. Computational requirements for this particular situation increase due to the fact that multiple depth buffer images need to be created for the different sectors and each visible pixel must be transformed into each of these sector's coordinate spaces. Williams admits that the only difficulty with this lies in its increased memory demands.

Williams notes that severe perspective, which may be introduced either by the observer's view or by the light source being near to the scene, can cause quantization problems to arise when transforming coordinates from one coordinate system to the other.

In an example to illustrate the quantization problems that can arise, Williams discusses the generation of a view of a scene consisting of four spheres. Each sphere casts a shadow on itself and several of the spheres cast shadows on each other.

The first problem he addresses deals with depth quantization in a depth buffer. Ideally, a coordinate triple (X, Y, Z) representing a point on a surface would transform into (X', Y', Z') and would fall exactly on the transformed surface. However, since depth values have been quantized, the transformed depth

value Z' will lie near the surface, either a bit above or below it. Williams's solution to this is to add a depth offset to the Z' value before comparing it with the shadow buffer depth value. He incorporates this depth offset into the observer space to light source space transformation matrix so that additional computation is not required.

Curved surfaces may have silhouette curves which are not defined by the boundaries of the object. When these curved surfaces are rendered in a depth buffer, the silhouette curve may appear noisy and jagged due to quantization effects. Sampling these silhouette curves in the shadow buffer may result in a moiré in the resultant image. Williams works around this problem by first dithering the shadow buffer image and then low-pass filtering it prior to sampling.

These silhouette problems are not encountered in scenes composed of polygonal objects, and the aforementioned steps of dithering and filtering need not be applied. Williams does mention that the filtering steps have the additional property of creating a penumbra effect, i.e. soft shadow edges, which may be desirable even if not truly accurate.

4.2 Parallelizing Shadowing Algorithms

Chapter 3 advocates the use of a tesselated rectangular spatial subdivision architecture as a method for parallelizing visible surface determination. This architecture associates a virtual processor with a subspace in such a way that neighboring subspaces reside in neighboring physical processors. Each physical processor is responsible for producing a visible surface image for each of its subspaces.

I wish to extend the capability of this multiprocessor system so that it can produce images with shadowing effects. In doing this, I realize that since shadowing is a non-local effect, I will have to provide for interprocessor communications. My selection of a shadowing algorithm will determine the connectivity and bandwidth of the interprocessor communications network. Thus the principal objective is to select or develop an algorithm that requires only very localized communications among processors and that keeps this interprocessor communications to a minimum.

Localized communications will allow processors to be built with a small number of interprocessor communication channels. Localization may also allow processors to be designed somewhat independently of the size of the multiprocessor. This implies that the size of the multiprocessor may be able to change without necessitating the redesign of the individual processors.

Minimizing interprocessor communications allows the interprocessor communication channels to be built for lowest possible data rates. This has both physical and economic benefits. Moderate bandwidth channels can be longer

than higher bandwidth channels, allowing processors to be housed further apart. Moderate bandwidth communication channels are also typically much less expensive to manufacture and maintain than channels that push technological limits.

Given these selection criteria, we can set about the search for an appropriate shadowing algorithm. The previous section reviewed the currently employed shadowing algorithms. All of these algorithms were initially designed for a uniprocessor model of computation. Of these algorithms, the ray tracing algorithm is the only one that has been recommended for multiprocessor implementations; however, the proponents of these architectures have not suggested that they are appropriate for real-time image generation.

The other algorithms suffer when one attempts to map them onto a spatial subdivision multiprocessor. The shadow volume algorithm [CROW77A] can be made to work in such an environment. As objects are distributed to processors, shadow volumes can be generated and the shadow polygons distributed to the processors. If this is done, shadow polygons will be unevenly distributed to the processors, since processors further from the light source will receive more shadow polygons than processors nearer the light source. This uneven distribution of shadow polygons suggests that the shadow volume algorithm does not parallelize best onto the regular spatial subdivision architectures.

The shadow polygon algorithm [ATHERT78] works best for static scenes since it must do a reasonably large amount of work to modify the environmental data base. However since the algorithm simply creates a view of the scene from the light source's point-of-view, a spatial subdivision multiprocessor could be used to perform this task. Another spatial subdivision multiprocessor would have to be used to generate the observer's view. The only real problem with this is that the scene model needs to be reconstructed and reclipped between these two visible surface passes. It is unreasonable to expect that this work can be performed fast enough for real-time system response.

The shadow buffer algorithm [WILLIA78] can also be mapped onto a spatial subdivision multiprocessor since it also creates a view from the light source's point-of-view. Both the shadow buffer image and the viewer's image could be computed in parallel in separate spatial subdivision multiprocessors. A final shadowing pass is needed to transform visible pixels into shadow buffer coordinates and then sample the shadow buffer to determine whether the pixel lies in shadow. This pass randomly accesses the entire shadow buffer image, which is spread across many processors. This results in non-localized communications between processors. A communications network could be constructed for this task, but the effort is probably not worthwhile since the algorithm has inherent sampling difficulties that will be further highlighted in Chapter 5.

The ray tracing architectures of Ullner [ULLNER83] and Dippé [DIPPÉ84] partition the visible surface algorithm among a group of processors and the communications was localized to neighboring processors. Unfortunately, communications was not limited. Many rays are cast during the scan conversion of a scene and each ray may travel through a number of processors.

Since none of these shadowing algorithms seems to work terribly well in a multiprocessor environment, a new shadowing algorithm, the ANIMAC algorithm, was developed to make use of the tesselated spatial subdivision architecture.

4.3 The ANIMAC Algorithm

The ANIMAC consists of a two-dimensional array of processors that render an image in parallel. Each processor is assigned several image regions and is responsible for producing visible surface shadowed images for these regions. Regions are assigned to processors using the virtual tesselated rectangular spatial subdivision method described in the previous chapter. This assignment of regions to processors distributes neighboring regions to neighboring processors.

Scene descriptions are distributed to each of the ANIMAC processors by a clipping subsystem. Each ANIMAC processor has detailed information about objects associated with its image space regions but must communicate with neighboring processors if information about other objects is needed.

Each processor in the ANIMAC is identical and executes the same algorithm. This algorithm determines which surface is visible at a particular pixel and whether that surface lies in shadow. Figure 4.4 provides an abstract picture of an ANIMAC processor.

Each processor can be thought of as a collection of several subprocessors with specific tasks to perform. The *Visibility Processor* (VP) determines which of the processor's objects are visible at a particular pixel. The *Shadowing Processor* (SP) checks whether visible surfaces lie in shadow. The *Illumination Processor* (IP) computes the color for a pixel based upon information from the VP and SP. Pixel data can then be stored in a frame buffer for viewing or it may be written to disk and composited with data from the other processors at a later time.

Since visible surface determination does not require interprocessor communication, most any visible surface algorithm may be implemented in the VP. The shadowing algorithm implemented by the SP must check whether objects in other processors cast shadows upon visible objects in this processor, and thus the SP algorithm is at the heart of the ANIMAC algorithm.

Figure 4.5 illustrates that the shadowing can be determined in the ANIMAC multiprocessor by casting an illumination ray from a visible surface, S_1,

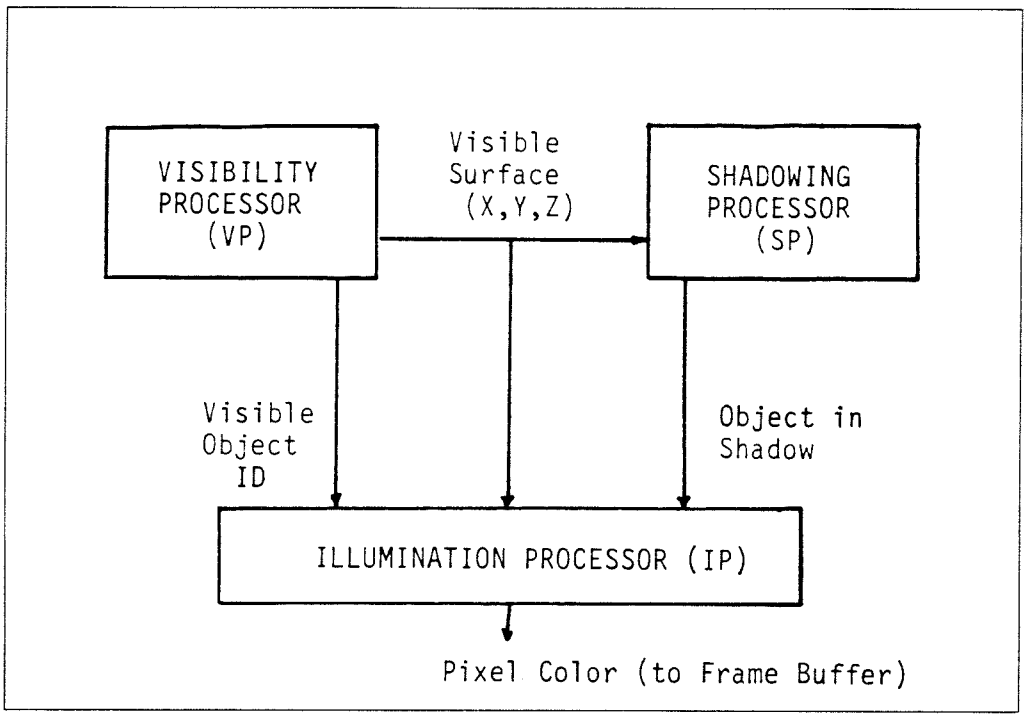


Figure 4.4: Functional diagram of an ANIMAC processor.

towards the light source, L. If any objects intersect this ray, the visible surface lies in shadow since it can receive no direct light from L.

As Figure 4.5 illustrates, the illumination ray, in traveling from the visible object, S_1, to the light source, L, passes through several processor's subspaces. Dividing the illumination ray into two segments, a local segment residing in the processor handling the pixel and a foreign segment residing in other processors, allows shadowing to be divided into two processes.

A pixel may be shadowed by a surface either local to the processor handling the pixel or local to some other processor. These two kinds of shadows require different treatment, and can be implemented with separate processors in the ANIMAC. Figure 4.6 illustrates how this affects the overall ANIMAC processor architecture. The SP has been replaced with two processors, the *Local Shadowing Processor* (LSP) and the *Foreign Shadowing Processor* (FSP). The LSP determines whether any of the objects local to the processor occlude the light source. The FSP determines whether foreign objects occlude the light source. A visible surface is now illuminated if both processors determine that the light source has not been occluded by any other object.

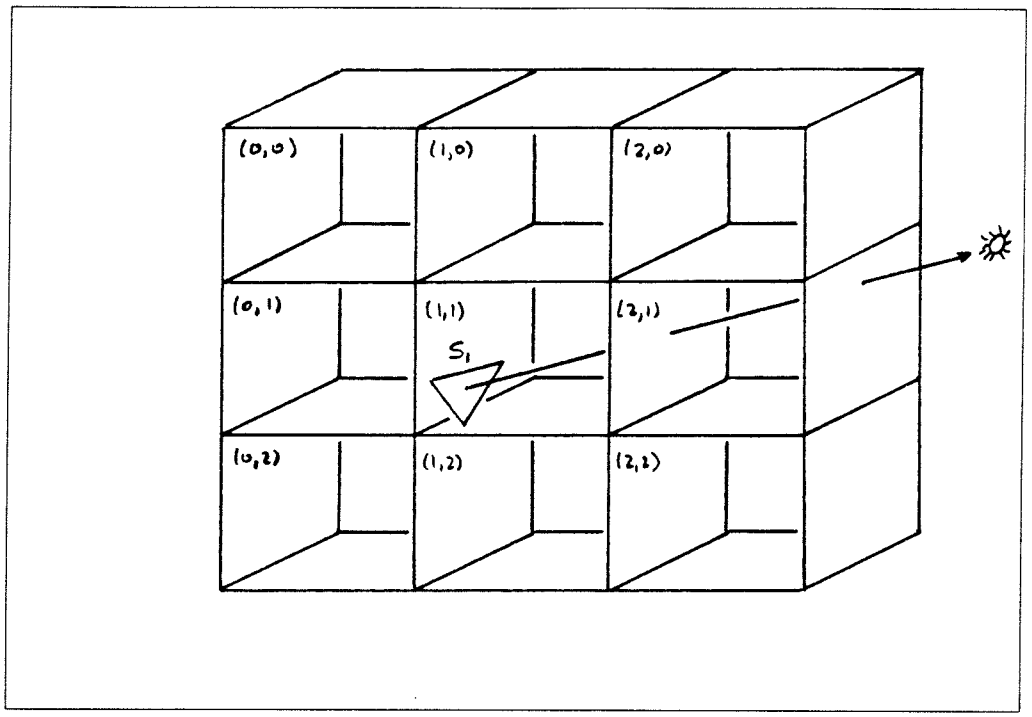


Figure 4.5: Shadowing effects can be determined by casting illumination rays and checking for intersections with other objects.

Dividing shadow determination into two processes has two potential benefits. First, different algorithms may be used to determine local and foreign light source occlusion. Second, these two processes may be executed sequentially or in parallel.

The freedom to use different shadowing algorithms for determining local and foreign light source occlusion allows us to use any of the previously reviewed shadowing algorithms in the LSP. The choice of algorithm is discussed in Chapters 5 and 6. Different algorithms may require a tighter binding between the VP and LSP than is abstractly illustrated in Figure 4.6.

Parallel determination of local and foreign light source occlusion may allow for higher performance implementations. Sequential implementations may take advantage of early termination to improve system performance. If either process decides the light source has been occluded, no more work need be done. Typically, most of the pixels in a scene are not shadowed, so both local and foreign light source occlusion need to be determined. If this is the case, the overall benefit of early termination may be minimal.

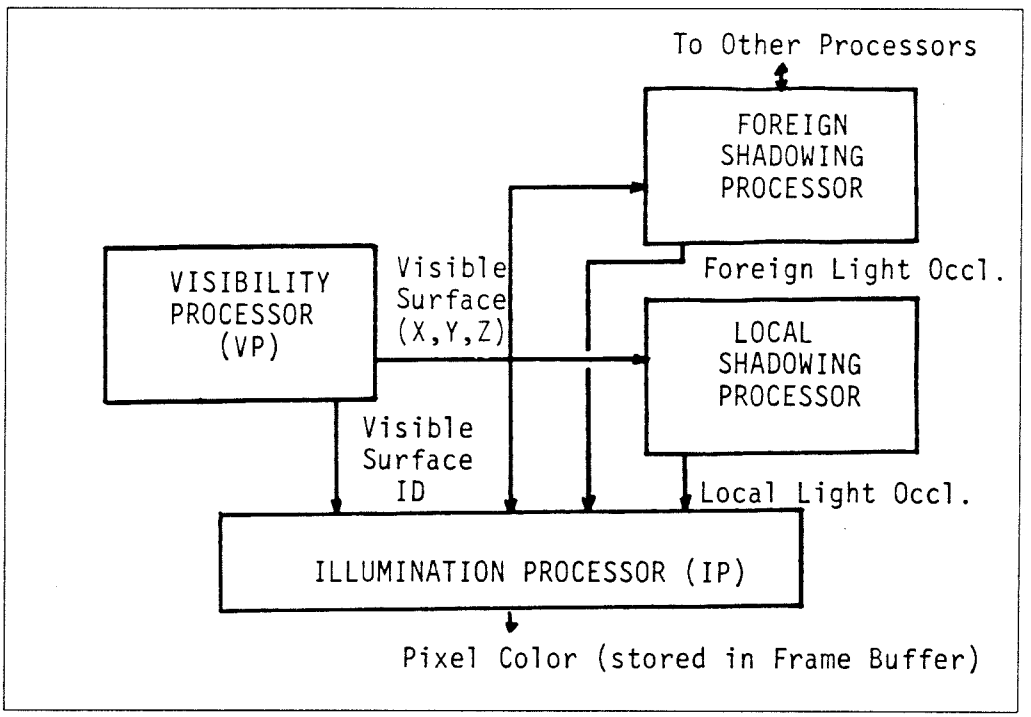


Figure 4.6: An augmented functional diagram of an ANIMAC processor. Shadowing is now determined by both the LSP and FSP.

We have developed the ANIMAC algorithm in an abstract manner. Its functional components have been described but we have not yet described how each of these functional units must be implemented. Explicit functional descriptions are not needed at this time for the VP, LSP, and IP since these processors depend only upon data which is local to the processor. The selection of which algorithms to use for these processors depends solely upon the realization of the ANIMAC architecture. The algorithm employed by the FSP requires non-local information. The ANIMAC architecture is not well described without the specification of a foreign shadowing algorithm since this will mandate a particular interprocessor communications network.

4.3.1 Shadow Maps: A Foreign Shadowing Algorithm

The shadow map algorithm provides a mechanism for determining whether objects foreign to a processor cast shadows upon visible surfaces. The shadow map algorithm requires low computational overhead and requires only nearest neighbor interprocessor communications.

In order to discuss the development of the shadow map algorithm, five restrictions will be imposed upon the discussion. First, views will be restricted to parallel projections; second, objects outside the view volume do not generate visible shadows; third, there is only one light source; fourth, the light source is distant and can be specified adequately with only a direction vector; and fifth, cast shadows will have distinct edges, simulating only the umbra of real shadows. These restrictions will serve to simplify the following discussion. A later section will discuss extensions to the algorithm that can lessen these restrictions.

Foreign shadowing can be determined by checking whether an illumination ray intersects any objects. This suggests a solution similar to ray tracing in which processors generate query messages that propagate through the multiprocessor eventually causing answer messages to be sent back to the originator. This quickly results in the generation of many messages and much work.

Foreign shadowing can also be determined using a method in which information propagates through the multiprocessor network and is collected by the local processors. After all the information has been distributed, each local processor has enough information to determine foreign shadowing. This method is a dual form to the ray tracing method.

I have named this solution the Shadow Map algorithm. It bears strong similarity to the reflectance maps of Blinn [BLINN76]. A shadow map is a description of all objects that might occlude any illumination rays cast from a point within a processor's view space region.

To check for foreign shadowing, the FSP need only compare the illumination ray with the shadow map. The exact nature of this comparison depends upon the exact representation chosen for the shadow maps. Such representations include but are not limited to: bit maps, strip trees, surface nets, and analytic representations. Each of these representations has different properties that might make it appropriate for a certain application.

Before a shadow map can be inspected, it must be constructed so that it contains the appropriate information. Figure 4.7 illustrates the construction of a shadow map for a processor. Given a light source direction, only certain processors hold objects that may shadow objects in processor (1,1). These processors are indicated in the illustration via hashing. A *Composite Shadow Map* (CSM) can be defined for processor (1,1) as the union of the projections of all potentially shadowing objects upon a *External Shadow Map* (ESM). The ESM lies in a plane that is perpendicular to the light source direction vector. The extents of the CSM are delimited by the projection of the bounding box

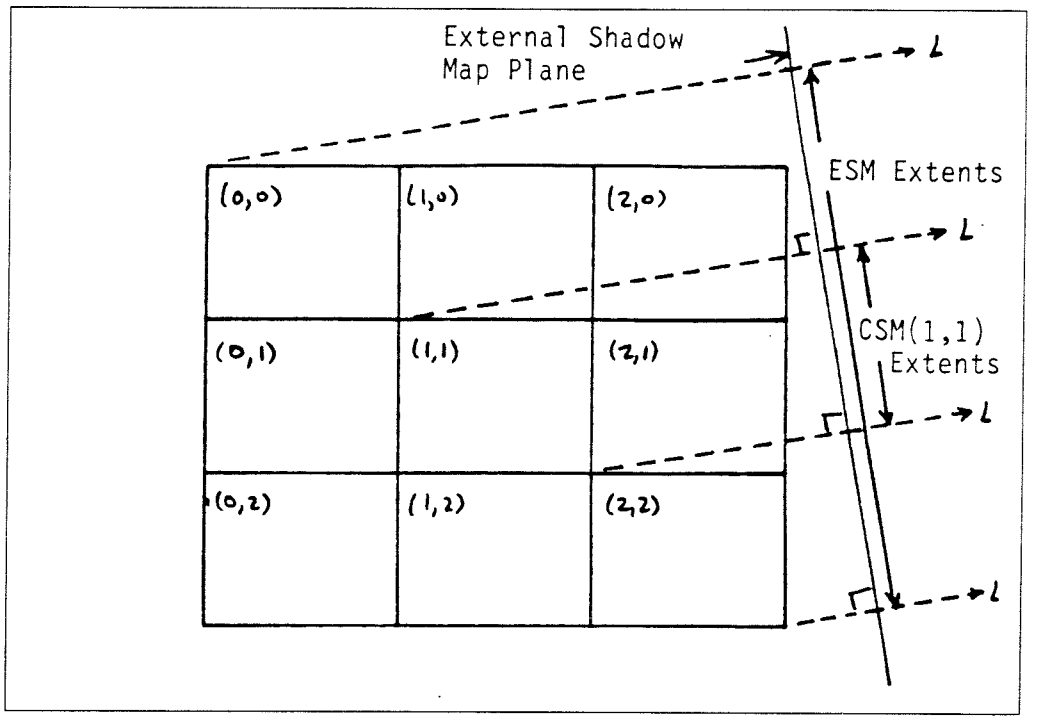


Figure 4.7: Construction of a processor's shadow map.

of processor (1,1) on the ESM. The extents of the ESM can be defined by the projection of the bounding box of the viewing volume on the ESM plane.

After the CSM has been constructed, foreign shadowing can be determined for any visible point, P, within the processor by finding the intersection of the illumination ray cast from P with the ESM plane and inquiring whether the intersection point is within the projection of any object in the CSM.

Shadow map inspection appears to be fairly economical, particularly when compared to computing ray tracing object-ray intersections. If the shadow map is represented as a bit map, each inspection consists of the projection of a point onto the shadow map plane and a two dimensional array lookup. This amounts to six multiplications and four additions for the projection and one multiplication and one addition for the array indexing. Seven multiplications and five additions is much less work than is normally done in ray tracing object-ray intersection computations.

Although shadow map inspection may be relatively inexpensive, the cost to create the shadow maps is a much more important question. Creating

shadow maps depends totally upon information in other processors. This information must be communicated in a local manner and the total amount of information transferred should be kept as small as possible.

Since the ANIMAC processors are defined in a rectangular array, at most three of a processor's neighbors may hold objects capable of casting shadows upon that processor. We refer to these processors as shadowing neighbors. A processor's CSM is composed from the union of all objects held by shadowing neighbors plus the union of all objects that can shadow shadowing neighbors.

A *Local Shadow Map* (LSM) can be constructed for a processor by forming the union of all of the processor's objects projected upon the ESM plane. This allows us to redefine a processor's CSM as the union of its shadowing neighbor's CSMs and its shadowing neighbor's LSMs. This relationship shows that the communication between processors can be easily localized. Each processor need only communicate with at most three of its nearest neighbors in order to construct its shadow map.

The amount of information to be transferred between processors depends upon the representation of the shadow maps. For example, if the shadow maps consist of an analytic descriptions of the union of objects projected upon the shadow map plane, the description may require an amount of information dependent upon the number of objects in the shadow map. If the shadow maps are represented as bit maps, they can be described by a fixed number of bits which is independent of the shadow map contents.

If shadow maps are represented as bit maps, the amount of information that must travel through any of a processor's communication channels (channel capacity) can be limited to the amount of information that processor is using to represent its shadow map.

Furthermore, the channel capacity does not need to increase as more processors are added to the multiprocessor. When the number of processors increases, one of two things may happen. Each processor may still store its shadow map with the same number of bits resulting in the overall shadow map being computed to a higher resolution since there are more processors. Channel capacity remains unchanged since the same amount of information must be transferred. Alternatively, each processor may reduce the amount of storage that it uses for shadow maps in order to compute the overall shadow map to the same spatial resolution. This reduces the amount of interprocessor communications.

The shadow map algorithm is well suited to the tesselated spatial subdivision environment employed by the ANIMAC architecture. When the shadow map algorithm is realized with bit maps, the algorithm requires extremely localized interprocessor communications and requires channel capacities which

are independent of the number of processors in the ANIMAC array. Furthermore, both LSM creation and CSM inspection are relatively inexpensive. All of these factors strongly suggest that the shadow map algorithm is an attractive algorithm to implement in the ANIMAC processor's FSP.

Figure 4.8 illustrates the shadow map algorithm on a four processor array for a simple test scene. In the test scene, Figure 4.8*a*, both objects and shadows cross the processor boundaries. Figure 4.8*b* illustrates a view of the scene from the light source's point-of-view. Figure 4.8*c* shows the scene from the observer's view point with foreign shadowing effects. Figure 4.8*d* combines local and foreign shadowing effects to yield a correctly shadowed image. Figures 4.8*e* – 4.8*j* illustrate the local and composite shadow maps which are computed by each of the four processors. Processor (1,0) has a null CSM since objects within that processor are correctly shadowed by the local shadowing algorithm. Processor (0,1) has a null LSM since no other processors depend upon this LSM. All of the shadow maps are are illustrated in ESM coordinates.

4.3.2 Shadow Map Extensions

The previous section imposed four restrictions upon the shadow map algorithm. This section will discuss how those restrictions may be eased by extensions to the shadow map algorithm.

4.3.2.1 Perspective Viewing Projections

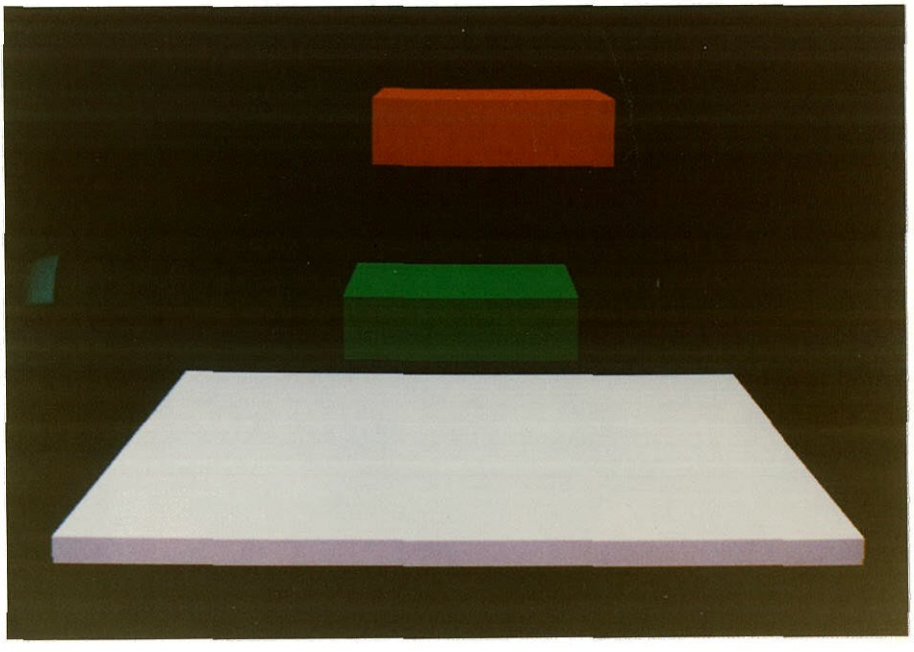
Most interesting images are created with perspective projections. The shadow map algorithm can be extended to handle perspective projections by reformulating the way composite shadow maps are created.

Parallel projections result in processor viewing volumes that are rectangular parallelepipeds. Light rays may enter a parallelepiped through at most three of its six sides. This allowed us to formulate $CSM_{i,j}$ with a simple recurrence relationship that depended upon at most three neighboring CSMs and LSMs.

Perspective projections result in each processor's viewing volume being a prismatoid. The near and far clipping planes form the parallel base planes while the four other clipping planes form the sides.

Figure 4.9 illustrates the nine viewing volumes that might result from a 3 by 3 subdivision of the viewing space. The central processor's viewing volume illustrates that when the light source is on the $-z$ axis, light rays may enter the processor's viewing volume through as many as five of its six sides.

Since light rays may enter the prismatoid through more than three of its six sides, the previous formulation for $CSM_{i,j}$ will not suffice. $CSM_{i,j}$ can be reformulated from its definition as the union of all LSMs which potentially



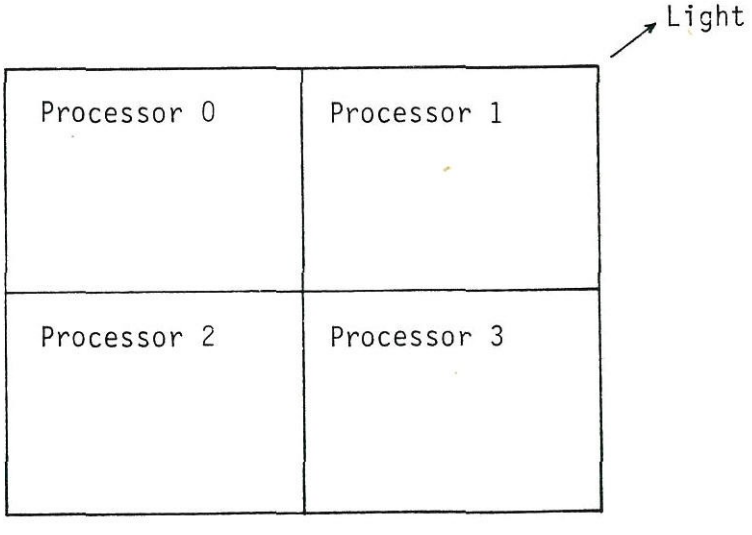


Figure 4.8a: Illustration of the shadow map algorithm for four processors. The photograph illustrates the test scene. The drawing below it illustrates the image space regions associated with the four processors.

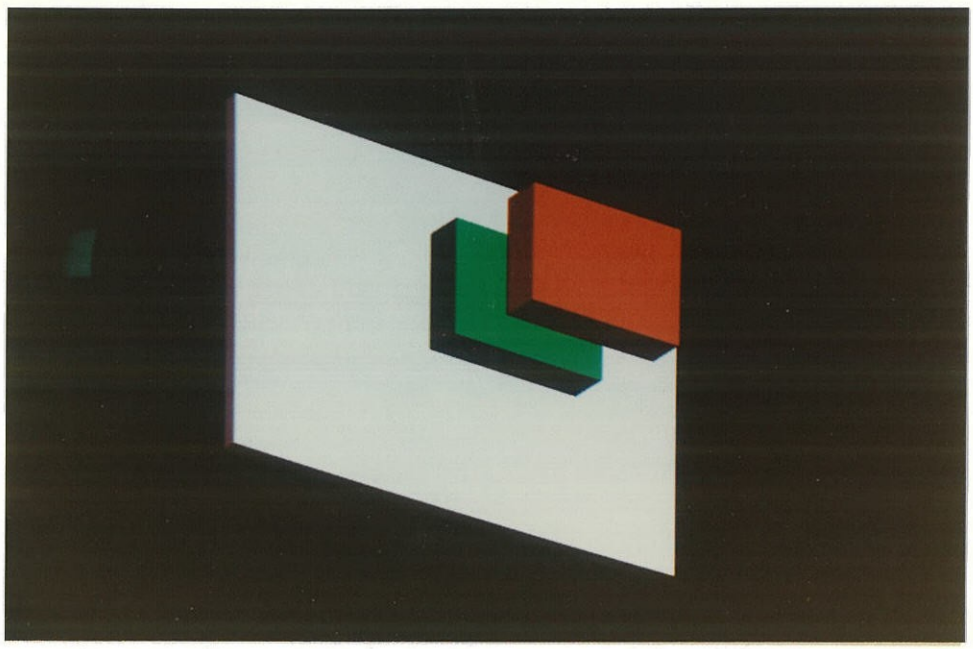


Figure 4.8b: A view of the test scene from the light source's point-of-view.

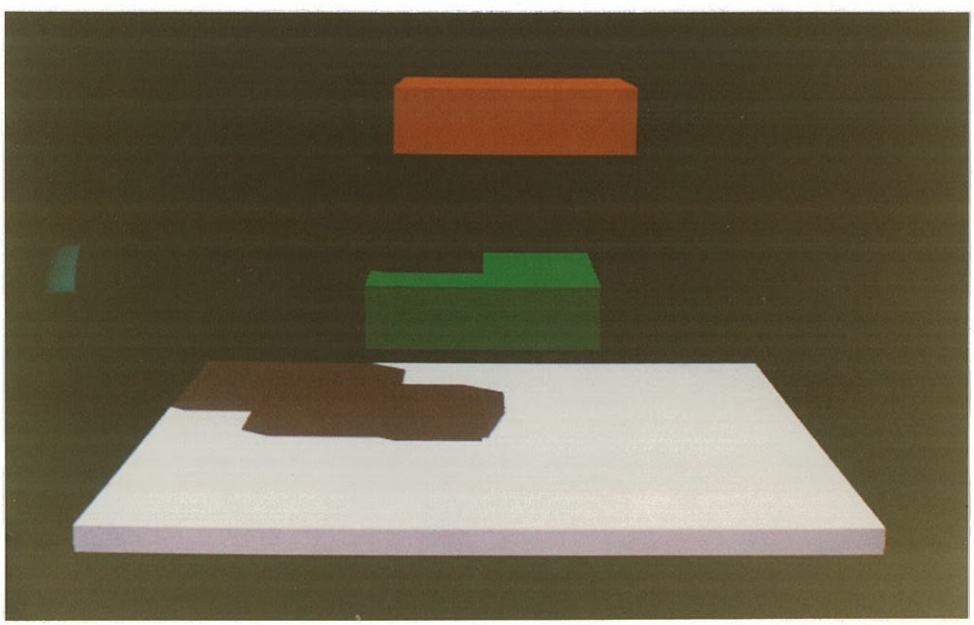


Figure 4.8c: A view of the test scene with only foreign shadowing effects.

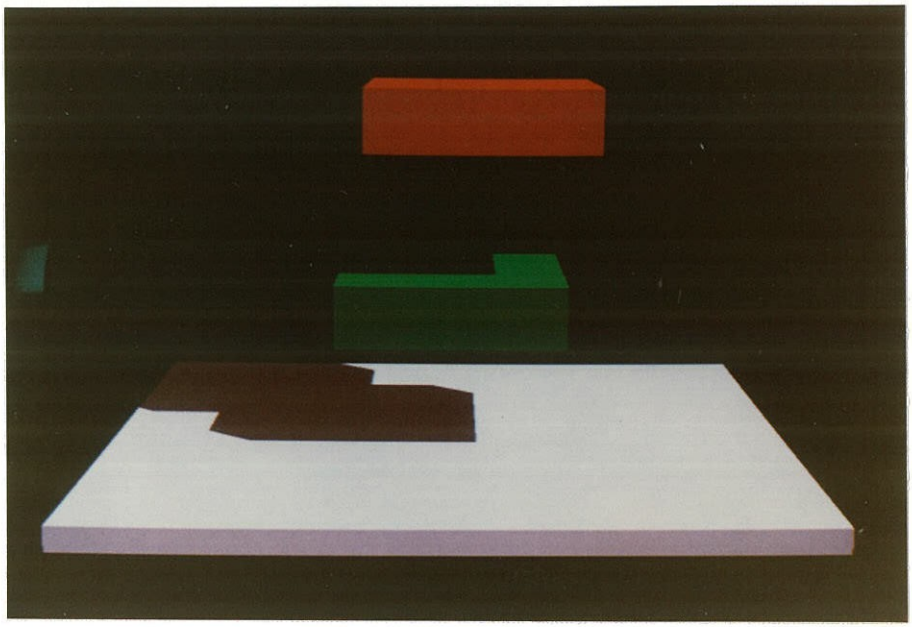


Figure 4.8d: A view of the test scene with both foreign and local shadowing effects.

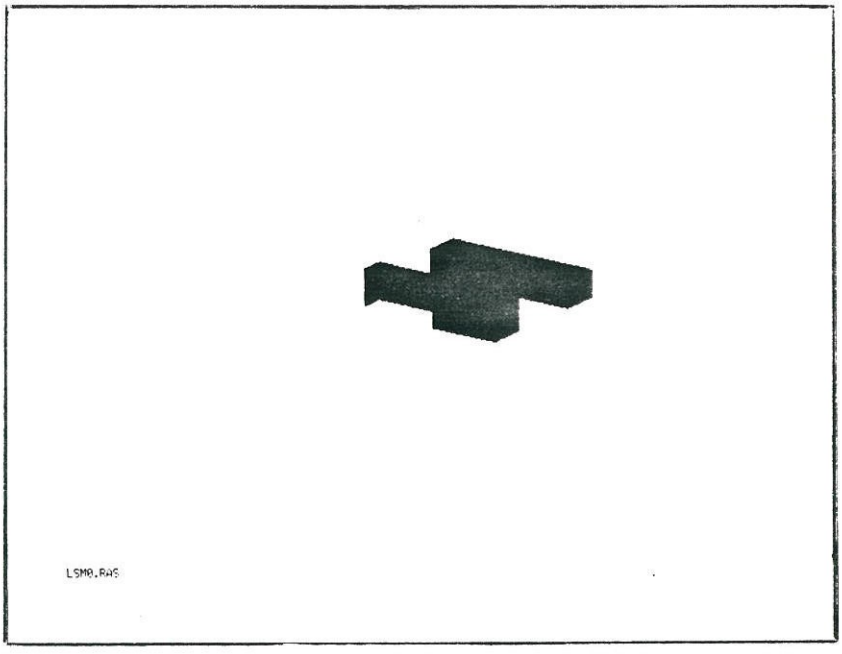


Figure 4.8e: LSM0—The local shadow map computed by processor 0.

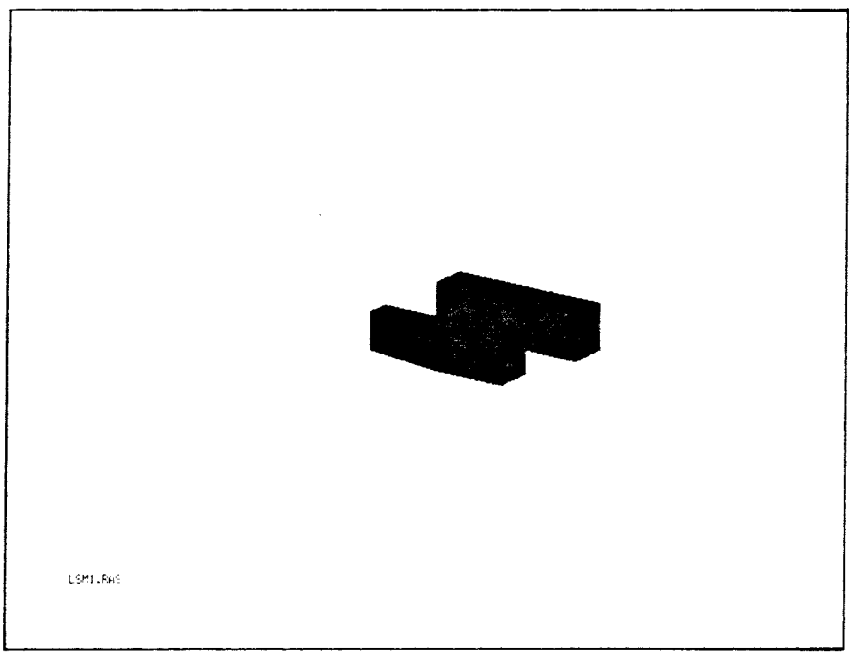


Figure 4.8f: LSM1—The local shadow map computed by processor 1.

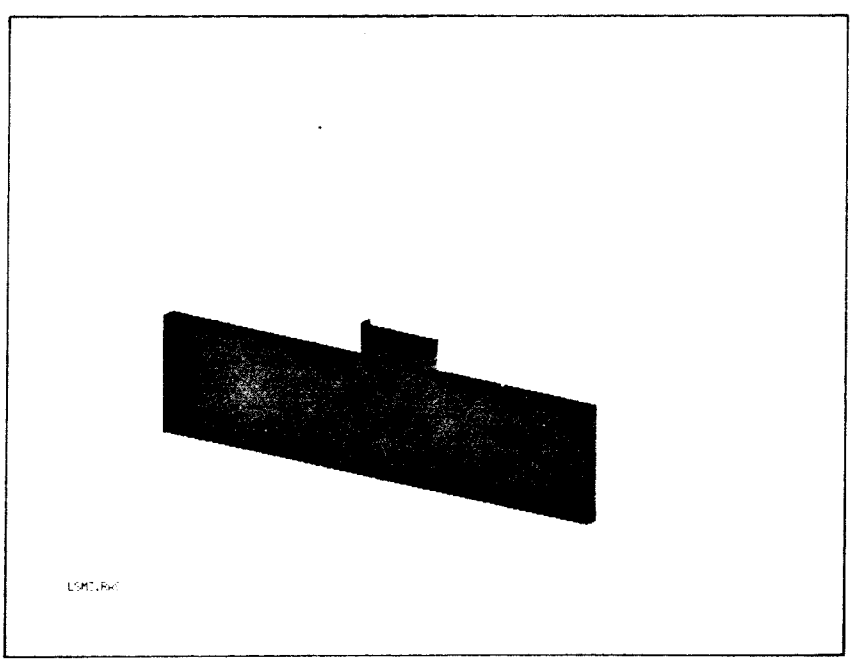


Figure 4.8g: LSM3—The local shadow map computed by processor 3.

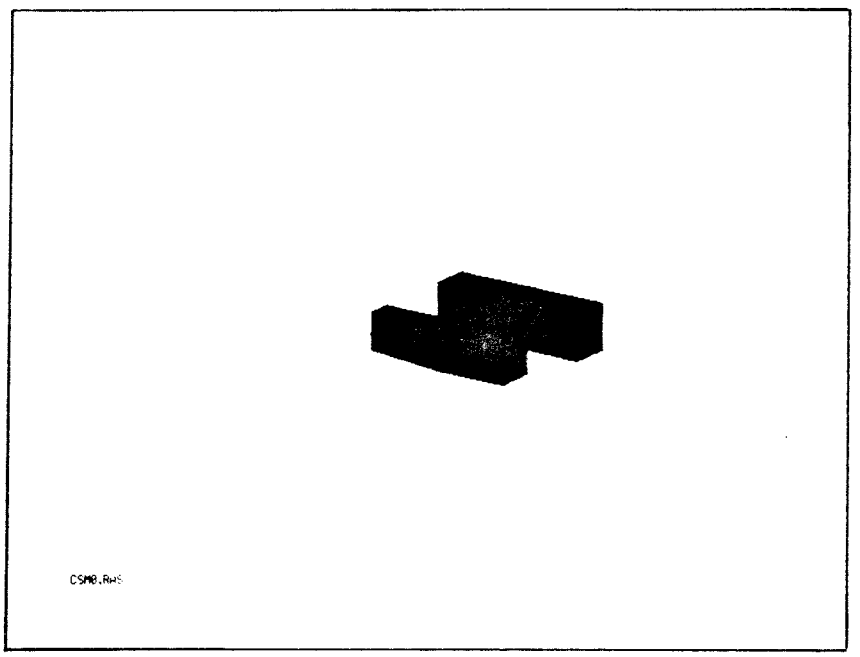


Figure 4.8h: CSM0—The composite shadow map computed by processor 0. (CSM0 = LSM1 ∪ CSM1)

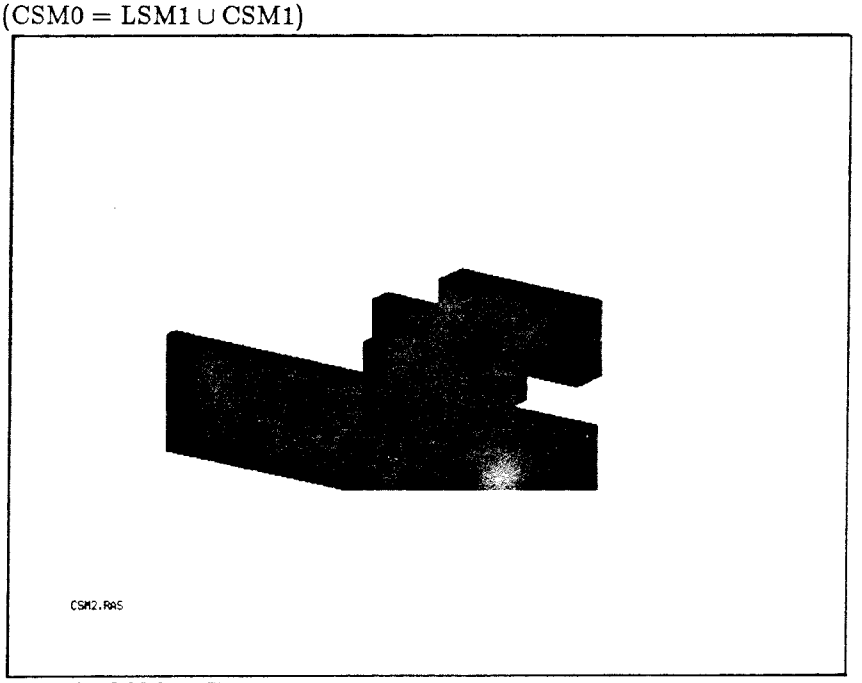


Figure 4.8i: CSM2—The composite shadow map computed by processor 2. (CSM2 = LSM0 ∪ CSM0 ∪ LSM1 ∪ CSM1 ∪ LSM3 ∪ CSM3)

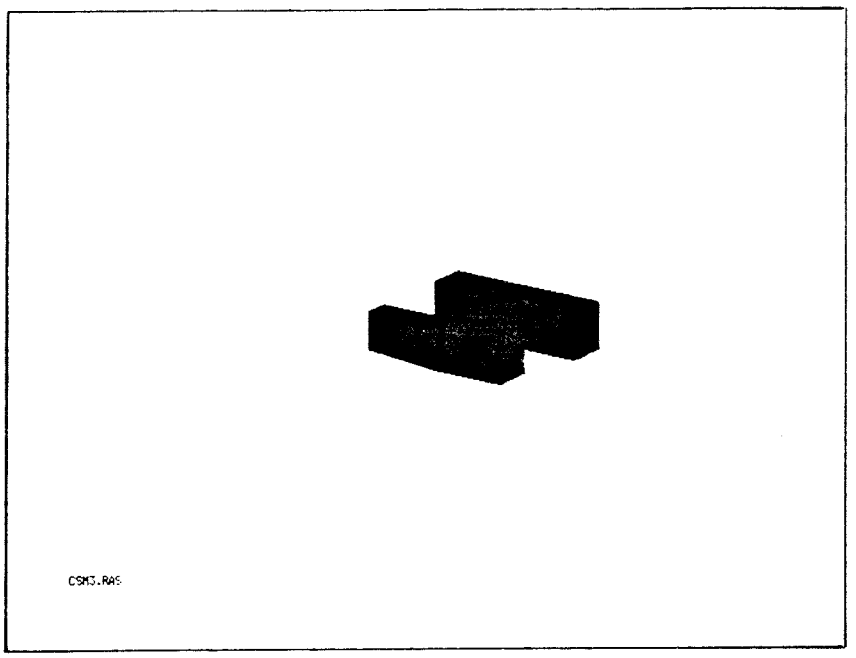


Figure 4.8j: CSM3—The composite shadow map computed by processor 3. (CSM3 = LSM1 ∪ CSM1)

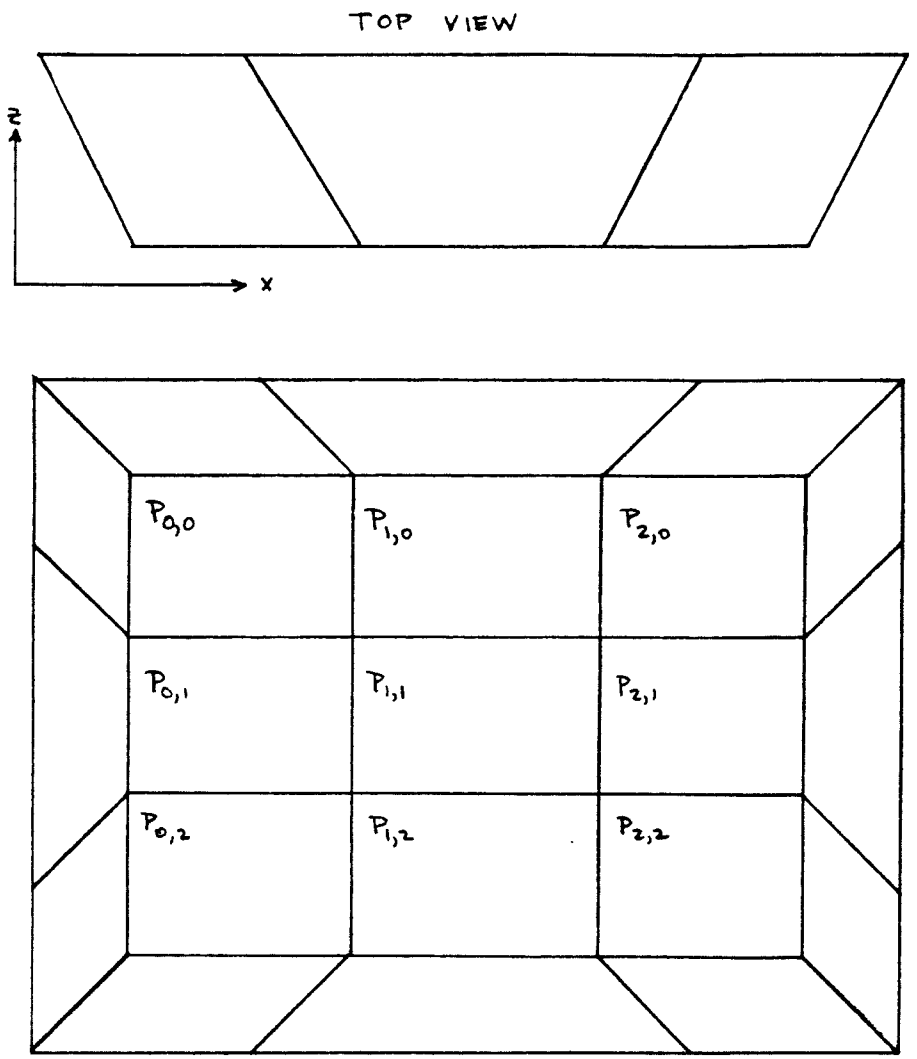


Figure 4.9: Perspective projections result in the viewing volume being a truncated pyramid. Regular subdivision of the image space results in processor viewing volumes which are prismatoids.

shadow objects within processor $p_{i,j}$. The LSMs which contribute to $\mathrm{CSM}_{i,j}$ can be determined by considering a little geometry.

The viewing space is divided by a number of planes which make up the faces of the processor's viewing volumes. Each viewing volume is bounded by six *boundary* planes. Two of these boundary planes are parallel and lie in constant z planes. The other four boundary planes intersect at the origin.

Other processors can potentially cast shadows on processor $p_{i,j}$ if they lie on the same side of a boundary plane as the light source and the light source lies on the opposite side of the boundary plane as $p_{i,j}$.

Other processors cannot cast shadows on p through the two constant z boundary planes since all processors lie on the same side of these planes. Only the left, right, upper and lower boundary planes need be considered. These four planes can be used to define four boundary sets of local shadow maps which can potentially shadow $p_{i,j}$. These sets are defined as:

$$U_{i,j} = \bigcup_{x=0}^{n} \bigcup_{y=j-1}^{0} \mathrm{LSM}_{x,y}$$

$$D_{i,j} = \bigcup_{x=0}^{n} \bigcup_{y=j+1}^{m} \mathrm{LSM}_{x,y}$$

$$L_{i,j} = \bigcup_{x=0}^{i-1} \bigcup_{y=0}^{m} \mathrm{LSM}_{x,y}$$

$$R_{i,j} = \bigcup_{x=i+1}^{n} \bigcup_{y=0}^{m} \mathrm{LSM}_{x,y}$$

where n and m are the processor array width and height.

$\mathrm{CSM}_{i,j}$, the composite shadow map for processor $p_{i,j}$ is the selective union of $U_{i,j}$, $D_{i,j}$, $R_{i,j}$ and $L_{i,j}$. These sets are unioned into $\mathrm{CSM}_{i,j}$ when their boundary plane separates the light source and processor viewing volume.

Unfortunately, this formulation of $\mathrm{CSM}_{i,j}$ requires global communications between processors. Communications can be limited by rewriting the four

boundary sets as recurrence relations.

$$U_{i,j} = U_{i,j-1} \bigcup_{x=0}^{n} \mathrm{LSM}_{x,j-1}$$

$$D_{i,j} = D_{i,j+1} \bigcup_{x=0}^{n} \mathrm{LSM}_{x,j+1}$$

$$L_{i,j} = L_{i-1,j} \bigcup_{y=0}^{m} \mathrm{LSM}_{i-1,y}$$

$$R_{i,j} = R_{i+1,j} \bigcup_{y=0}^{m} \mathrm{LSM}_{i+1,y}$$

This obvious recurrence relationship helps to localize communications but still requires that each processor communicate with many other processors. Communications can be restricted to a processor's eight nearest neighbors by introducing four new recurrence relations:

$$\mathrm{QSM1}_{i,j} = \mathrm{LSM}_{i,j} \cup \mathrm{QSM1}_{i+1,j} \cup \mathrm{QSM1}_{i+1,j-1} \cup \mathrm{QSM1}_{i,j-1}$$
$$\mathrm{QSM2}_{i,j} = \mathrm{LSM}_{i,j} \cup \mathrm{QSM2}_{i-1,j} \cup \mathrm{QSM2}_{i-1,j-1} \cup \mathrm{QSM2}_{i,j-1}$$
$$\mathrm{QSM3}_{i,j} = \mathrm{LSM}_{i,j} \cup \mathrm{QSM3}_{i-1,j} \cup \mathrm{QSM3}_{i-1,j+1} \cup \mathrm{QSM3}_{i,j+1}$$
$$\mathrm{QSM4}_{i,j} = \mathrm{LSM}_{i,j} \cup \mathrm{QSM4}_{i+1,j} \cup \mathrm{QSM4}_{i+1,j+1} \cup \mathrm{QSM4}_{i,j+1}$$

These relationships define quadrant shadow maps (QSMs). A quadrant shadow map consists of the union of all local shadow maps that lie in a quadrant. For example, $\mathrm{QSM1}_{i,j}$ consists of the unions of all $\mathrm{LSM}_{x,y}$ for $x \geq i$ and $y \leq j$.

The four boundary sets can be rewritten using the quadrant shadow map recurrence functions as:

$$U = \mathrm{QSM2}_{i-1,j-1} \cup \mathrm{QSM2}_{i,j-1} \cup \mathrm{QSM1}_{i,j-1} \cup \mathrm{QSM1}_{i+1,j-1}$$
$$D = \mathrm{QSM3}_{i-1,j+1} \cup \mathrm{QSM3}_{i,j+1} \cup \mathrm{QSM4}_{i,j+1} \cup \mathrm{QSM4}_{i+1,j+1}$$
$$L = \mathrm{QSM2}_{i-1,j-1} \cup \mathrm{QSM2}_{i-1,j} \cup \mathrm{QSM3}_{i-1,j} \cup \mathrm{QSM3}_{i-1,j+1}$$
$$R = \mathrm{QSM1}_{i+1,j-1} \cup \mathrm{QSM1}_{i+1,j} \cup \mathrm{QSM4}_{i+1,j} \cup \mathrm{QSM4}_{i+1,j+1}$$

This formulation for the boundary sets, allows the CSM to be computed as the union of at most twelve quadrant shadow maps. These twelve quadrant shadow maps are precisely the same twelve shadow maps which the processor requires to compute its four QSMs.

Interprocessor communication need only occur during the computation of the quadrant shadow maps. The computation of each quadrant shadow map is easily pipelined and the four QSMs can be computed in parallel.

CSMs can be implemented in hardware as bit maps with 12 bits per pixel. Each bit corresponds to one of the twelve QSMs. CSMs can be inspected by accessing the 12-bit pixel value and masking out the bits which correspond to the needed QSMs. The mask depends upon the light source direction and the processor viewing volume and can be computed once per frame.

4.3.2.2 Shadows Cast by Non-Visible Objects

Objects which are not visible can cast shadows upon visible objects. The Shadow Map algorithm can be extended to handle these shadows. This extension requires two changes to the way scenes are computed.

Only certain non-visible objects are capable of generating visible shadows. These objects lie in a restricted portion of the modeling space that can be determined from the light source direction vector and the viewing volume coordinates. Potential shadow casting objects may be selected by clipping the model appropriately. An *external shadow map* (ESM) can be formed from the union of the projections of these objects on the ESM plane.

Once the ESM has been created, it needs to be unioned into the CSMs associated with all processors on the boundaries of the array that face towards the light source. This unioning requires that processors have communications channels to the ESM processor.

4.3.2.3 Multiple Light Sources

The Shadow Map algorithm is easily extended to handle multiple light sources. Each light source requires the creation of a local shadow map and a composite shadow map in each processor. A certain amount of interprocessor channel capacity is also required for the unioning of shadow maps. Multiple light sources require each processor to maintain multiple shadow maps and need increased interprocessor channel capacity. The shadow maps are entirely independent and may be computed in parallel or may be sequentialized to reduce processing and communications requirements.

4.3.2.3 Localized Light Sources

Distant light sources are modeled with an intensity and a direction vector. Since localized light sources result in non-parallel illumination rays, they cannot be represented with a direction vector but instead are represented with a modeling space coordinate. Illumination models usually model the intensity of a localized light source with a function that decreases with the square of

the distance from the light source. These intensity functions produce images that often appear both much more realistic and more visually complex than images produced with the constant intensity functions utilized with distant light sources.

Shadows cast by localized light sources can be computed using shadow maps. As with other shadowing algorithms, special care must be taken if the light source is within the viewing volume. We first consider the case when the light source is outside of the viewing volume.

With distant light sources, the view of the scene from the light source was an orthographic projection. Localizing the light sources causes this view to be a perspective projection. The Shadow Map algorithm introduced the External Shadow Map Plane as a virtual screen. Shadow maps could be thought of as rectangles in the ESM plane. With localized light sources, the ESM plane can still be treated as a virtual screen and shadow maps can still be thought of as rectangles in the ESM plane. The ESM can no longer be described as existing in a plane perpendicular to the light source direction vector but instead must be constructed so that it is a rectangle through which the light source views the world.

Shadow maps are created by projecting objects onto the ESM plane. With localized light sources, a plane can be constructed that contains the light source point and bisects the modeling space such that objects to one side of the plane are capable of casting shadows upon objects within the viewing volume. Objects on the other side of this dividing plane cannot shadow visible objects and need to be discarded. The remaining objects can be projected upon the ESM during the creation of the local and external shadow maps.

Shadow maps must be inspected differently than with distant light sources. Given a visible point P, we must compute the intersection of the ray PL with the ESM plane. Fortunately, inspection can still be treated as the transformation of a point, but in this case, the transformation is a full perspective transformation which requires more arithmetic operations than before.

Since all shadow maps are defined in the same coordinate system, the coordinate system of the ESM, localizing the light source does not change how they are unioned. Unioning is still a function of the shadow map representation.

The light source existing within the viewing volume, i.e. within a processor's region, complicates matters. In this case we must assure that the local shadowing algorithm can handle this condition. Both Crow's Shadow Volume and Williams's Shadow Map algorithms can be made to handle this condition by sectoring space and running independent algorithms on each sector.

Foreign shadowing effects should not be computed for the region containing the light source. Normally, processors map their shadow maps onto the External Shadow Map which consists of a view of the entire viewing volume

from the point-of-view of the light source. If the light source is within the scene, one unique ESM does not exist. The multiprocessor array can be partitioned into four subarrays. Each subarray requires a separate notion of what the ESM is but since the light source is outside each of these subarrays, foreign shadowing can be treated exactly as it is for ordinary local light sources. Figure 4.10 illustrates how a nine processor array might be partitioned into four subarrays.

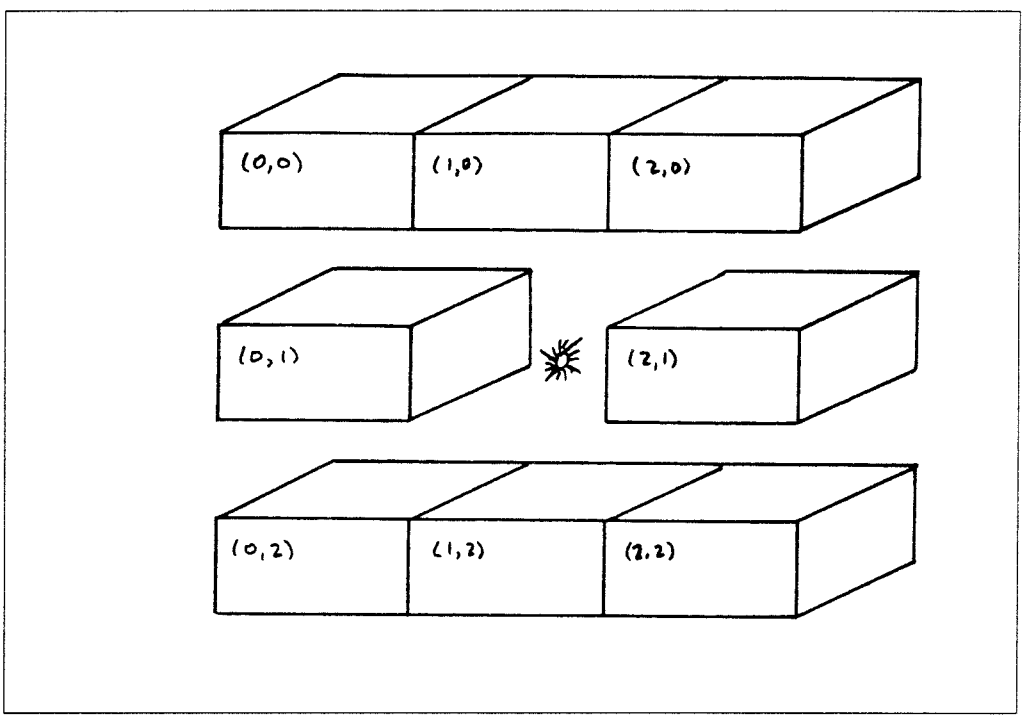


Figure 4.10: The processor array can be partitioned into four subarrays when the light source lies within a processor, processor (1,1) in this case.

Localized light sources require additional work be done when creating the local shadow maps. Most of this additional work results from the increase computation required by perspective projections over othrographic projections. The shadow map algorithm extends naturally to handle localized light sources.

4.3.2.5 Soft Shadows

Soft shadows (penumbra) cannot be adequately represented with this algorithm. Williams has suggested that low-pass filtering of his shadow buffer

images results in a pleasing soft shadow effect. This effect is, of course, totally without physical basis.

Similar low-pass filtering of shadow maps may be performed with this algorithm. Again, any pleasing effects would be totally without a physical basis. The author has no interest in this area and has not attempted to achieve such effects. Obviously, care must be taken if one is to attempt this, since it is important to have a consistent view of the shadowing environment from each processor.

4.4 Conclusions

The ANIMAC algorithm and architecture have been further developed in this chapter. ANIMAC still utilizes the virtual tesselated spatial subdivision architecture advocated in Chapter 3 but now includes a nearest neighbor communications network. The visible surface algorithm, employed by the ANIMAC engines, has been extended to include shadowing effects.

The ANIMAC architecture has been described functionally. Many implementations meet this functional description. The choice of visible surface and local shadowing algorithms has been left to the implementor. The next two chapters discuss implementations. Chapter 5 discusses a software implementation which has served as an ANIMAC simulator. Chapter 6 discusses how emerging VLSI architectures can be used to create a hardware ANIMAC implementation.

5

Simulation of the ANIMAC Shadowing Algorithm

This chapter discusses a software implementation of the ANIMAC shadowing algorithm. The ANIMAC shadowing algorithm was implemented within the framework of the author's *render* program. In order to discuss the implementation details, it becomes necessary to understand how *render* creates images.

Render uses any of a number of different visible surface algorithms to create an image of a model. *Render* requires a model to be a hierarchy of objects. Only the leaf nodes in the model hierarchy can represent geometric objects. These nodes are referred to as *primitive objects*. All other nodes in the hierarchy are called *modeling objects*. Modeling objects are used to compose primitive objects into more complex objects.

Render creates an image by traversing the model hierarchy. Primitive objects are transformed into the viewing space, clipped against the viewing volume, and finally projected onto a view plane.

A series of transformations transforms a point p in a primitive object's coordinate system successively into modeling coordinates, viewing coordinates, clipping coordinates, and image coordinates. By composing transformations, *render* uses one transform, T_{ctm}, to transform points from a primitive object's coordinate space into viewing space. T_{ctm} is composed as:

$$T_{ctm} = MV$$

M is a composite modeling transformation which transforms p from an object's coordinate system into the global model coordinate system. V is a

viewing transform which transforms the modeling coordinate system into a left-handed viewing coordinate system. *Render*'s viewing coordinate system is arranged such that the $+Z$ axis goes into the screen. The X axis maps onto the screens horizontal axis with positive being to the right. The Y axis maps onto the screen's vertical axis, positive being up.

Points are transformed from viewing space to clipping space with another transform T_p. T_p is composed as:

$$T_p = PT_tT_a$$

where P is a projection transformation that implements either a perspective or parallel projection. *Render* can process an image by dividing it up into tiles. A particular tile can be made to fill the clipping volume by applying scale and translation transforms represented here as the tiling transform T_t. Different output devices have different aspect ratios. Aspect ratios are compensated for by the aspect transform T_a. Perspective division is performed after clipping the object. A final viewport transform, T_v, translates clipping coordinates into image pixel coordinates.

Details of the implementation of T_p and T_v are of particular interest since discussion of the shadow buffer and shadow map algorithms use these transformations to create transformations that map image space coordinates into shadow buffer or shadow map coordinates.

The form of P depends upon whether the projection is perspective or parallel. P is simply an identity matrix for parallel projections. Perspective projections are implemented in a form similar to [BLINN82]. Given a viewer at the origin, a field-of-view 2θ, and near and far clipping planes, $Z = Z_{near}$ and $Z = Z_{far}$, the perspective projection matrix can be written as:

$$P = \begin{bmatrix} \cos\theta & 0 & 0 & 0 \\ 0 & \cos\theta & 0 & 0 \\ 0 & 0 & a & \sin\theta \\ 0 & 0 & b & 0 \end{bmatrix}$$

where:

$$a = \frac{Z_{far}\sin\theta}{Z_{far} - Z_{near}}$$

$$b = \frac{-Z_{near}Z_{far}\sin\theta}{Z_{far} - Z_{near}}$$

The tiling transform, T_t, depends only upon the tiling ratios, t_{sx} and t_{sy}, and the tile center coordinates (x_t, y_t). The tiling ratio is determined by the

number of tiles in the x or the y direction. The tile centers are defined by the clipping space coordinates at the middle of a tile. T_t is formed as:

$$T_t = \begin{bmatrix} t_{sx} & 0 & 0 & 0 \\ 0 & t_{sy} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_{sx}x_t & t_{sy}y_t & 0 & 1 \end{bmatrix}$$

The aspect ratio correction transform, T_a, scales y values in the range $[-Aspect, Aspect]$ to map onto $[-1, 1]$.

$$T_a = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1/Aspect & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The composite projection transform, T_p, can be represented as:

$$T_p = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ t_x \sin\theta & t_y \sin\theta & a & \sin\theta \\ 0 & 0 & b & 0 \end{bmatrix}$$

where:

$$\begin{aligned} s_x &= t_{sx} \cos\theta \\ s_y &= t_{sy} \cos\theta / Aspect \\ t_x &= t_{sx} x_t \\ t_y &= t_{sy} y_t / Aspect \end{aligned}$$

The viewport transformation maps clipping coordinates into pixel coordinates. Given physical device dimensions, x_d and y_d, *render* constructs T_v as:

$$T_v = \begin{bmatrix} v_{sx} & 0 & 0 & 0 \\ 0 & v_{sy} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ v_{dx} & v_{dy} & 0 & 0 \end{bmatrix}$$

where:

$$\begin{aligned} v_{sx} &= \frac{x_d}{2} \\ v_{sy} &= \frac{y_d}{2Aspect} \\ v_{dx} &= \frac{x_d - 1}{2} \\ v_{dy} &= \frac{y_d - 1}{2} \end{aligned}$$

Render implements T_p and T_v with explicit calculations instead of using a vector-matrix multiplication. The perspective transformation T_p is implemented as:

$$
\begin{aligned}
z' &:= az + bw; \\
w' &:= z \sin\theta; \\
x' &:= s_x x + t_x w'; \\
y' &:= s_y y + t_y w';
\end{aligned}
$$

Perspective division and the viewport transform are implemented as:

$$
\begin{aligned}
x'' &:= \frac{x' v_{sx}}{w'} + v_{dx}; \\
y'' &:= \frac{y' v_{sy}}{w'} + v_{dy}; \\
z'' &:= \frac{z'}{w'};
\end{aligned}
$$

A point p'' in image coordinates can be transformed back into the viewing coordinate point p by a transformation T_{pv}^{-1}. *Render* implements this transform as:

$$
\begin{aligned}
z &:= \frac{b}{z'' \sin\theta - a}; \\
x &:= z \sin\theta \frac{x'' - v_{dx} - v_{sx} t_x}{v_{sx} s_x}; \\
y &:= z \sin\theta \frac{y'' - v_{dy} - v_{sy} t_y}{v_{sy} s_y}; \\
w &:= 1;
\end{aligned}
$$

T_{pv}^{-1} turns out to be a very useful transformation. Both the shadow buffer and shadow map algorithms use this transformation.

Besides constructing the needed transformations, *render* computes additional information about the model prior to rendering it. Two bounding boxes are computed for each object in the modeling hierarchy. Since the scene model is considered as an object, these bounding boxes are also computed for it. One bounding box, *bb*, represents the model's extents in the modeling space. The other bounding box, bb_s, represents the bounding box of all shadow casting objects within the model in the modeling space. *Render* allows objects to be tagged as *shadow casting* or not in order to increase the effective spatial resolution of shadow buffers and shadow maps. Only objects tagged as shadow casting can shadow other objects.

Render simulates an ANIMAC processor by associating with each processor (1) a subspace within the viewing volume and (2) a portion of the screen. The program provides control over (1) the X and Y dimensions of the processor array, (2) the dimensions of the screen region associated with each processor, (3) a pixel subsampling ratio used for antialiasing, and (4) the dimensions of the External Shadow Map (ESM). The dimensions of both a processor's screen region and the ESM are specified by a width and height measured in pixels.

The pseudo-code in Figure 5.1 outlines the method used to sequentialize the ANIMAC algorithm. First, the local shadow maps are computed. Composite shadow maps are computed next. After the shadow maps have been computed, each processor computes its visible surface image. A local shadow algorithm is then executed. Finally, pixels are checked to see whether they lie in local or foreign shadows. If they do, the pixel's color is attenuated to approximate shadowing effects.

When antialiasing measures are being taken, pixel colors are computed by tiling the image on a subpixel raster. Pixel colors are assigned by filtering the subpixel raster. *Render* creates a subpixel raster by dividing the screen region associated with each processor into tiles. Visible surfaces and local shadowing effects are computed independently for each of a processor's tiles. The final image is reconstructed by filtering and assembling each processor's tiles.

```
foreach processor do
      Compute Local Shadow Map
od

foreach processor do
      Compute Composite Shadow Map
od

foreach processor do
      foreach tile do
            Compute Visible Surface Image
            Compute Local Shadowing Effects
            Merge Local and Foreign Shadowing Effects
      od
od
```

Figure 5.1: Sequentializing the ANIMAC algorithm.

Two local shadowing algorithms were implemented: a simplified version of Williams's shadow buffer algorithm [WILLIA78] and a depth buffer implementation of Crow's shadow volume algorithm [CROW77A]. The following sections will discuss the implementation of both of these algorithms in detail.

Foreign shadowing was implemented with the shadow map algorithm developed in the preceding chapter. In this implementation, shadow maps were implemented as high resolution bit maps. The section on foreign shadowing will discuss design decisions relating to bit map shadow map implementations.

The implementation was tested by computing a number of test images. Since the goal of the ANIMAC algorithm and architecture is to provide for an animation environment, motion tests were made to study how well cast shadows animate.

5.1 Implementing the Shadow Buffer Algorithm

The shadow buffer algorithm proposed by Williams [WILLIA78] was the first local shadowing algorithm implemented within the *render* framework. Since both the shadow buffer algorithm and the shadow map algorithm are image space shadowing algorithms, it was hoped that an ANIMAC implementation might be able to use the shadow buffer algorithm to compute the local shadow maps in the process of computing local shadowing effects.

For this implementation, the shadow buffer was implemented as a 32-bit per pixel buffer. Each buffer entry consists of a single-precision floating point number which represents the depth of the visible surface at that pixel. The program provides control over the shadow buffer dimensions. Shadow buffer dimensions default to 512 by 512 pixels which requires one megabyte of storage.

The shadow buffer algorithm can be divided into two phases. The first phase creates a depth buffer image of the scene as viewed from the light source. The second phase determines whether points on visible surfaces lie in shadow. Each point, in image space coordinates, is transformed into a shadow buffer address and depth. The depth is then compared with the depth value stored in the shadow buffer to determine whether the object lies in shadow.

To create the shadow buffer image, *render* allocates storage for a shadow buffer, and then creates a viewing transformation which positions the camera at the light source, looking down the light source direction vector, L, at the scene model. This viewing transformation should scale the scene model in X and Y to make use of as much of the shadow buffer's spatial resolution as is possible. Skewing can also be incorporated in the viewing transformation to make more use of the available spatial resolution. Depth resolution is as important as spatial resolution; the viewing transformation should crop the image to make full use of shadow buffer's depth resolution.

Render uses the LView() procedure to create a viewing transformation. LView() takes as arguments a bounding box which encloses the scene model and a light source direction vector, L. LView() implements a specialized *lookat* transformation in which the viewer gazes down the light direction vector at the model. The model is scaled and translated within this view to fill the viewer's clipping volume.

The LView() procedure creates a view of the scene model from an arbitrary direction. This is accomplished by a series of transformations. L can be used to define a plane, P_l, which represents the projection plane in modeling space. We construct P_l to be perpendicular to L and to contain the bounding box vertex, V_l, nearest to the light source. P_l must be transformed so that it becomes the $Z = 0$ plane in the viewing coordinate system. This is accomplished by two simple transformations. The first transformation, T_1, translates the modeling space so that P_l now contains the origin. The second transformation, R, rotates the modeling space so that P_l is the $Z = 0$ plane. Two additional transformations complete the modeling to viewing space transformation by translating (T_2) the model in X and Y so that its projection on the $Z = 0$ plane is centered about the origin and scaling (S) it so that it fills the clipping volume. The resultant viewing transformation, V_{sbuf}, is simply the composition of the transformations:

$$V_{sbuf} = T_1 R T_2 S$$

Determining the rotation transformation is the only difficult part of this task. The rotation can be treated as a generalized rotation that rotates any three orthogonal basis vectors onto the principal axes. A rotation transformation that maps the (v_x, v_y, v_z) vectors onto the X, Y, and Z axes is constructed as:

$$\begin{bmatrix} v_{x_1} & v_{y_1} & v_{z_1} & 0 \\ v_{x_2} & v_{y_2} & v_{z_2} & 0 \\ v_{x_3} & v_{y_3} & v_{z_3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This leaves us with the task of selecting the three orthogonal basis vectors. Since we want to create a view looking down the $-L$ axis, we want the rotation matrix to rotate $-L$ onto the Z axis. This leaves us with only having to define one more basis vector since the third can be constructed as the cross product of the first two. We chose to select v_y which will rotate onto the Y axis; v_y may be any vector perpendicular to $-L$ but some choices may be better than others. In selecting v_y, we desire to find a projection of the scene model upon the $Z = 0$ plane that ultimately makes best use of the spatial resolution of the shadow buffer.

This implementation selects v_y in a non-optimal manner. It projects V_l onto P_l to find V_{pl}. The other bounding box vertices are also projected onto P_l and a point V_{pd} is selected as being the vertex which lies most distant from V_{pl}. The vector $(V_{pd} - V_{pl})$ is then used as v_y resulting in a rotation matrix:

$$v = \frac{(V_{pd} - V_{pl})}{\|V_{pd} - V_{pl}\|}$$

$$u = v \times -L$$

$$R = \begin{bmatrix} u_1 & v_1 & -L_1 & 0 \\ u_2 & v_2 & -L_2 & 0 \\ u_3 & v_3 & -L_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The final rotation and scaling transformations are determined from the scenes new bounding box dimensions. The new bounding box, bb', is calculated by transforming the scene's bounding box, bb, by $T_1 R$. Bounding boxes are transformed by transforming the eight vertices, which represent the bounding box's corners, and then constructing a new bounding box that encloses these transformed vertices. Given the transformed bounding box bb', the translate and scale transforms are:

$$t_x = \frac{bb'_{min_x} + bb'_{max_x}}{2}$$

$$t_y = \frac{bb'_{min_y} + bb'_{max_y}}{2}$$

$$s_x = \frac{2}{bb'_{max_x} - bb'_{min_x}}$$

$$s_y = \frac{2}{bb'_{max_y} - bb'_{min_y}}$$

$$s_z = \frac{1}{bb'_{max_z}}$$

$$T_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & 0 & 1 \end{bmatrix}$$

$$S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

After the viewing transformation has been constructed, the scene is rendered into the shadow buffer. *Render* renders a scene by traversing the model hierarchy. During this traversal, object coordinates are first transformed into viewing space coordinates and then clipped against the viewing volume. Visible objects are then tiled in the shadow buffer. *Render* makes use of extent testing to quickly cull non-visible objects. Backface culling is also performed on closed polyhedral objects that do not intersect the near clipping plane.

After the scene has been rendered into the shadow buffer, points on visible surfaces can be checked to determine whether they lie in shadow. The *render* implementation of the ANIMAC algorithm uses a final post-processing pass to determine whether points on visible surfaces lie in shadow, attenuating the color of those pixels which are found to be in shadow. Shadows may be cast by either local or foreign objects. This discussion will focus on determining local shadowing when using the shadow buffer algorithm.

A transformation, S_{buf}, was constructed to transform coordinates from image space, ie. (x_i, y_i, z_i), to shadow buffer space, (x', y', z'). This transformation essentially transforms the pixel coordinates back into modeling space coordinates; then transforms the modeling space coordinates by the shadow buffer viewing coordinates. The complete transformation was constructed as:

$$S_{buf} = T_{pv}^{-1} V^{-1} V_{sbuf} T_{sv}$$

T_{pv}^{-1} transforms image space coordinates back into eye viewing space. V^{-1} transforms eye viewing space coordinates back into modeling space. V_{sbuf} transforms modeling space coordinates into virtual shadow buffer coordinates and the viewport transformation, T_{sv}, maps virtual shadow buffer coordinates into physical shadow buffer coordinates.

Sampling problems arise when transforming the visible surface coordinates into shadow buffer coordinates. Both the visible surface and shadow buffer images were constructed by point sampling the scene. During the generation of each of these images, great care was taken to consistently sample objects on pixel centers so that depth comparisons between objects were meaningful. Unfortunately, sample points in the image buffer do not necessarily transform into sample points in the shadow buffer. Figure 5.2 illustrates that comparing the transformed depth with the depth stored in the shadow buffer introduces an error ΔZ. This error depends upon the plane equation of the object stored in the shadow buffer.

Given a particular planar surface, S, in the shadow buffer coordinate system, we can find an upper bound on the depth comparison error ΔZ. S can be defined as:

$$ax + by + cz + d = 0$$

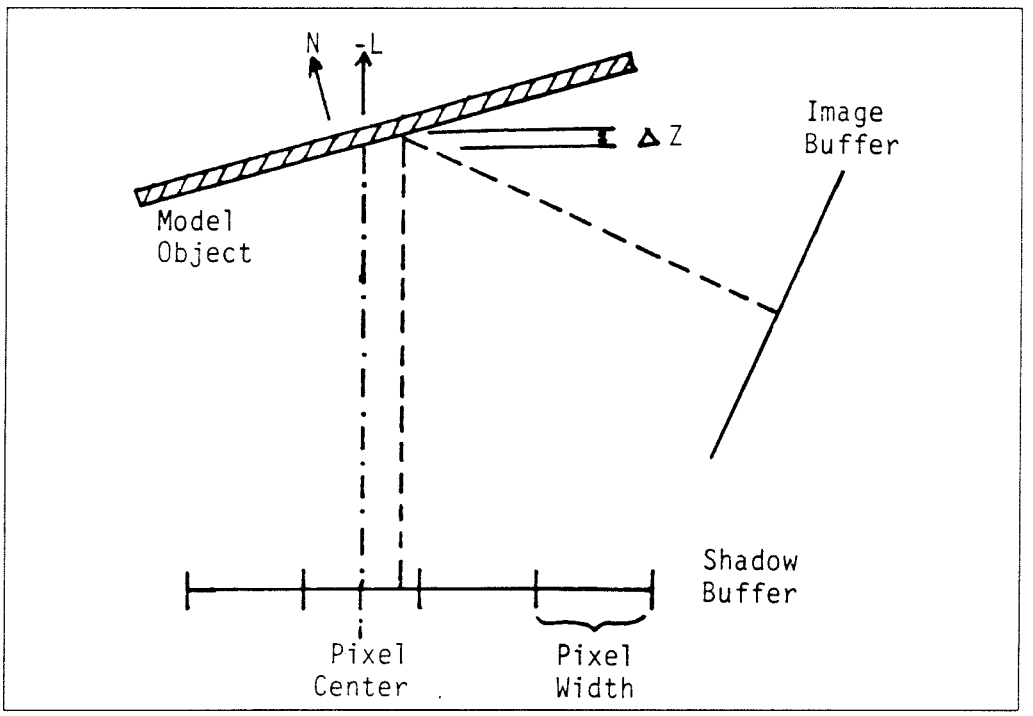


Figure 5.2: Transformation between coordinate system leads to sampling problems with the shadow buffer algorithm.

The value of z is sampled on each shadow buffer pixel center (x', y'). Since S is planar, the worst case depth occurs on one of the shadow buffer pixel's corners, ie. $(x' \pm 0.5, y' \pm 0)$. The depth at a corner, z'_c can be computed as:

$$z'_c = z' \pm \frac{1}{2}\frac{\partial z}{\partial x} \pm \frac{1}{2}\frac{\partial z}{\partial y}$$

Thus the depth error is constrained:

$$\Delta Z \leq \frac{1}{2}\left(\left|\frac{\partial z}{\partial x}\right| + \left|\frac{\partial z}{\partial y}\right|\right)$$

The partial derivatives are easily computed from the surface normal $[a \quad b \quad c]^t$ as:

$$\frac{\partial z}{\partial x} = \frac{c}{a}$$
$$\frac{\partial z}{\partial y} = \frac{c}{b}$$

To compensate for this depth comparison error, I used the technique proposed by Williams. He simply adds a depth bias factor to the transformed depth z' before comparing it with the depth value stored in the shadow buffer. This depth bias must be large enough to prevent objects from shadowing themselves. If it is too large, shadows will appear to have been incorrectly cast. This depth bias can be incorporated in the S_{buf} transformation by modifying the shadow buffer viewport transform, T_{sv}, such that:

$$T_{sv} = \begin{bmatrix} v_{sbx} & 0 & 0 & 0 \\ 0 & v_{sby} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ v_{sdx} & v_{sdy} & z_{bias} & 0 \end{bmatrix}$$

where:

$$v_{sbx} = \frac{\text{sbuf}_{width}}{2}$$

$$v_{sby} = \frac{\text{sbuf}_{height}}{2}$$

$$v_{sdx} = \frac{\text{sbuf}_{width} - 1}{2}$$

$$v_{sdy} = \frac{\text{sbuf}_{height} - 1}{2}$$

Point sampling the shadow buffer produces shadows with sharp transitions between dark and light. It has been suggested that smoother shadow transitions can be computed by sampling the shadow map pixels around the point (x', y') and bilinearly interpolating their values to compute an attenuation factor. This does not seem to have any physical basis since neighboring shadow buffer entries do not indicate that points near (x', y', z') lie in shadow.

Instead of bilinearly interpolating shadow buffer values, *render* point samples the shadow buffer at the four pixel corners, $(x_i \pm 0.5, y_i \pm 0.5, z_i)$ and computes an attenuation factor based upon the average of the shadow buffer values. This technique is not without its problems, i.e. it presumes that the z_i is the correct depth value at the pixel corners, but it is physically more sound than bilinearly interpolating shadow buffer values.

Figure 5.3*a* illustrates an image generated with the shadow buffer algorithm. As the figure illustrates, this algorithm can be made to work reasonably well despite drawbacks inherent to the algorithm. These drawbacks

include sampling problems that introduce depth comparison errors and limited shadow buffer spatial resolution.

Figures 5.3*b* and 5.3*c* illustrate that improper selections of the depth bias value leads to improperly shadowed scenes. Figure 5.3*b* illustrates a scene produced without any depth bias. Notice that surfaces tend to shadow themselves. Figure 5.3*c* illustrates a scene with too large a depth bias. In this scene, shadows do not appear where they should.

Figure 5.3*d* illustrates the problems associated with limited shadow buffer spatial resolution. This figure was produced with a smaller shadow buffer using 256 by 256 pixels instead of the 512 by 512 pixels used for Figures 5.3*a*–5.3*c*. Notice the jaggies apparent in the object's shadow. Careful inspection will show that these jaggies are apparent in both images. Furthermore, notice that the size of these jaggies appears larger in portions of the shadow which lie closer to the viewer. This results from the viewer's perspective transformation which results in a non-uniform sampling of the shadow buffer.

These shadow buffer problems can be combated in several ways. Several techniques can be taken to improving depth comparison. The first technique makes use of the fact that surfaces that are tangential to, or nearly tangential to, the light source direction vector cause the worst depth comparison problems. These surfaces are typically found near silhouette edges and can be culled without much effect on the shadowed image. The *render* implementation currently culls only the surfaces tangential to the light source direction vector.

Other techniques can be used to automatically determine an optimal value for the depth bias term. When a polygon is tiled in the shadow buffer, its error contribution can be determined from its plane equation and the number of pixels it covers. Information about the distribution of error contributions could be maintained allowing a depth bias term to be statistically determined. One might select a depth bias value such that at least a certain percentage of the tiled pixels are guaranteed to be correctly shadowed. This technique has some problems since the error distribution will include contributions from objects that do not end up being visible. This problem could be avoided by tiling objects in the shadow buffer with a visible surface algorithm that only tiles each pixel once.

A better solution might be to try to reduce the source of the depth comparison error problem. If we provide additional storage for each shadow buffer entry, we can store a surface normal for each pixel. Instead of using the pixel's depth value for shadowing depth comparisons, the pixel's depth at (x', y') can now be computed more precisely using the surface normal. There will still be depth comparison errors due to numerical quantization but these effects will be much smaller than the errors introduced by incorrect sampling.

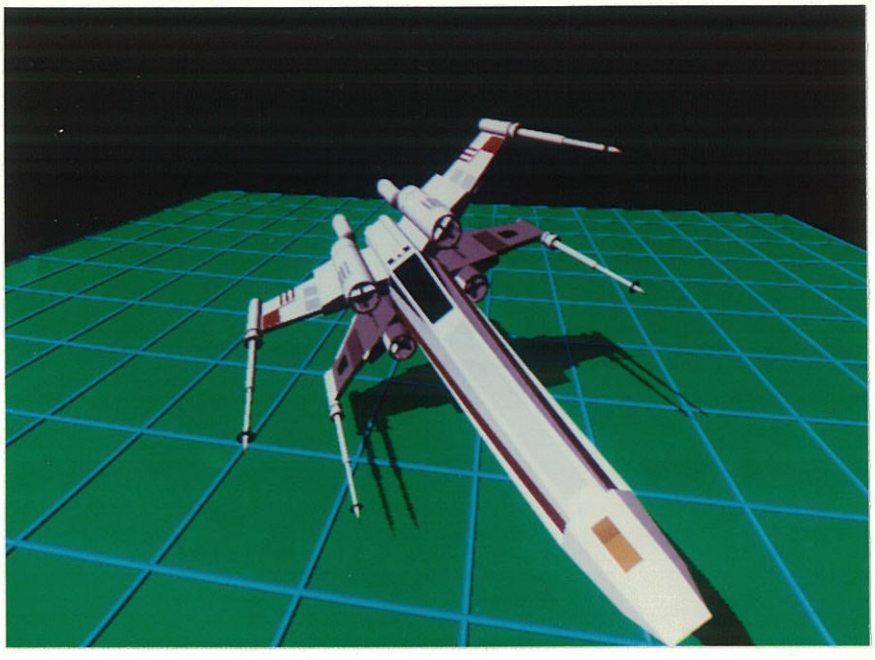


Figure 5.3a: A properly shadowed image created with the shadow buffer algorithm.

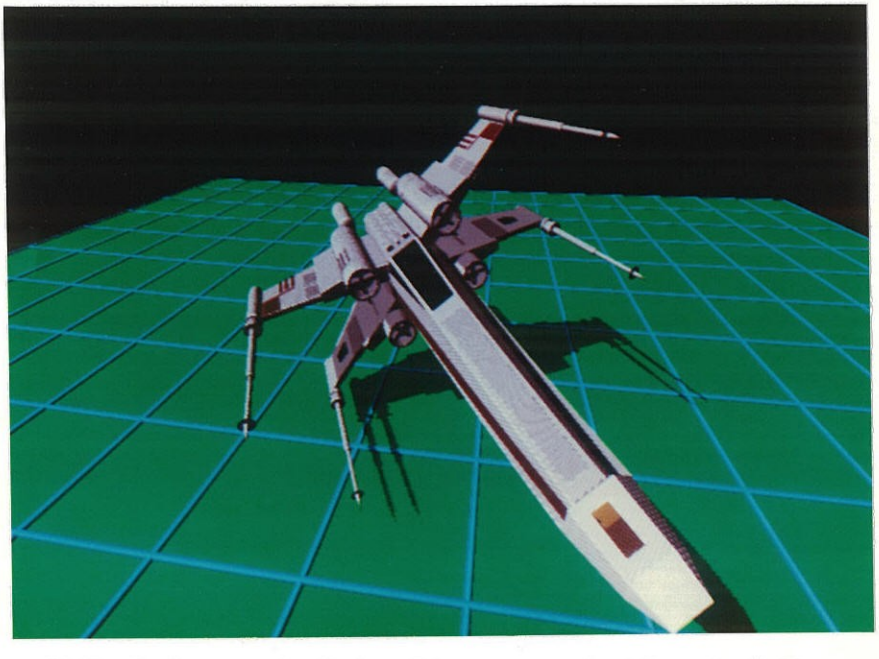


Figure 5.3b: An improperly shadowed image created with no depth bias.

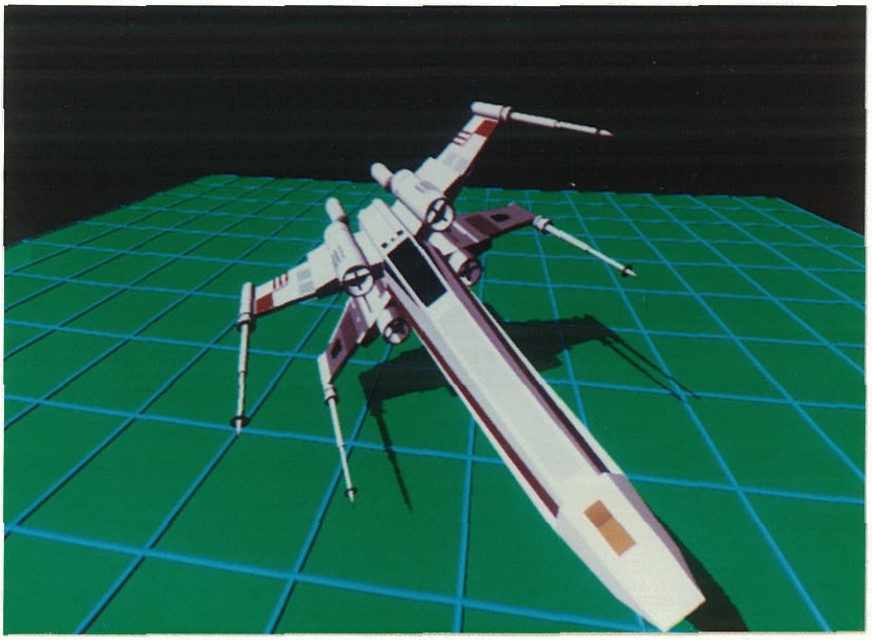


Figure 5.3c: An improperly shadowed image created with too large a depth bias.

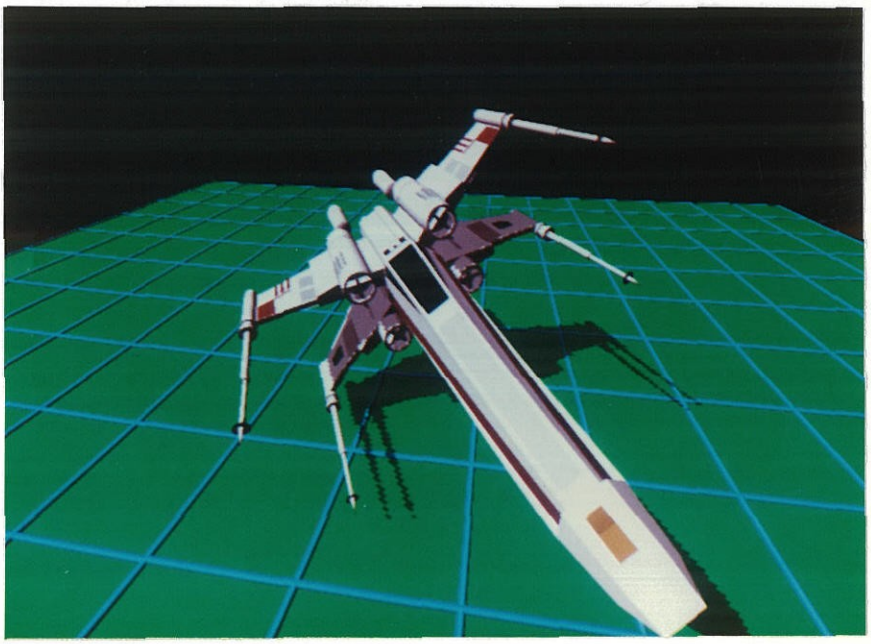


Figure 5.3d: A shadowed image created with a low resolution (256 by 256) shadow buffer.

Several techniques can be used to reduce the amount of additional information that has to be stored for each pixel. The visible surface's normal vector provides more information than is needed. Instead of storing surface normals, we can store the partial derivatives, $\partial z/\partial x$ and $\partial z/\partial y$, which provide all of the needed information in a more usable form. The partial derivatives may be represented as floating point numbers or as limited resolution fixed point numbers to decrease storage requirements.

The amount of storage needed for each pixel can be further reduced. Instead of storing a pixel depth, we can store a surface identifier (SID) in the shadow buffer. The SID is treated as an index into a surface property table. During the tiling and shadowing determination phases, pixel depth can be easily determined from information stored in the table. SIDs could be represented as 24-bit integers which would reduce the total amount of storage needed by the shadow buffer over the depth buffer implementation. Additional storage would be required for the surface property table.

The SID shadow buffer reduces storage requirements by increasing the amount of computation which must be performed to retrieve the pixel depth. Besides reducing storage requirements, the SID shadow buffer can also eliminate the self-shadowing polygon problem. If both image and shadow buffers are represented as SID buffers, then during shadow determination self-shadowing can be avoided by comparing the SIDs of the surfaces in each buffer. If they are identical, the pixel should not be shadowed. This technique only works for scenes composed of planar surfaces since non-planar (curved) surfaces must be allowed to shadow themselves.

The problem of inadequate shadow map spatial resolution has one very obvious solution. Spatial resolution can be improved by increasing the size of the shadow buffer. Increasing the shadow buffer dimensions will increase the amount of time needed to create the shadow buffer image since tiling requires time proportional to the image surface area.

In the *render* implementation, the additional storage required for the shadow buffer caused a marked increase in the number of observed page faults. This page faulting caused the shadowed image computation to require much more than twice the real-time needed to create non-shadowed depth buffered images. Increasing the size of the shadow buffer makes the page faulting problem even worse.

The page faulting behavior associated with shadow buffers could be managed by dividing the shadow buffer into tiles and swapping these tiles onto disk. We would expect page faulting behavior to improve because less memory would be required. Swapping shadow map tiles causes problems during the shadow determination phase during which the shadow map is sampled randomly. In the non-tiled shadow buffer implementation, this activity results

in a lot of page faults. In a tiled shadow buffer implementation, this random sampling would result in a lot of tile swapping. Tile swapping is effectively the same as page faulting but must be managed by the *render* program. It is sometimes useful for a user program to manage its own swapping but usually only when the swapping behavior is predictable.

The shadow buffer algorithm is not currently implemented within the ANIMAC simulation. It still exists within the *render* framework but was not selected for use in the simulation because of the aforementioned problems. The depth comparison error could have been improved in the ways mentioned, but without these improvements there were serious concerns as to how shadowed scenes, created with the shadow buffer algorithm, might animate. A particular depth bias value might be adequate for one frame and inadequate for the next.

Another reason for not selecting the shadow buffer algorithm for use in the simulation was the general inefficiency of the implementation. Even with small shadow buffers, *render* page faulted too much, suggesting that it might be inappropriate for the ANIMAC simulation which would make even larger memory demands.

It should be noted here that the shadow buffer algorithm does have an advantage when implemented in hardware. In the creation of the shadow buffer image, the shadow buffer algorithm is creating an implicit local shadow map for that processor. A hardware implementation could capitalize upon this by using less hardware to create the local shadow map. Hardware implementations would also have to provide for much larger shadow buffers than the *render* implementation provided since it would be necessary to have the shadow buffer dimensions be equal to the local shadow map dimensions. Shadow map dimensions are often much larger than shadow buffers, but can be implemented with much less storage than shadow buffers.

5.2 Implementing the Shadow Volume Algorithm

Dissatisfaction with the shadow buffer algorithm encouraged the author to explore implementing a depth buffer variant of Crow's shadow volume algorithm [CROW77A]. It was expected that this algorithm would produce more accurate shadows and with fewer problems simply because this algorithm is a object space shadowing algorithm. It was also expected that the shadow volume algorithm would require more computational time than the shadow buffer algorithm since it determines shadow boundaries much more precisely. It was hoped that the shadow volume implementation would not page fault as severely as the shadow buffer algorithm and thus be more practical to use.

The shadow volume algorithm determines whether a point on a visible surface lies in shadow by determining whether that point lies within any shadow volume. Shadow volumes are polyhedra whose faces are composed of shadow

polygons. These shadow polygons may be labeled as either frontfacing or backfacing depending upon whether the polygons face towards or away from the observer.

Figure 5.4 illustrates the shadowing environment that must be considered in determining whether a point P lies in shadow. The figure illustrates a *pixel ray*, OP, which was constructed to have the observer's viewpoint O as its origin and pass through the visible point P. The figure shows that shadow polygons may intersect this ray. Given this environment, we can decide if P is in shadow by inspecting the half open interval [OP). If in this interval, we find that there are more shadow polygon-ray intersections with frontfacing shadow polygons than with backfacing shadow polygons, then the point P lies in shadow because it must lie in the interior of a shadow volume.

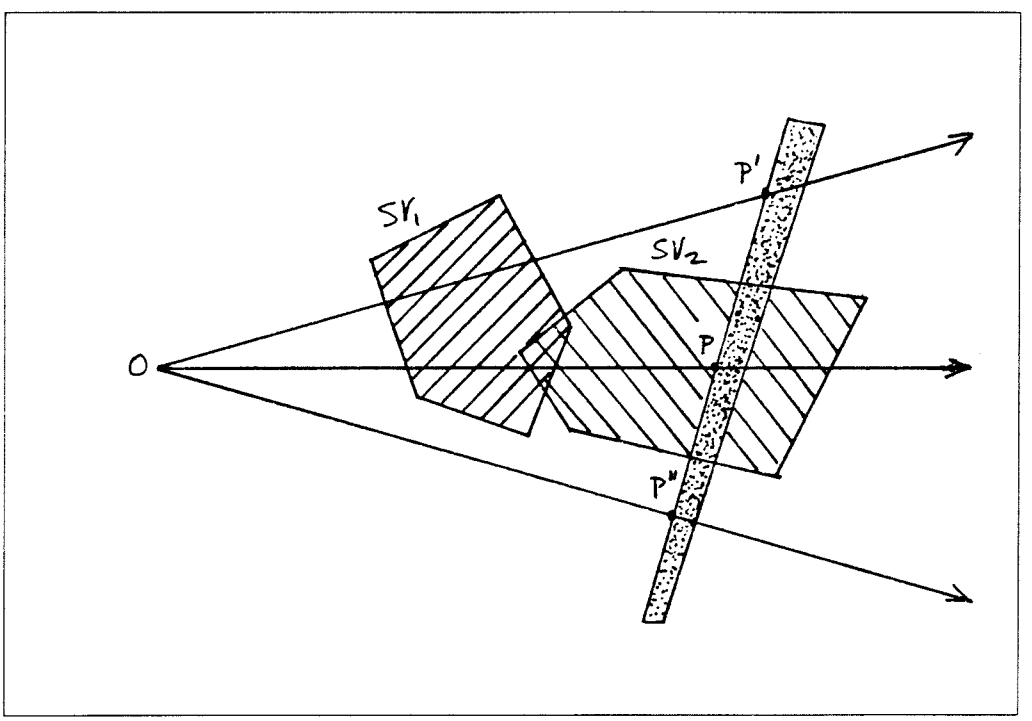


Figure 5.4: Shadow volume environment

In his description of the shadowing criteria, Crow worried about a special case that occurs when the shadow volume pierces the near clipping plane. In this case, there may not be a frontfacing shadow polygon. We eliminate this special case from consideration during shadow determination by requiring that the shadow volume clipping algorithm recognize this condition and generate an appropriate frontfacing shadow polygon.

This shadow volume algorithm can be implemented in a depth buffer framework by allocating an additional buffer along with the depth buffer. This buffer, the *sv-buffer*, provides a special counter for each pixel. Shadowed images are created by first generating a visible surface image. A second pass generates shadow volumes for all of the objects, storing them in a temporary file. Shadow volumes are later read back from this file and clipped against the viewing volume to produce shadow polygons. These shadow polygons are then tiled in the *sv-buffer*.

A special tiler is used to tile shadow polygons. This tiler incrementally calculates the shadow polygon depth for each pixel in the shadow polygon's interior. If the tiler finds that the shadow polygon depth lies in front of the image buffer depth, then depending upon whether the polygon is front or back facing, the tiler either increments or decrements that pixel's *sv-buffer* counter.

After all shadow polygons have been tiled in this manner, pixels that are in shadow will have counters that are positive. A post-processing pass attenuates the colors of all pixels that are found to be in shadow. This post-processing pass linearly scans the *sv-buffer*. This localized behavior should result in fewer page faults than the random sampling activity used by the shadow buffer algorithm.

If shadow volume tiling is considered an atomic operation (that is all of a shadow volume's shadow polygons are tiled before another shadow volume is started) the number of bits needed to represent the shadow polygon counter can be limited, reducing the amount of storage needed to implement the *sv-buffer*.

The tiling of a shadow volume results in a net counter change of $+1$ if the pixel lies within the shadow volume or of 0 if the pixel lies outside the shadow volume. Tiling a shadow volume can only mark a pixel as being in shadow, it cannot remove a pixel from shadow. This is physically intuitive because shadowing can be viewed as a subtractive process; once light has been removed by an object occluding the light source, adding or deleting other objects can not change whether a point lies in shadow.

The atomic tiling of shadow volumes allows us to state that a pixel's shadow polygon counter may increase by at most one count after a shadow volume is tiled. If we are careful, we can guarantee that during the tiling of a shadow volume's shadow polygons the counter needs only to have a dynamic range of $[1, -1]$. This means that if the counter has an initial value i, then

during the tiling of a shadow volume, the counter value can never exceed $i+1$ nor can it be less than $i-1$. Furthermore, when tiling a shadow volume sets the counter value to 1, no further work need to be done to that pixel since its outcome has been determined.

The *sv-buffer* counter can be represented with only four states. Three of these states represent the counts -1, 0, and 1. The fourth state represents all of the counts greater than 1. This four state representation allows the shadow volume algorithm to be implemented with only an additional two bits per pixel.

It can be easily shown that the counter need only have a dynamic range of $[-1,1]$ during the tiling of a shadow volume. The shadow volume clipper clips a shadow volume against the viewing volume creating a closed polyhedron whose faces are the shadow polygons. The intersection of this polyhedron and any plane is a closed curve or in degenerate cases either a line or a point. Figure 5.5 illustrates the intersection of a shadow polyhedron with a plane that contains the ray OP. Since the shadow volume is a closed polyhedron with non-intersecting faces, the ray OP may only enter the shadow volume by piercing a frontfacing shadow polygon. Once the ray enters the shadow volume, it must exit through a backfacing shadow polygon. Moving down the ray from its origin O towards P, we must observe intersections with faces whose sign alternate. If shadow polygons are tiled in either a front to back or a back to front depth ordering then the counter need only have a dynamic range of $[-1,1]$.

The *render* implementation of this algorithm generates shadow volumes for each shadow casting polygon. Figure 5.6 illustrates the procedure used to generate shadow volumes and to clip them against the viewing volume.

Given a visible shadow casting polygon p with vertices $v_0,\ldots,v_n$ and a light source direction vector, L, we construct another set of vertices $b_0,\ldots,b_n$ by projecting the polygon's vertices some distance d along $-L$ such that:

$$b_i = v_i - dL$$

These vertices are projected along $-L$ because the shadow volume is cast away from the light source direction. We can construct the shadow volumes side faces, $s_0,\ldots,s_n$ as quadrilaterals such that a side face has the vertices:

$$s_i = (v_i, v_{\mathrm{succ}(i)}, b_{\mathrm{succ}(i)}, b_i)$$

where:

$$\mathrm{succ}(i) = \begin{cases} i+1, & \text{if } i \neq n; \\ 0, & \text{otherwise.} \end{cases}$$

Two end polygons are needed to close the polyhedron. One end polygon is the shadow casting polygon p, the other is b which consists of vertices $b_0,\ldots,b_n$.

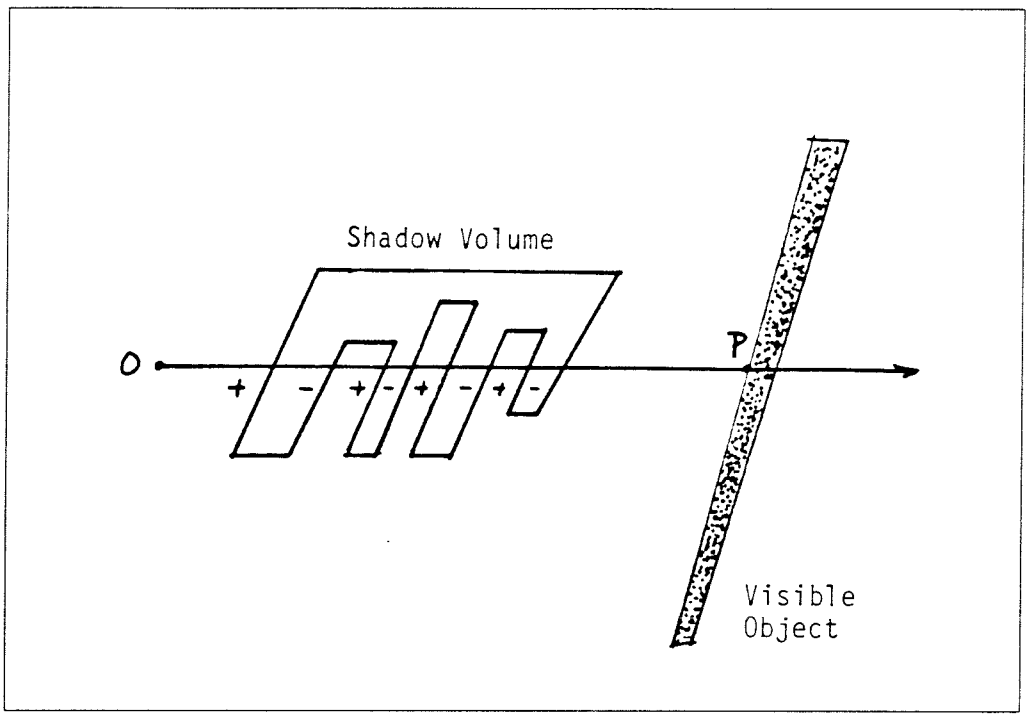


Figure 5.5: The ray OP alternatively passes through frontfacing then backfacing shadow polygons.

The vertices b_i were constructed by projecting the vertices v_i a distance d along L. Since the shadow volume will be clipped against the viewing volume, the distance d need only be large enough to guarantee that the end face b is outside the viewing volume.

The clipping of shadow volumes against the viewing volume can take advantage of certain properties. First, pixel rays never penetrate the four planes that make up the side walls of the viewing volume. This means that the clipper need not generate shadow polygons when the shadow volume exits the viewing volume through these four walls.

Second, the shadow casting polygon p need not be tiled as a shadow polygon because it can never intersect any pixel ray in the interval [OP) (P must be on p or closer to the viewer since p was rendered in the image depth buffer). Thus p can have no affect upon shadowing and can be culled.

Third, if the shadow volume penetrates the near clipping plane, a frontfacing shadow polygon must be generated otherwise the shadow determination scheme will not work. This frontfacing shadow polygon can be computed as b

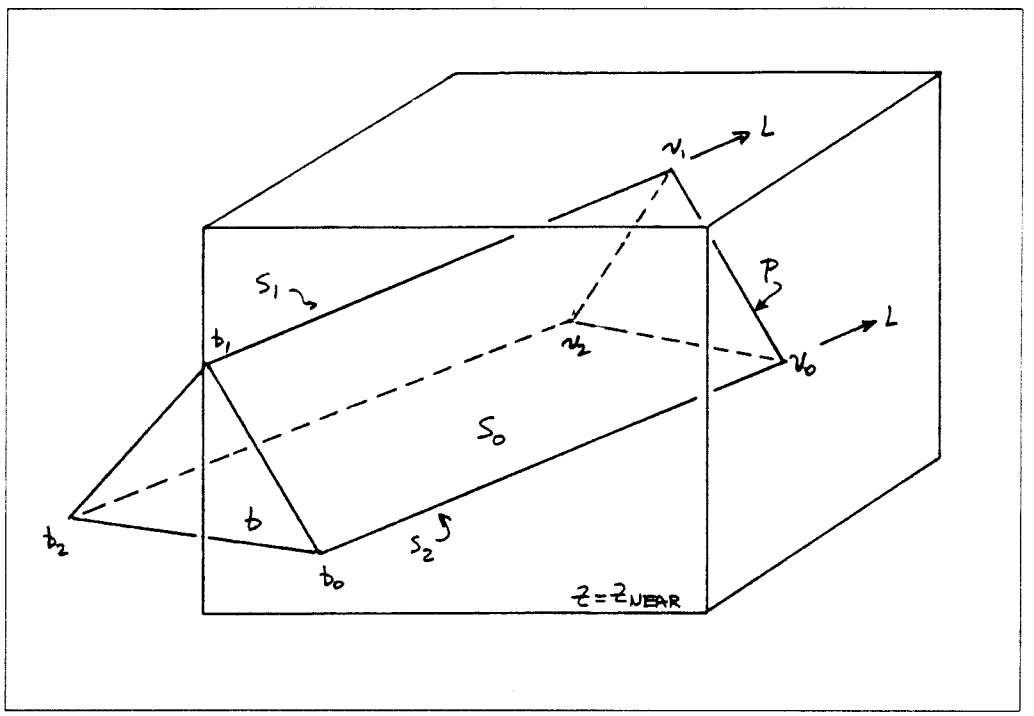


Figure 5.6: Constructing a shadow volume from a visible shadow casting polygon.

if we alter the definition of b to make it coplanar with the near clipping plane ($Z = Z_{near}$).

$$b_i = v_i - d_i L$$

The distance d_i is determined by:

$$d_i = \begin{cases} (v_{i_3} - Z_{near})/L_3, & \text{if } L_3 > 0; \\ d, & \text{otherwise.} \end{cases}$$

Face b only needs to be considered if it is created in the Z_{near} plane since only then can it affect the *sv-buffer* counter state. If b is not in the Z_{near} plane then the shadow volume exits the viewing volume through planes other than the Z_{near} plane. Since *pixel rays* cannot intersect any of the side planes, no shadow polygons coplanar with the side planes need to be generated by the shadow volume clipper. Shadow polygons coplanar with the far clip plane ($Z = Z_{far}$) need not be generated either because such a shadow polygon would intersect a *pixel ray* at Z_{far} which cannot be in the half-open interval $[0, Z_{far})$.

If we construct b as advocated, the shadow volume can be clipped against the viewing volume by clipping the side faces, s_i, and b, when coplanar with Z_{near}, against the viewing volume. The clipping of these faces is independent of each other and is performed with an ordinary polygon clipper [SUTHER74].

Before the shadow polygons are tiled, they must be inspected to see if they are front or backfacing. Face b is always front facing when it is drawn. The faceness of the side faces can be determined by computing a surface normal vector, N_i, for each face and transforming N_i into perspective space, N_P. If $N_{P_3} < 0$, then the polygon is determined to be frontfacing. If $N_{P_3} > 0$, then the polygon is determined to be backfacing. When $N_{P_3} = 0$, the polygon is tangential to the *pixel rays* and should be discarded.

N_i can be computed from the cross product of its edge on p with the light direction vector L such that:

$$N_i = (s_{\text{succ}(i)} - s_i) \times L$$

When using this technique, one must be careful that L does not lie in the plane p (these shadow volumes can be culled) and that N_i is consistently calculated for each shadow polygon.

Consider a second polygon, p', which has vertices $v_n \ldots v_0$. The shadow polygons computed for p' will be the same as the ones computed for p except that the order of polygon edges will be reversed. This will cause N_i to be computed differently for these two shadow volumes. Clearly, this is improper behavior and N_i needs to be redefined as:

$$N_i = \begin{cases} (s_{\text{succ}(i)} - s_i) \times L, & \text{if } p \text{ is clockwise} \\ -(s_{\text{succ}(i)} - s_i) \times L, & \text{otherwise} \end{cases}$$

Render determines a polygon's direction by insisting that all polygons in the model be created in a consistent order. Both viewing and modeling transformations may change the polygon direction, i.e. viewing it from behind. *Render* inspects its transformation matrix to determine whether it needs to flip the polygon's direction.

Extreme care must be taken in the design and implementation of the shadow polygon tiler. Figure 5.7 illustrates some of the tiling conditions that must be taken into account. The job of the tiler is to change the *sv-buffer* state of certain pixels within the shadow polyhedron. The shadow polyhedron consists of faces determined by the clipped shadow polygons and edges that exist between the faces. Edges may be classified as either silhouette or interior. Silhouette edges separate frontfacing and backfacing faces while interior edges separate two frontfacing or two backfacing faces.

As Figure 5.7 illustrates, the tiler must make certain that the pixels along edges are treated properly. For the algorithm to work, pixels on interior edges must be tiled by only one of the faces on the edge. If these pixels are inadvertently tiled twice, shadowing effects will be improperly computed. Pixels along silhouette edges must be tiled by both faces or none of them.

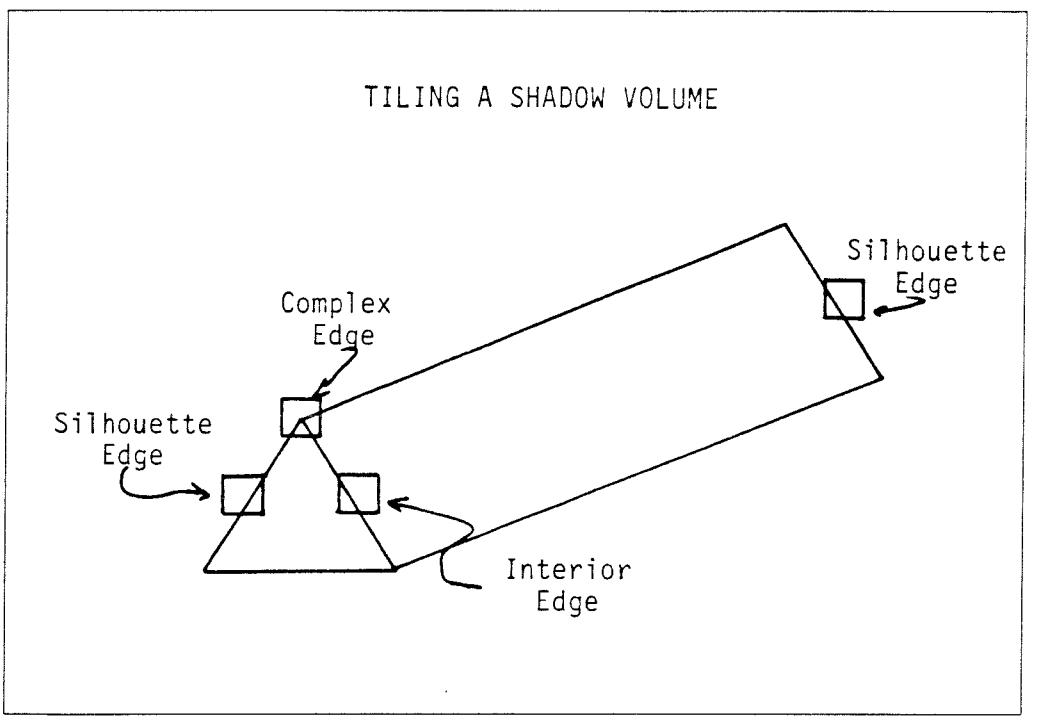


Figure 5.7: The shadow polygon tiler must be designed to account for all the special cases that occur during the tiling of a shadow volume.

The scan-conversion algorithm used in *render* tiles a polygon by calling a tiler routine for each scan-line segment which the polygon intersects. For non-convex polygons, this tiler routine may be called several times during the processing of a scan-line.

The tiler routine receives as arguments: (1) the scan-line number and (2) the x-extents, $[x_{\min}, x\max]$, of the scan-line segment. The tiler routine uses the floating point x-extents to select a range of pixels $[x_l \ldots x_r]$ to tile. The selection of x_l and x_r is extremely important to the functioning of the

tiler. In order that pixels along interior shadow volume edges are properly tiled, the shadow polygon tiler selects:

$$x_l = \lfloor x_{\min} + 1.0 \rfloor$$
$$x_r = \lfloor x_{\max} \rfloor$$

This selection of x_l and x_r values has the effect of removing a pixel from the left of the scan-line segment. When adjacent polygons are tiled, the pixels along a shared edge are tiled by the polygon to the left of the edge. Pixels along a shadow volume's left silhouette edge are not tiled while pixels along the right silhouette edge are tiled twice, once for the frontfacing polygon and again for the backfacing polygon. Narrow scan-line segments which do not cross pixel centers, i.e. $x_l > x_r$, are not tiled.

This tiler correctly tiles shadow volumes in that pixels along interior edges are tiled only once while pixels along silhouette edges are tiled twice or not at all. The only shortcoming this tiler might be considered to have is that since it does not tile pixels along a shadow volume's left edge, these pixels cannot appear in shadow because of this particular shadow volume. This behavior potentially strips shadows of their left-most pixel. This behavior is perfectly acceptable since it only happens on the boundaries of cast shadows and not in the interior.

After the tiler routine decides which pixels to tile, it compares the depth stored in the depth buffer with the depth of the shadow polygon at that pixel. If the shadow polygon is closer to the observer than the visible surface stored in the depth buffer, the *sv-buffer* entry is updated.

Sv-buffer entries are stored as one of four states. The tiler uses a lookup table to implement the state transitions illustrated in Table 5.1. Different lookup tables are selected depending upon whether the shadow polygon is frontfacing or back facing.

SV-Buffer State Transitions		
Current State	Next State (+)	Next State (-)
Minus1	Zero	Minus1*
Zero	Plus1	Minus1
Plus1	InShadow	Zero
InShadow	InShadow	InShadow†

* (Non-occurring state transition)

† (Once InShadow, stay InShadow)

Table 5.1: SV-Buffer state transition table. The + transition is used for frontfacing shadow polygons and the − transition is used with backfacing shadow polygons.

The implementation of the shadow volume algorithm has been extremely successful. It is capable of generating properly shadowed images without any noticeable artifacts. Figure 5.8 illustrates several shadowed images produced with this algorithm.

As was expected, the shadow volume algorithm required more CPU time than the shadow buffer algorithm. This results from having to tile many more polygons than the shadow buffer algorithm. Not only are there more polygons to tile but these shadow polygons are larger since they extend to the boundaries of the viewing volume. The tiling of these polygons is simpler than the tiling of visible polygons which compensates for some of the additional work.

It was hoped that this implementation would not page fault very much. Experimental results indicate that this expectation has been realized. The implementation of the *sv-buffer* as two bit per pixel entries has certainly helped achieve this performance.

The performance of this implementation could be further improved by creating a single shadow volume for each closed polyhedral object. Currently, *render* creates a shadow volume for each shadow casting polygon. This technique was first suggested by Crow [CROW77A]. It should be implemented as it is likely to have a considerable impact upon shadow polygon tiling time.

5.3 Implementing the Shadow Map Algorithm

The ANIMAC architecture utilizes the shadow map algorithm to determine whether objects in other processors shadow local objects. The *render* implementation of the shadow map algorithm has been divided into several phases. The first phase defines the external shadow map (ESM) coordinate system which serves as a frame of reference for all other shadow maps. The second phase computes local shadow maps (LSM) for each processor. The third phase computes composite shadow maps (CSM) for each processor. The final phase determines whether points lie in shadow by sampling the processor's composite shadow map.

The ESM has been defined to lie in a plane that is perpendicular to the light source direction vector L. The ESM can be thought of as a rectangle in the ESM plane. The extents of this rectangle are defined such that they fully enclose the projection of all objects, within the viewing volume, onto the ESM plane.

A transformation, T_{esm}, that projects objects from viewing space coordinates into ESM coordinates is constructed to provide a frame of reference for creating and sampling shadow maps. *Render* creates this transformation by first using the LView() procedure to compute a viewing transformation, E, which transforms model coordinates into ESM virtual device coordinates. A

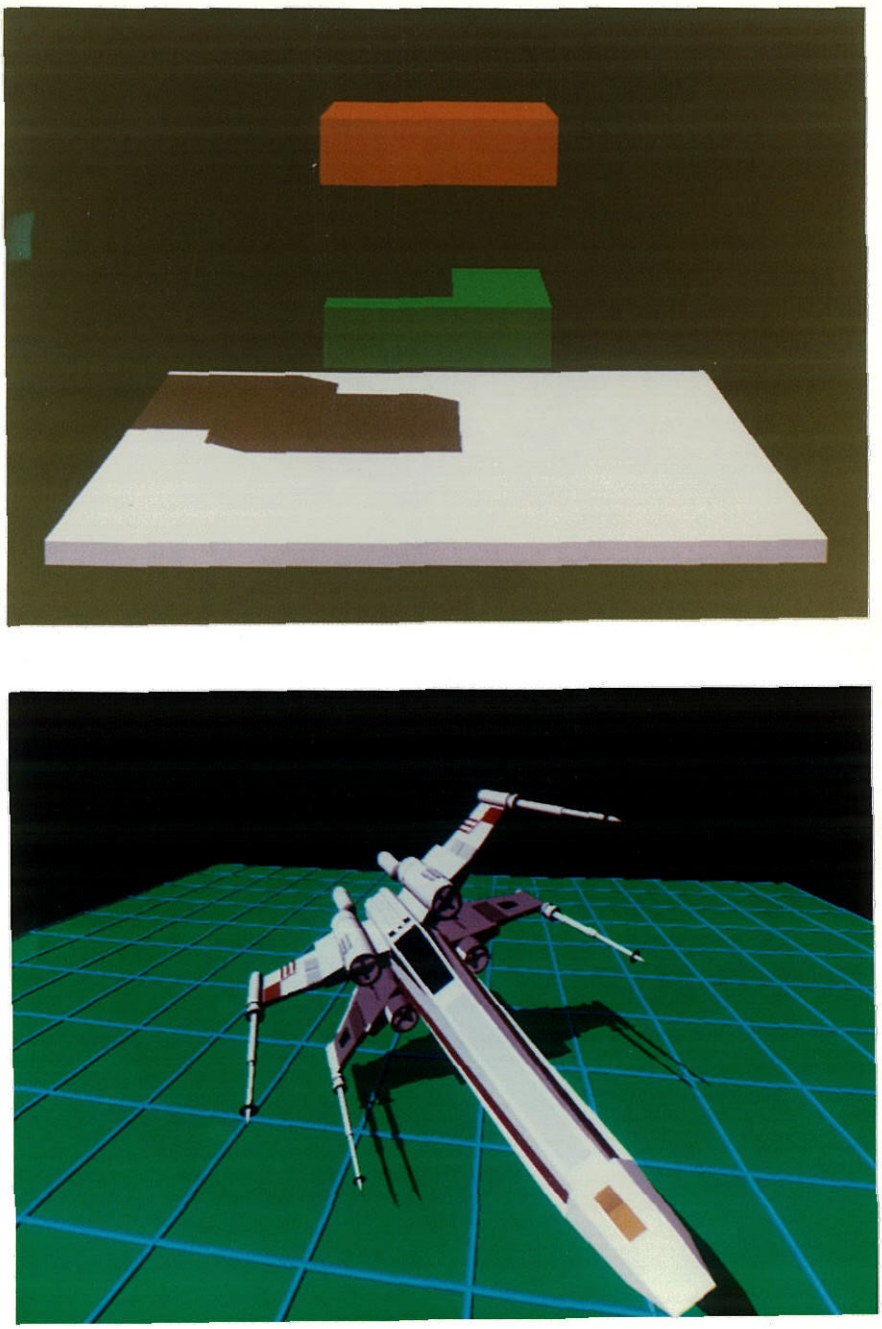


Figure 5.8: Two properly shadowed images created with the shadow volume algorithm.

viewport transform V_{esm} is constructed from the ESM X and Y dimensions. T_{esm} is constructed as:

$$T_{esm} = EV_{esm}$$

The ESM coordinate system allows shadow maps to be created such that they are aligned on ESM pixel centers with each other. This will allow shadow maps to be unioned together without introducing alignment errors.

Local shadow maps must be constructed for each processor. A processor's LSM consists of a bit map and its two dimensional extent on the ESM. *Render* implements a bit map as an array of unsigned integers. Each array entry contains sixteen contiguous pixels along a scan-line. Bit maps are allocated based upon the ESM extents of the shadow map. For efficiency reasons, the shadow map's left edge is aligned with a word (16-bit) boundary. This simplifies the inspection of a shadow map entry and speeds up the unioning of shadow maps.

The extents of a processor's LSM can be determined by transforming a bounding box, bb_p, which encloses the processor's viewing volume subspace by T_{esm}. LSM extents generated in this manner are guaranteed to be large enough to encompass any objects that might reside in the processor. Since a processor's objects often do not occupy the entire subspace, smaller LSMs may be used.

Render constructs smaller LSMs by determining a bounding box, $bb_{p \cap s}$, which encompasses all shadow casting objects within the processor's subspace. This bounding box is simply the intersection of bb_p with a bounding box, bb_s, which encloses all shadow casting objects within the scene model. If $bb_{p \cap s}$ is null, the processor does not need to allocate a LSM. Otherwise, $bb_{p \cap s}$ is transformed by T_{esm} to yield the LSM extents, bb_{lsm}, on the ESM.

If a processor has non-null LSM extents, *render* allocates a shadow map for the processor and creates a transformation, T_{lsm}, which transforms model space coordinates into the LSM bit map coordinates. Bit maps always have their origin at $(0,0)$ requiring ESM coordinates to be translated in X and Y before being used as bit map indices. T_{lsm} is constructed as:

$$T_{lsm} = T_{esm} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -bb_{lsm}.\min_x & -bb_{lsm}.\min_y & 0 & 1 \end{bmatrix}$$

After T_{lsm} has been constructed, the processor's objects are tiled into the local shadow map. *Render* accomplishes this by adjusting its viewing transformation so that the processor's subspace fills the clipping volume. The scene model hierarchy is then traversed clipping objects as they are encountered.

Objects which remain after being clipped against the processor's subspace are then transformed by T_{lsm} and tiled into the LSM.

The shadow map tiler unions objects into the local shadow map by ORing scan-line segments into the LSM bit map. This tiler uses table lookup methods to rapidly tile polygons.

After creating all of the local shadow maps, *render* creates the composite shadow maps. A processor's CSM is formed from the union of its shadowing neighbor's LSMs and CSMs. CSMs must be formed in an order that depends upon the light source direction vector, L. *Render* creates a directed graph which it uses to determine the dependencies between processors.

Figure 5.9 illustrates a processor array and a dependency graph that would result from the indicated light source direction vector, L. Each processor is represented as a node in the graph. Each node may have up to three edges emanating from it. Theses edges point to the processor's neighbors upon which it depends for shadowing information.

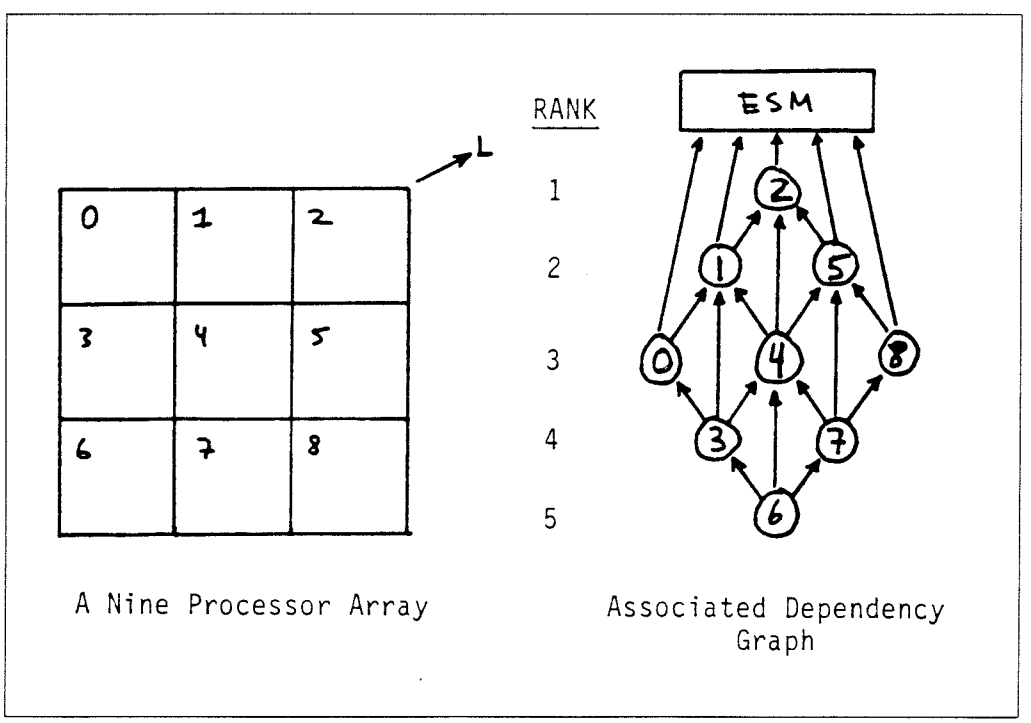


Figure 5.9: A processor array and its dependency graph. The dependency graph is constructed to indicate which of a processor's neighbors potentially cast shadows upon the processor.

Render represents each graph node with three pointers. Each pointer may point to another processor's graph node or may be NULL. A processor's shadowing neighbors can be determined by inspecting the x and y components of the light source direction vector. The following pseudo-code procedure determines which neighbors a processor depends upon.

```
ProcDependency(x, y)
int x, y;
{
      int Column, Row;

      /* Set Column and Row to be outside Processor Array */
      Column = Row = -1;

      /* Find an adjacent column closer to the light source */
      if (L.x > 0.0)
            Column = x + 1;
      else if (L.x < 0.0)
            Column = x - 1;

      /* Find an adjacent row closer to the light source */
      if (L.y > 0.0)
            Row = y + 1;
      else if (L.y < 0.0)
            Row = y - 1;

      /* Set the Node Pointers */
      NodePtr1 = AddressofProc(x, Row);
      NodePtr2 = AddressofProc(Column, y);
      if (NodePtr1 != NULL && NodePtr2 != NULL)
            NodePtr3 = AddressofProc(Column, Row);
}
```

Render constructs the dependency graph by calling ProcDependency() for each processor. AddressofProc() sets a processors node pointer to a neighboring processor. If the coordinates of the neighboring processor are outside of the processor array boundaries, AddressofProc returns either NULL or the address of the ESM processor.

The ESM node must be the terminal node in the graph since it depends upon no other processors. Each node in the graph can be ranked by its maximal distance from the ESM node. A processor's rank determines the order in which CSMs must be created. Since a processor's CSM can only depend upon processors of lesser rank, CSMs must be created in a rank ordering, from smallest to largest.

Render determines a node's rank recursively. The procedure RankProc() is called for each root node in the graph. RankProc() sets the rank of a node to be one greater than the maximum ranks of the processors upon which this node depends. RankProc() calls itself to determine the rank of dependent nodes. The rank of a terminal node (the ESM node) is defined to be 0.

After the nodes in the dependency graph have been ranked, all nodes that represent physical processors have ranks of at least one. *Render* sorts the graph's processor nodes into a list ordered by rank. Using this sorted list, each processor's CSM is created in the proper order.

A processor's CSM is created by first determining the extents of the CSM, allocating a bit map of that size, and then unioning the processor's shadowing neighbor's LSMs and CSMs into the bit map.

The extents of a CSM need be no larger than the projection of a processor's subspace upon the ESM. Previously, bb_p was defined as a processor's extents upon the ESM. The CSM is created as the union of a processor's shadowing neighbors LSMs and CSMs. *Render* determines a CSMs extents as:

$$bb_{csm} = ((bb_{lsm_1} \cup \ldots bb_{lsm_i}) \cup (bb_{csm_1} \cup \ldots bb_{csm_i})) \cap bb_p$$

where bb_{lsm_i} and bb_{csm_i} denote shadow maps belonging to this processor's shadowing neighbors.

After bb_{csm} has been determined, the CSMs bit map is allocated and the LSMs and CSMs are unioned into this processor's CSM. Bit map unioning is performed with a procedure similar to the RasterOp function [NEWMAN79]. A bit map is unioned into another by finding the portion of the bit maps that overlap in ESM coordinates and then ORing one bit map into the other over this two dimensional region. Since both bit maps have been created so that they are word aligned, no bit shifting needs to be done to align the two bit maps. *Render* unions bit maps very fast.

Any particular LSM contributes to at most three CSMs. After these three CSMs have been created, the LSM may be discarded since they are not needed during shadow determination. Likewise, a CSM only contributes to at most three other CSMs. A processor's CSM can only be discarded when the processor has no objects. Since only one CSM is needed at a time during shadow determination, CSMs may be swapped onto disk to reduce memory usage. A good time to swap a CSM onto disk is after it is no longer needed for the creation of other CSMs.

The current implementation discards all the LSMs after all CSMs have been created. It does not currently swap CSMs to and from disk. CSMs reside in the processes virtual address space and although they can require reasonably large amounts of memory, they don't tend to cause excessive page

faulting behavior because at most two CSMs are ever accessed at any one time. Swapping becomes more attractive on machines with limited virtual address spaces.

Composite shadow maps must be sampled to determine whether a pixel lies in shadow. A pixel's image space coordinates can be transformed into CSM coordinates by a simple transformation, T_{csm}, which is constructed as:

$$T_{csm} = T_{pv}^{-1} T_{esm} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -bb_{csm}.\min_x & -bb_{csm}.\min_y & 0 & 1 \end{bmatrix}$$

T_{pv}^{-1} transforms the point into the viewing coordinate system. T_{esm} translates from viewing coordinates into ESM coordinates. The final transform maps ESM coordinates into CSM bit map coordinates.

Transforming the pixel's visible surface coordinate into shadow map coordinates point samples the shadow map. The shadow map pixel can be inspected and a binary decision can be made as to whether the pixel should be shadowed. *Render* shadows pixels by multiplying their color components by an attenuation factor σ which is in the range $[0 \leq \sigma \leq 1]$.

Shadow effects with smoother edges can be obtained by bilinearly interpolating the shadow buffer values associated with the four pixel centers around (X_{csm}, Y_{csm}). This provides an attenuation factor ϕ which varies between $[\sigma, 1]$.

5.4 Simulation of the ANIMAC Algorithm

The implementation of the shadow volume algorithm, for local shadowing, and the shadow map algorithm, for foreign shadowing, together with *render's* depth buffer visible surface algorithm provides all of the utilities needed to simulate the ANIMAC algorithm.

Several test scenes were produced with the ANIMAC algorithm. Figure 5.10 illustrates two types of anomalous behavior that were observed. Objects were observed to shadow themselves along processor boundaries and shadow dropouts were observed in the interior of shadows.

Figure 5.11 illustrates how dropouts and false shadows arise. The figure illustrates a scene composed of two objects being processed by two processors. Object p_1 is oriented with respect to the light source direction, L, such that it casts a shadow upon object p_2. Object p_1 happens to cross a processor boundary and has been bisected into two objects, p_1' and p_1''. The bisection introduces an artificial edge, e.

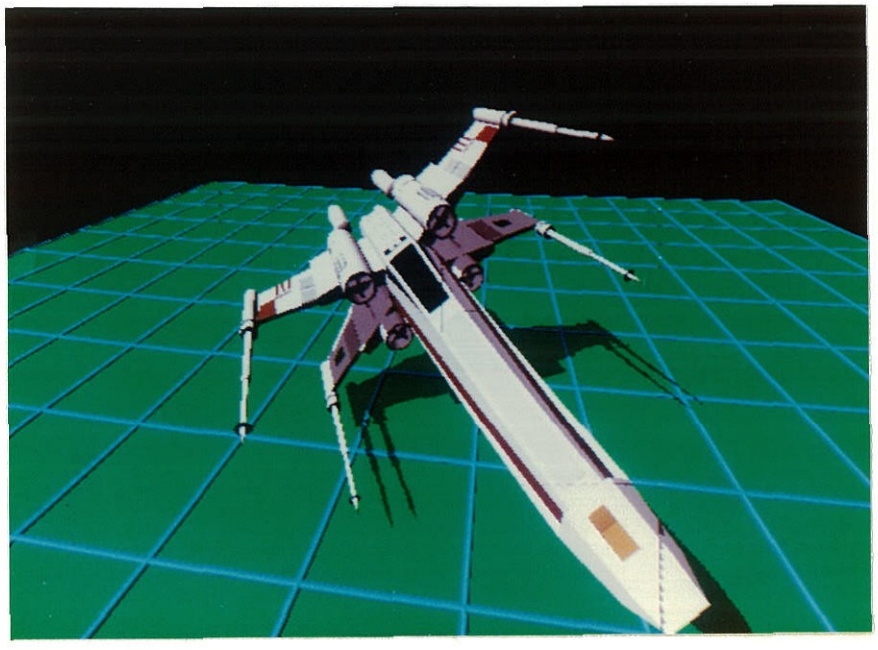


Figure 5.10: This image demonstrates that the initial ANIMAC implementation has two problems: shadow dropouts and false shadows along processor boundaries.

Shadow dropouts can occur on object p_2 near the projection of edge e upon p_2. Pixels to one side of this edge are shadowed by the local shadowing algorithm while pixels on the other side are shadowed by the foreign shadowing algorithm. Shadow dropouts occur when points on p_2 are transformed into CSM coordinates and fall near the projection of e on the shadow map. Some of these pixels fall to the outside of e when they should fall inside. This problem arises because local shadowing was implemented in a different sampling space than foreign shadowing. Increasing shadow map resolution reduces the number of shadow dropouts but can not eliminate the problem. The problem can be eliminated if local shadowing is implemented with a shadow buffer algorithm that samples in ESM coordinates.

The problem of surfaces shadowing themselves along processor boundaries is most noticeable for surfaces that are oblique to the light source and that are bisected by processor boundaries. Illumination rays cast from these surfaces may intersect the same surface in the shadow map causing false shadows. In Figure 5.11, false shadows may occur on object p_1' along edge e. The surface at these pixels may transform onto the edge e in the CSM causing the pixels

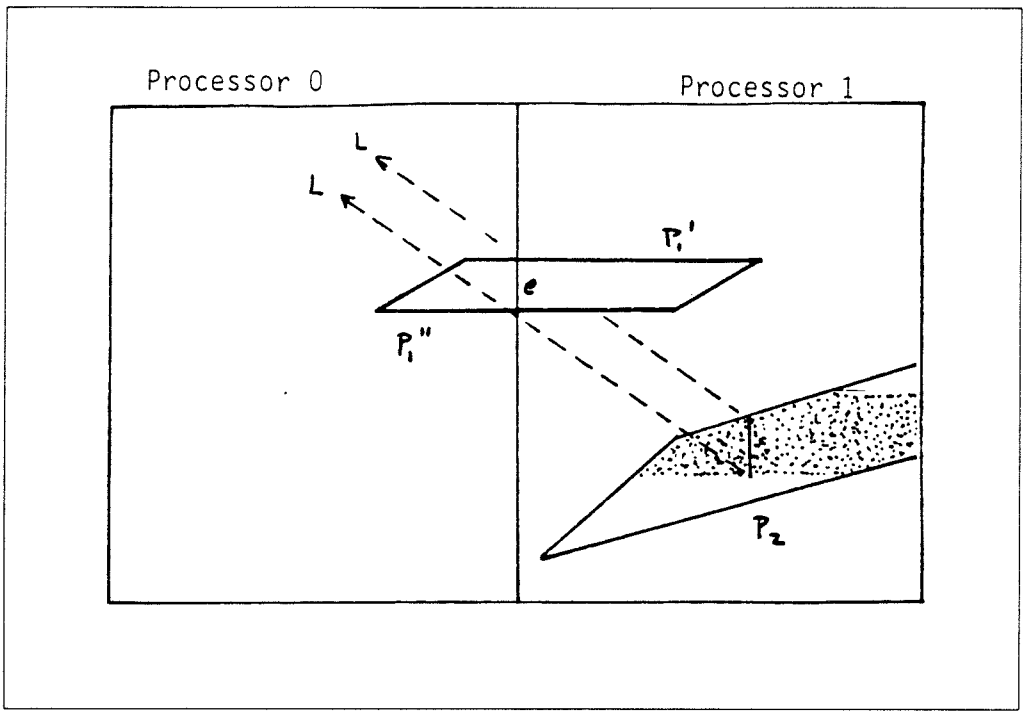


Figure 5.11: Shadow dropouts may result from polygons being bisected by processor boundaries.

to be incorrectly shadowed. These falsely shadowed pixels always project onto polygon edges in the CSM.

Both of these problems are really the same problem which shows itself in two different manifestations. The problem is that shadow maps implemented as bit maps can only be sampled with a precision determined by the bit map resolution. Other realizations of shadow maps might maintain exact descriptions of boundaries. These representations will require much more effort to construct, union and inspect than bit map implementations therefore, we seek solutions that can mask these problems inherent in bit map implementations.

One might be inclined to suggest simple solutions to this sampling problem. For example, the shadow dropout problem arises from illumination rays that barely miss striking the interior of objects in the shadow map. Bloating objects in the shadow map would cause these illumination rays to strike objects in the shadow map; however, at the same time, it would cause objects which shouldn't be in shadow to fall in shadow, i.e. false shadows. Alternatively, shrinking objects in the shadow map could eliminate false shadows but

would introduce dropouts. This sampling problem is not amenable to simple solutions.

Render implements two strategies to mask this sampling problem. It eliminates false shadowing by shrinking the shadow map image. Figure 5.12 illustrates how this solves the false shadowing problem. The figure illustrates two shadow map images. The image on the right has been shrunk by one pixel. The sample point p which fell on an edge in the left shadow map image now must fall outside the polygon in the right shadow map image. The sample point can be moved an arbitrary distance, ϵ, away from the polygon's edge by shrinking the image by ϵ.

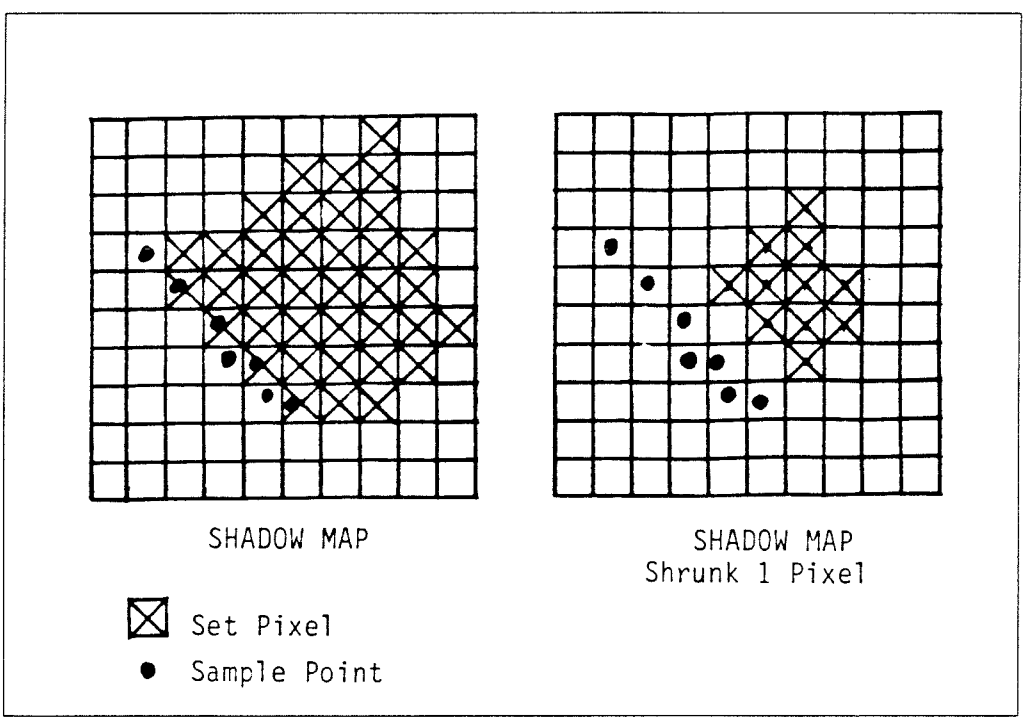


Figure 5.12: False shadows can be eliminated by performing a global shrink on the shadow map image.

Shrinking the shadow map eliminates false shadows but may cause more shadow dropouts to occur. Fortunately, dropouts are easy to remedy. Figure 5.13 illustrates that dropouts that occur due to the bisection of object p_1 can be eliminated by including the entire p_1 object in the local shadowing computations. Pixels that previously had dropouts will now be shadowed by

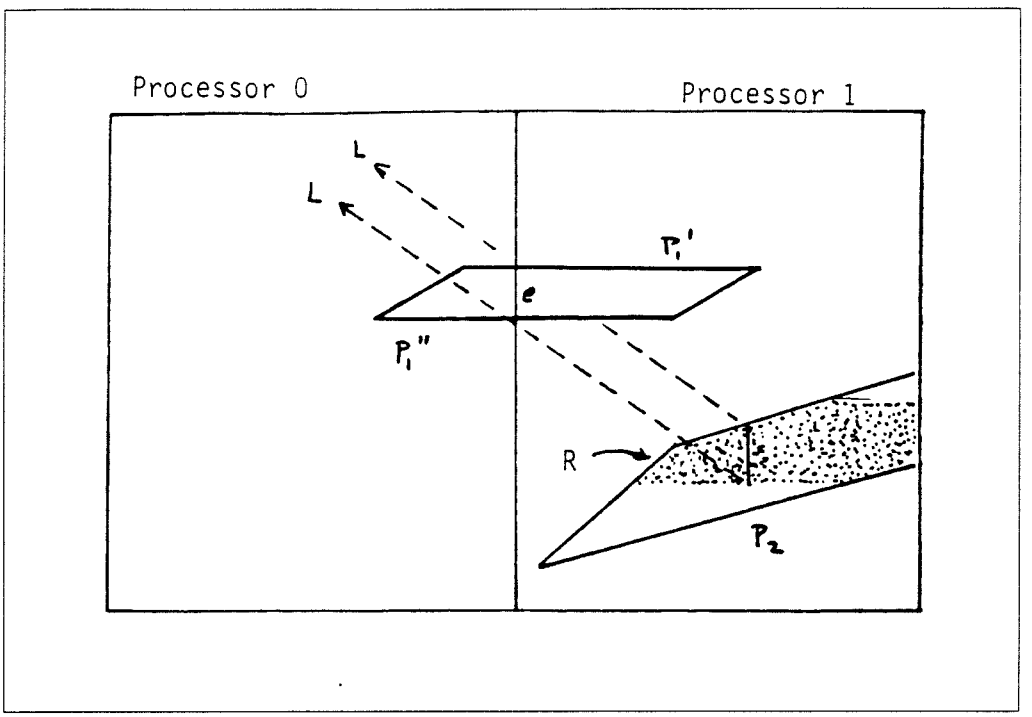


Figure 5.13: Shadow dropouts can be eliminated by including the offending shadow casting polygon in the local shadowing computations. Pixels, in the region labeled R, may be shadowed by both local and foreign shadowing effects.

the local shadowing algorithm. Some pixels will be shadowed by both local and foreign shadowing algorithms which doesn't hurt matters.

The method that *render* uses to make local shadowing regions overlap foreign shadowing regions is an approximate solution. Ideally, any object that is bisected by a processor's boundaries should be included in the local shadowing computation. The current *render* implementation may not include the entire object since it only expands the size of the clipping volume used by the local shadowing algorithm. The *render* implementation makes shadow dropouts highly unlikely but not impossible. In practice, shadow dropouts are not as noticeable as false shadows since subsampling reduces their total contribution to any pixel.

Shadow maps can be shrunk by convolving them with an appropriate filter kernel. This requires time proportional to the the number of shadow map pixels. Since shadow maps are sparsely sampled during shadow determination, the filter can be applied to localized regions of the shadow map during the sampling process requiring only time proportional to the number of image

pixels. A simple filter function, which seems to work fairly well, has been implemented. This filter inspects the pixels that surround a sample point in the shadow map and shadows the pixel only if all of these pixels have been set. This filter effectively shrinks the shadow map image by one pixel since if the sample point lies on an edge, its neighbors will not all be set and the sample point will be deemed to be not in shadow.

To demonstrate the feasibility of the ANIMAC algorithm, test images were made and time spent performing certain tasks was monitored. Figure 5.14 presents two of the test images that were created. CPU time measurements were made to determine how much time was spent in each of four major tasks. These tasks consist of: (1) computing the visible surface image, (2) computing local shadowing, (3) computing foreign shadowing, and (4) shadow determination. Times for computing local and foreign shadowing do not include the time required to determine whether a visible pixel lies in shadow. This time and the time required to attenuate pixel intensities is included in the fourth task. The foreign shadow task was further broken down to study the amount of time spent creating local shadow maps and composite shadow maps.

These CPU measurements are presented in Table 5.2. These CPU time measurements were made on a VAX 11/780 computer. Both images were computed with 2 by 2 subsampling and a 3,000 by 3,000 pixel shadow map. The image of the cubes was computed simulating four processors while the X-Wing image was computed simulating sixteen processors.

Computation	Cubes		X-Wing	
Visible Surfaces	13.17	(6.6%)	349.68	(22.5%)
Local Shadowing	24.87	(12.5%)	704.72	(45.3%)
Foreign Shadowing	9.75	(4.9%)	56.57	(3.6%)
LSM Creation	2.48	(1.3%)	34.43	(2.2%)
CSM Creation	7.27	(3.7%)	22.13	(1.4%)
Shadow Attenuation	150.55	(75.9%)	443.72	(28.5%)
Total CPU	198.33	(100.0%)	1554.68	(100.0%)

Table 5.2: VAX CPU time (seconds) required to compute the images of Figure 5.14 broken down by time spent in computational task.

The image of the cubes is relatively simple and little CPU time was required to determine visible surfaces, local shadowing, and foreign shadowing. Most of the time was spent checking if pixels lie in shadow and attenuating them when necessary. Local shadowing required about twice as much time as visible surface determination. Foreign shadowing required the least time with the majority of that time spent merging LSMs into CSMs.

The X-Wing image is quite a bit more complex than the image of the cubes. Pixel shadowing still requires a considerable amount of time but no longer monopolizes as it did for the image of the cubes. Local shadowing now dominates and requires twice as much CPU time as visible surface determination, roughly the same ratio found for the image of the cubes. Foreign shadowing requires very little time with LSM creation now dominating CSM creation.

A bottom line comparison of the total CPU times indicates that this implementation is reasonably efficient. A simple image like the cube image can be computed in several CPU minutes. More complex images require tens of minutes. The ANIMAC algorithm seems very appropriate for producing animation sequences on conventional computers.

Two short animation sequences were computed to study how well the ANIMAC algorithm synthesizes moving shadows. The first animation piece consisted of a view of the cube image illustrated in Figure 5.14. The model was rotated about its Z axis. Five seconds (150 frames) of animation was produced and studied. The animation showed the shadows moving properly with no apparent anomalies. The degree of aliasing appeared to be no worse in shadowed portions of the image than in non-shadowed regions.

Another five second animation sequence was produced using the X-Wing model. This model is sufficiently complex to illustrate nearly all shadow interplay effects. The X-Wing animation appeared quite natural and no anomalies were noticed by several very astute observers.

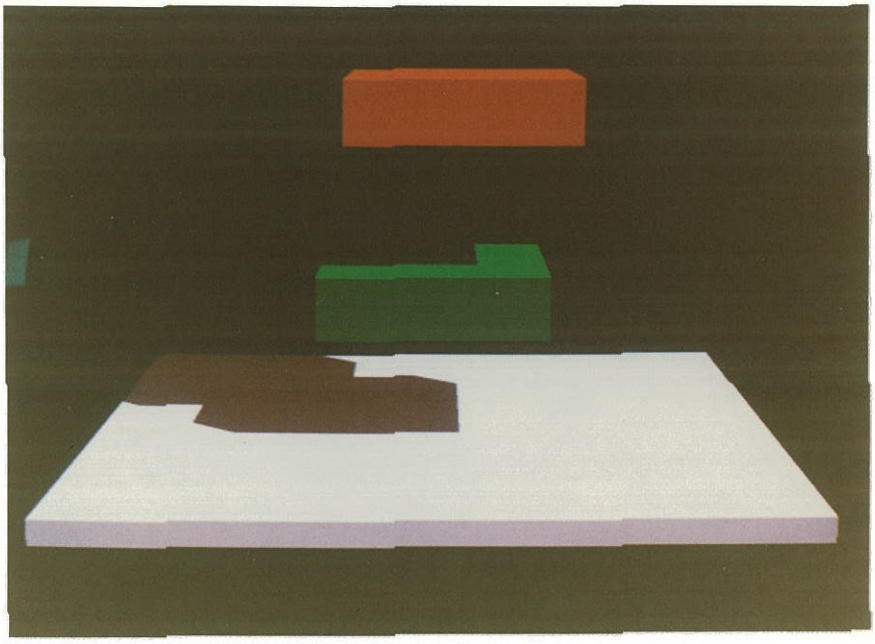
5.5 Conclusions

Implementing the ANIMAC algorithm within the *render* framework has proved to be very worthwhile. It has illustrated some of the shortcomings inherent in image space shadowing algorithms and has allowed the implementation of measures that prevent these shortcomings from visually manifesting themselves.

Incorporating the ANIMAC simulation within a widely used program allowed the algorithm to be tested with scenes that the author did not model. Hopefully, this has helped remove a source of bias from the observations.

The ANIMAC algorithm appears to be quite amenable to uniprocessor software implementations. It provides a natural method for managing the size of the visible surface algorithm by subdividing an image into tiles.

Experiences with implementing the ANIMAC algorithm on a uniprocessor have interested the author in multiprocessor software implementations. The ANIMAC algorithm should be easily ported to nearest-neighbor multiprocessors. If these machines can be made to provide adequate floating point performance, overall scene creation time should be greatly improved.



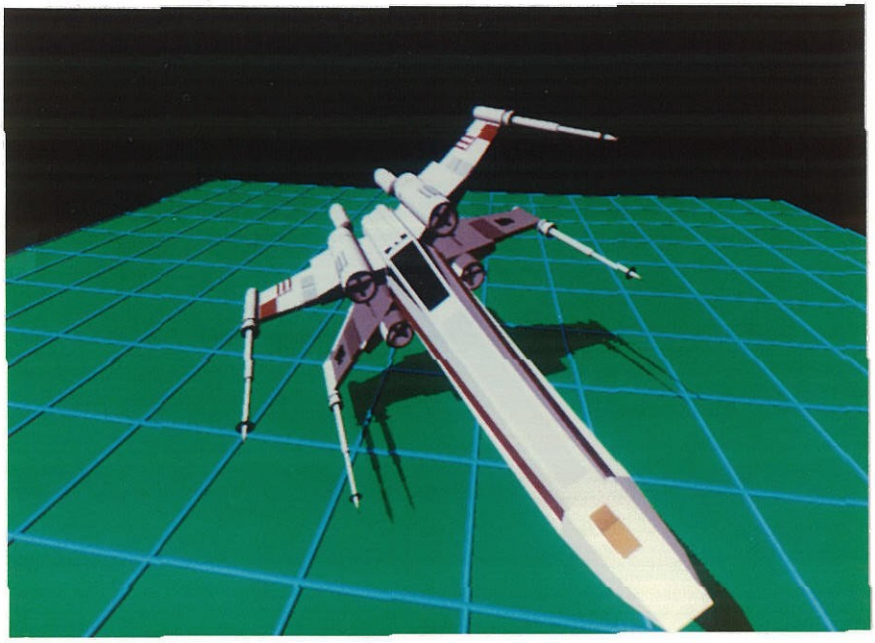


Figure 5.14: Two properly shadowed images created with the ANIMAC algorithm.

In the course of developing the ANIMAC simulation, the author has studied drawbacks inherent in the shadow buffer algorithm and has proposed new solutions to these problems. The author also developed a new depth buffer oriented implementation of the shadow volume algorithm. Both the author's shadow volume implementation and his SID shadow buffer algorithm provide adequate shadowing effects when used with conventional depth buffer visibility algorithms.

6

Hardware Realizations of the ANIMAC Architecture

The purpose of this chapter is to demonstrate that it will be feasible to implement real-time animation systems based upon the ANIMAC architecture. The ANIMAC architecture consists of two facets, a technique for computing subimages in parallel, and a shadowing algorithm designed to work in a parallel environment.

This chapter will focus upon using the ANIMAC architecture to realize two different real-time systems. The first system, the ANIMAC-1, uses the ANIMAC parallelization techniques to implement a high performance real-time animation system. This system produces antialiased shaded visible surface images without shadowing effects. The second system, the ANIMAC-2, extends the ANIMAC-1 architecture to implement shadowing effects.

Performance goals for the ANIMAC implementations have been set significantly higher than what represents today's state-of-the-art in real-time simulation. Today's systems are capable of producing scenes of several thousand polygons at thirty frames a second. The ANIMAC systems have been targeted to produce 512 by 512 pixel images of scenes composed of 100,000 polygons at thirty frames a second.

In order to focus discussion on the important issues, several constraints have been placed upon the scene environment. The scene model is required to be constructed from closed convex polyhedra and each polygon must be a triangle. These conditions are easy to meet. Nonconvex closed polyhedra can be split into closed convex polyhedra and arbitrary polygons can be split into triangles.

The ANIMAC-1 and ANIMAC-2 architectures exploit VLSI architectures to make feasible the construction of such high performance systems. These systems either borrow or extend architectures proposed by other researchers. Knowledge of the processor per object architectures proposed by Cohen [COHEN80] and Weinberg [WEINBE82] and of the processor per pixel architecture proposed by Fuchs [FUCHS81] [FUCHS82] is required to fully appreciate the ANIMAC architectures.

The main emphasis of this chapter is to demonstrate that it will become technologically feasible to build the ANIMAC-1 and the ANIMAC-2 systems in the not too distant future. For this reason, the discussions will focus upon technological bottlenecks. These technological bottlenecks involve being able to implement a computation fast enough or being able to implement the hardware with a reasonable number of integrated circuits. Pipeline techniques will be used to show that computations can be performed within a certain amount of time.

The number of integrated circuits required to implement a computation depends upon the technology that the computation is implemented in. Many forecasts can be found that parameterize future technologies. I present two of these forecast to give an idea of how MOS technologies will evolve during the next fifteen years.

Rideout presents a conservative forecast of MOS technologies for 1990 [RIDEOU81]. He suggests that production integrated circuits will be produced using one micron lithography. He suggests that microprocessors will utilize 250,000 transistors, or about 50,000 logic gate equivalents. This forecast seems very conservative since integrated circuits of this complexity are being fabricated today.

Mohsen [MOHSEN79] described the development of MOS technologies. He summarized his findings by suggesting that in the late 1990's, MOS integrated circuits will be fabricated with $0.3\mu m$ design rules yielding 10^7 devices per square centimeter. He suggested that logic chips will, at that time, be made with millions of logic gates and operate at 100 MHz with a 0.5 volt supply.

These two forecasts provide ball park estimates for the technology that will become available during the next fifteen years. Some of this chapter's discussions will utilize these forecasts to estimate when certain VLSI architectures will become economically practical to pursue.

Both of the ANIMAC systems examined within this chapter make use of the generalized architecture illustrated in Figure 6.1. This system architecture is composed of a modeling subsystem, a clipping subsystem, a visibility determination subsystem, a frame store, and a supervisory processor.

The modeling subsystem is responsible for accessing a data base and generating geometric models of the scene which is to be viewed. It must respond

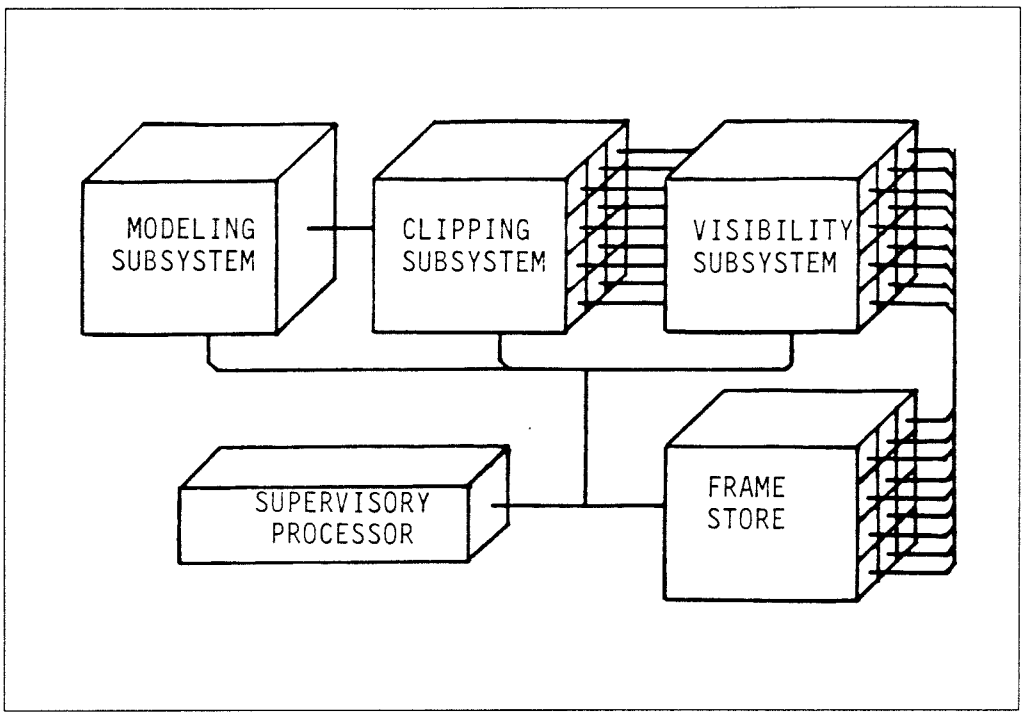


Figure 6.1: Both ANIMAC systems utilize a generalized system architecture which consists of a specialized processors to model the scene, clip the scene, and generate visible images of the scene.

to real-time inputs from the system's users and simulate the task being performed. It must generate geometric data at high rates, i.e. 3,000,000 triangles per second. Specific discussion of the modeling subsystem design is considered to be outside the scope of this thesis, and may very well be a good subject for other doctoral dissertations. At least one commercially available graphics system, the E&S PS300, implements a modeling engine capable of performance within a factor of three of what the ANIMAC systems require, suggesting that implementing the modeling subsystem will be technologically feasible within the near future.

The clipping subsystem receives triangles from the modeling subsystem and distributes polygons to each visibility processor implemented within the visibility subsystem. The ANIMAC architecture implements the virtual tesselated rectangular clipping space subdivision strategy proposed in Chapter 3. Figure 6.2 illustrates how the clipping space is divided among virtual processors, v_{xy}. The clipping space x-axis is equally divided into v_X columns while the y-axis is equally divided into v_Y columns. The total number of virtual

processors is $N_v = v_X v_Y$. Each virtual processor, v_{ij}, is associated with a screen region, $S_{v_{ij}}$, and a subspace of the viewing space, $V_{v_{ij}}$.

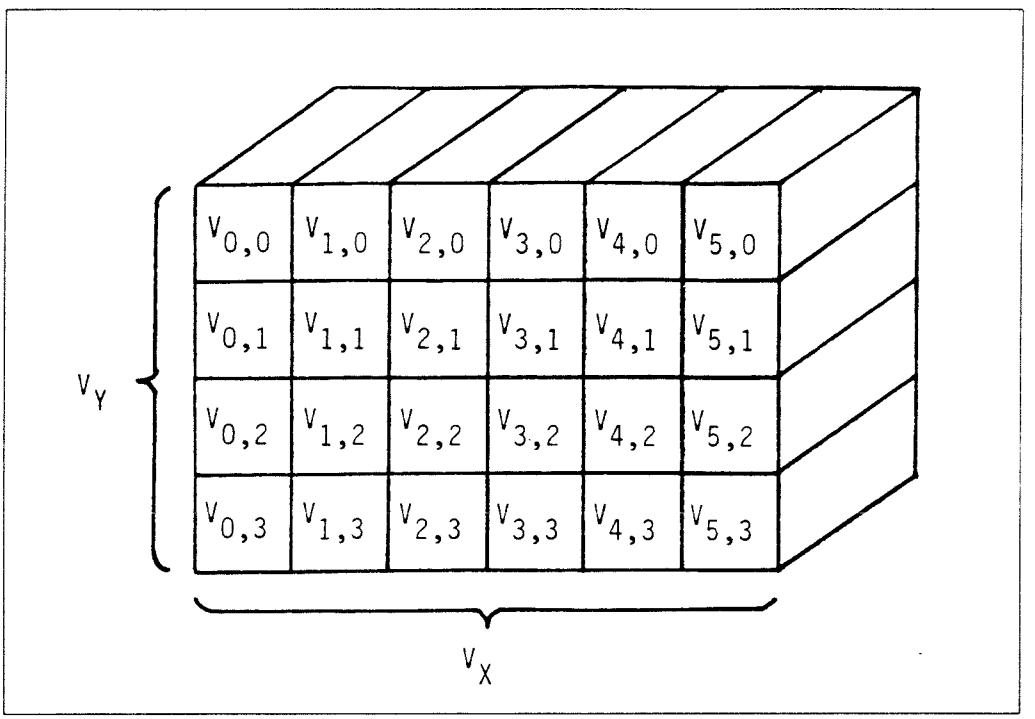


Figure 6.2: The clipping space is divided equally along the x and y axes. A virtual processor is assigned to each of the clipping space subvolumes.

A many-to-one mapping exists which associates virtual processors with physical processors. Each physical processor is responsible for producing a visible surface image for each of its virtual processors. Figure 6.3 illustrates an array of physical processors, p_{xy}. The array consists of N_p processors divided into rows of p_X processors and columns of p_Y processors.

I require v_X to be an integer multiple of p_X and v_Y to be an integer multiple of p_Y. I define ratios of virtual to physical processors as:

$$\sigma_X = v_X / p_X$$
$$\sigma_Y = v_Y / p_Y$$
$$\sigma = N_v / N_p$$

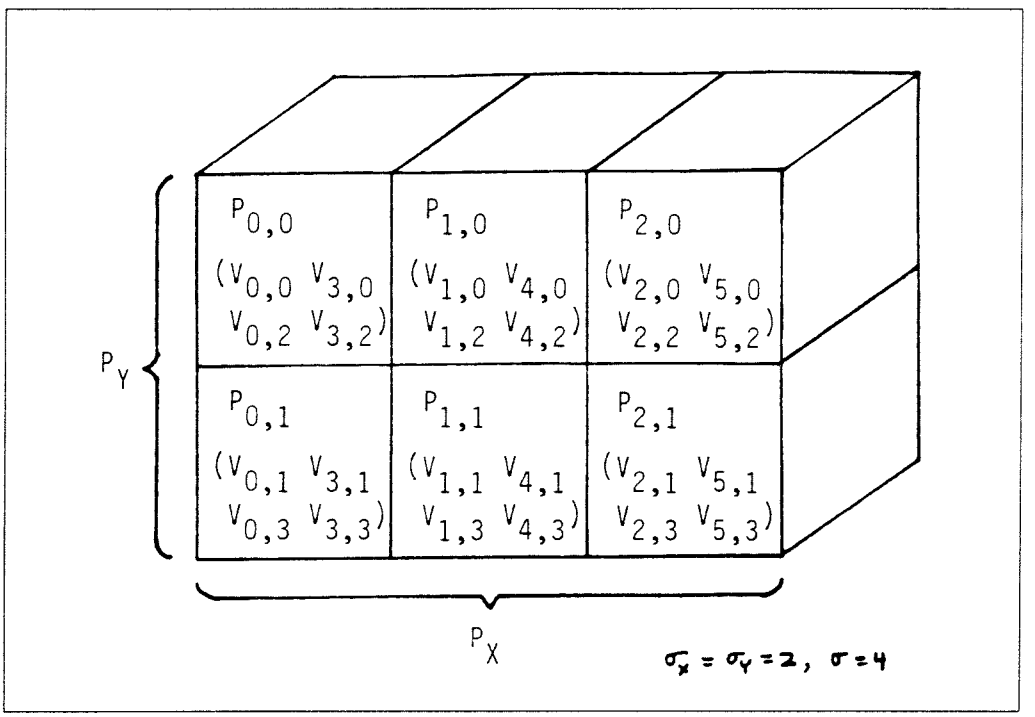


Figure 6.3: An array of physical processors implements the visible surface algorithm for the virtual processors.

A mapping of virtual processors onto physical processors, p_{ij}, is defined as:

$$\bigcup_{k=0}^{\sigma_X-1} \bigcup_{l=0}^{\sigma_Y-1} v_{i+k\sigma_X,j+l\sigma_Y} \mapsto p_{i,j}$$

It is the job of the clipping subsystem to transform polygons into clipping coordinates and to clip an object against each virtual processor's boundaries. Visible polygons are emitted from the clipping subsystem on a separate channel for each physical visibility processor.

The frame store consists of a double buffered image memory. During a frame time, the visibility subsystem writes pixels into one image memory. That image memory is displayed during the following frame time.

The supervisory processor coordinates the activities of all of the subsystems. It only needs to operate at the frame rate and can be implemented with most any conventional processor.

The visibility subsystem consists of a two-dimensional array of visibility processors. The physical dimensions of this array are adjusted to achieve a specific system performance. The two ANIMAC systems are targeted to generate images of scenes consisting of 100,000 triangles. Since the scene is composed of closed polyhedra, about half of these triangles will face away from the viewer and will not be visible. These backfacing triangles are culled by the clipping subsystem leaving 50,000 triangles that need to be handled by the visibility subsystem.

Chapter 3 suggested that these 50,000 triangles will be distributed fairly uniformly to the N_p physical processors. Representing the system's parallel efficiency as η allows us to determine, N_T, the number of triangles that will be distributed to each visibility processor.

$$N_T = \frac{50,000}{\eta N_p}$$

Chapter 3 suggested that a system with sixteen physical processors could be constructed to have $\eta \geq 50\%$ with $\sigma \leq 16$. Substituting these values:

$$N_T = \frac{50,000}{8} = 6,250$$

Thus each of the processors must be capable of processing about six thousand triangles during each frame time. This figure falls near the performance claimed for both the processor per object and processor per pixel architectures so, I chose to implement each of the ANIMAC systems with sixteen visibility processors. Each visibility processor must handle sixteen virtual visibility processors.

The ANIMAC-2 uses shadow maps to determine foreign shadowing. The resolution of these shadow maps is dependent upon the overall ESM resolution and the number of virtual processors. The software simulation indicated that external shadow maps created with about 8 Mb provided adequate resolution. The resolution of individual shadow maps scales with the square root of the number of virtual processors. Since the ANIMAC-2 employs 256 virtual processors, the ANIMAC-2 shadow map resolution has been set at $8Mb/\sqrt{256} = 0.5Mb$ each.

The following sections discuss how the ANIMAC-1 and the ANIMAC-2 visibility subsystems might be implemented.

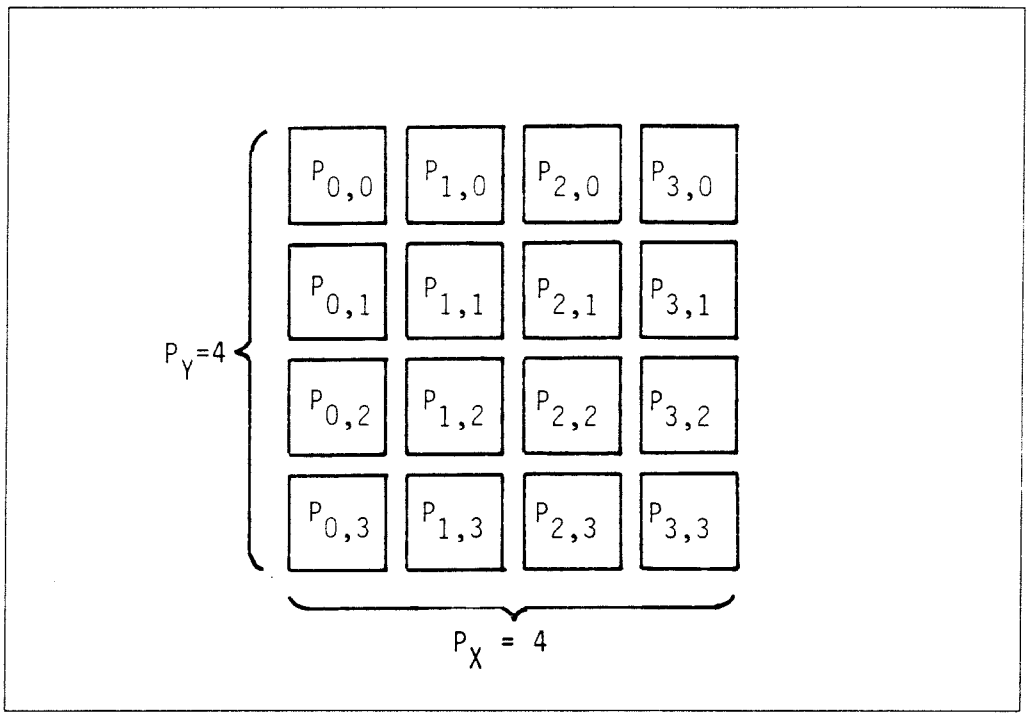


Figure 6.4: The ANIMAC-1 visibility subsystem architecture requires no interprocessor communications.

6.1 Implementing the ANIMAC-1 System

Figure 6.4 illustrates the overall visibility subsystem architecture used in the ANIMAC-1 system. Since the ANIMAC-1 does not produce shadowing effects, no interprocessor communication is required between visibility processors.

Each visibility processor tiles the screen regions, $S_{v_{ij}}$, associated with each of its virtual processors. Figure 6.5 illustrates that a viewporting transformation can be used to make each of a processor's virtual images appear contiguous. A different viewport transformation is needed for each virtual processor and is implemented in the clipping subsystem.

It is advantageous to have a visibility processor process a single image since this allows the clipping subsystem to deliver polygons in any order. If separate images are to be created for each virtual processor, the visible polygons arriving from the clipping subsystem should arrive in an order that is sorted by virtual processor.

Both processor per pixel and processor per object architectures are capable of creating a visibility processor's image in the required time. The ANIMAC-1 system uses a processor per object architecture to implement visible surface

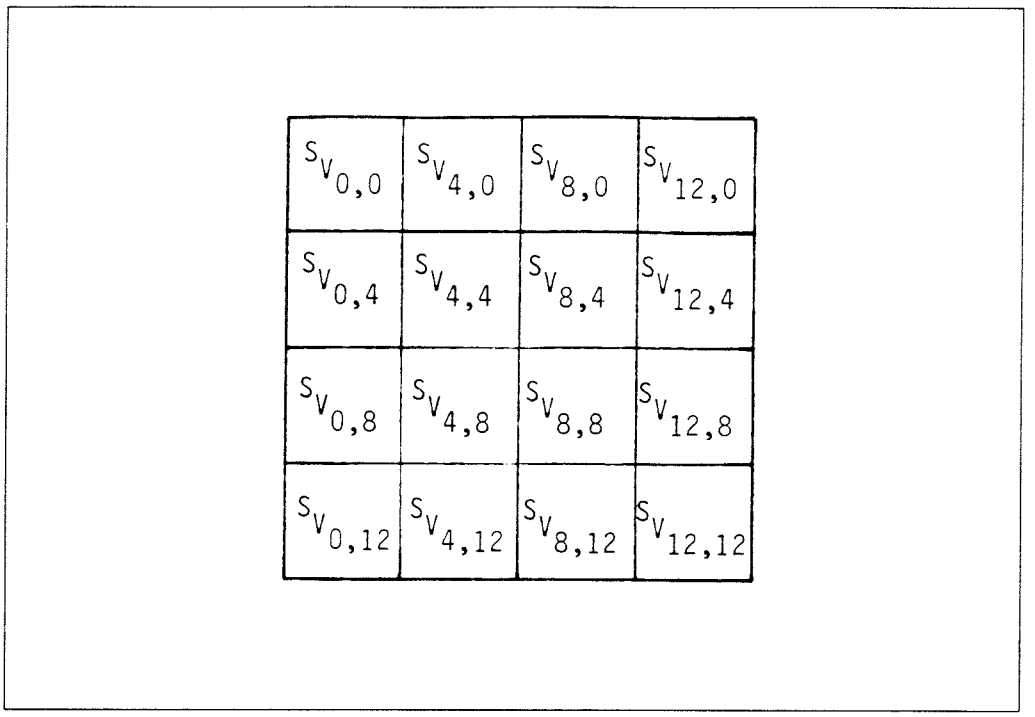


Figure 6.5: The clipping subsystem can implement a viewporting transformation that causes each of a physical processor's virtual processor's screen regions to be adjacent.

determination for two reasons. First, Weinberg [WEINBE82] has shown how to construct a processor per object system which implements effective antialiasing measures that do not require more object processors or faster object processors. Antialiased images can be produced with processor per pixel architectures by increasing the number of pixel processors to subsample the image.

The second reason for using a processor per object architecture in the ANIMAC-1 system is that processor per object architectures require a clock rate that is proportional to the number of pixels to be tiled. Spatial subdivision techniques reduce the number of pixels that each visibility processor must tile and thus reduce the clock rate that the object processors must operate at. This will be shown to make object processors easily realizable in MOS technologies.

Weinberg's processor per object architecture is illustrated in Figure 6.6. It consists of an *Object Preprocessor*, *Object Processors*, and a *Pixel Tiler*. The object preprocessor splits polygons into scan-line aligned trapezoids which are loaded into the object processors. Each trapezoid is assign a unique identifier (TID) which is associated with it in the object processor. The preprocessor

constructs a trapezoid property table (TPT) which contains an entry associated with each TID. TPT entries contain trapezoid geometrical and color information and are used by the pixel tiling engine to compute pixel coverage and coloring.

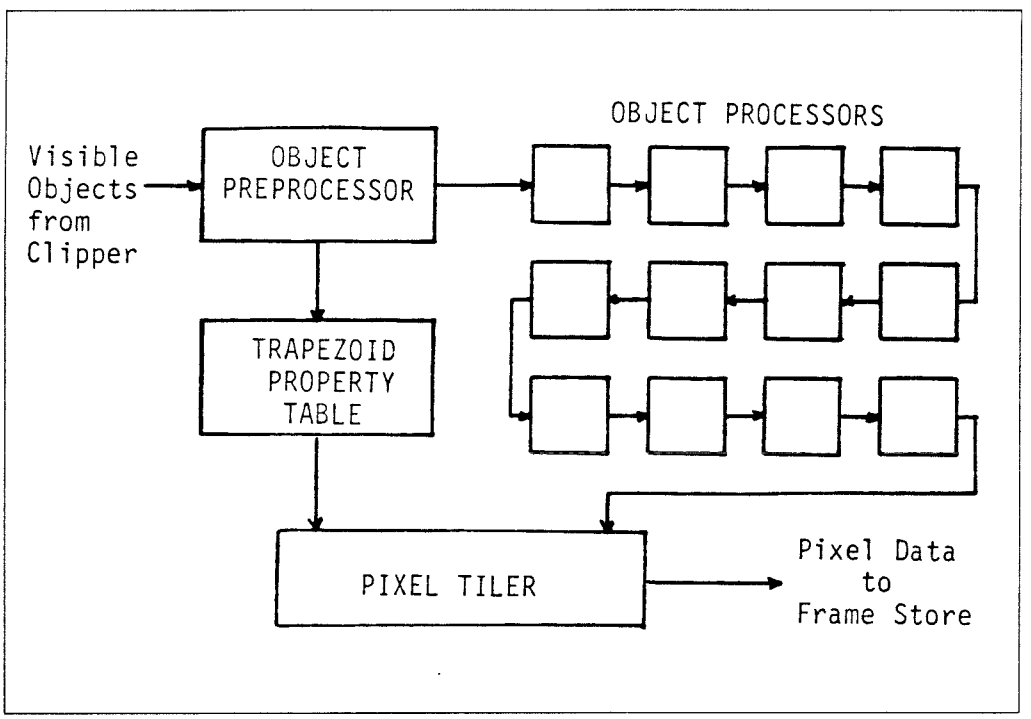


Figure 6.6: Weinberg's processor per object architecture.

Each object processor is initialized with a concise geometrical description of a trapezoid and its TID. During scan conversion, each processor decides whether its trapezoid intersects with a region centered about the current pixel. If it does, the processor attempts to insert its trapezoid's TID into a list which is passed from processor to processor in the pipeline. The entire object processor pipeline produces a stream of TIDs of objects that intersect pixel regions. This stream is sorted in a yxz ordering so that a sublist of TIDs can be associated with each pixel. This sublist is sorted by depth, the closest objects arriving first.

The pixel tiling unit uses a pixel's list of TIDs to tile a region surrounding the pixel on a fine grid. Each TID is used to recompute the intersection of the trapezoid with the pixel region and that intersection is tiled in a subpixel

buffer. A trapezoid's subpixels are convolved with a spatial filter kernel and the trapezoid's color contribution is added into the pixel color. After all of a pixel's trapezoids have been processed, the pixel has a correct color value associated with it and the pixel can be displayed.

The performance of Weinberg's architecture depends primarily upon the rate at which object processors can make their decisions and pass information along to the next processor. The preprocessor and tiling unit must keep up with the object processors but since only one object preprocessor and one pixel tiler are needed, Weinberg can afford to use a fair amount of fast logic in these units.

Weinberg determines the number of clock cycles, c, that his architecture will require to produce a visible surface image as:

$$c = f * (x * y * d_c + p * s)$$

where:

x = screen x dimension in pixels
y = screen y dimension in pixels
d_c = average number of objects at each pixel
s = cycles to load each processor
p = number of processors

Weinberg suggests that an average of three visible surfaces are emitted from the processor pipeline for each pixel. His design requires that nine values be loaded into each processor. Implementing a processor per object architecture for 6,000 trapezoids, and a 512 by 512 pixel screen requires:

$$c = 30 * (512 * 512 * 3 + 6000 * 9) = 25,212,960 \text{ cycles}$$

This requires the processors to be designed with a clock cycle of ≈ 40 nsec. Weinberg argues that this is possible with an ECL gate array implementation.

The spatial subdivision technique employed in the ANIMAC architecture allows the object processor's clock period to be increased significantly. Each visibility processor in the ANIMAC-1 system is responsible for the tiling of sixteen virtual processors. The total screen region associated with each visibility processor, N_{pix}, is the total screen area divided by the number of processors:

$$N_{pix} = \frac{512 * 512}{N_p} = \frac{262,144}{16} = 16,384 \text{ pixels}$$

Substituting N_{pix} for the $x * y$ term in Weinberg's equation, we find:

$$c = 30 * (16384 * 3 + 6000 * 9) = 3,094,560 \text{ cycles}$$

which allows the processors to be implemented in a much slower technology since the clock period has been increased to $\approx$ 325 nsec.

The number 6,000 has been used in these examples because Weinberg used it to illustrate the performance of his architecture in his dissertation. The ANIMAC-1 system must handle more than 6,000 trapezoids. Earlier we decided that each visibility processor must be designed to handle 6,250 polygons. These 6,250 polygons will be fractured into a number of trapezoids that depends upon the number of vertices in the visible polygons. All 6,250 polygons were initially triangles before being clipped. Based upon the observations of Chapters 2 and 3, we assume that most triangles will not cross a virtual processor boundary and will arrive in the visibility processor as triangles. Each triangle will be fractured into at most two trapezoids. This suggests that each processor be designed to handle 13,000 trapezoids. To be conservative, the ANIMAC-1 visibility processor has been designed to handle 20,000 trapezoids. This requires:

$$c = 30 * (16384 * 3 + 20000 * 9) = 6,874,560 \text{ cycles}$$

Thus a processor per object architecture can be realized with a processor that operates with a $\approx$ 145 nsec. clock cycle. The clock period can be increased more by realizing that the majority of the clock cycles are being used to load trapezoids into the processors, i.e. the $p * s$ term.

We can decrease the loading time by breaking the processor pipeline into N_c independent chains as Weinberg has suggested. Each chain is loaded in parallel and the chains are reconfigured as a pipeline during image tiling. The total number of clock cycles needed to compute an image can be rewritten as:

$$c = f * (N_{pix} * d_c + s * p/N_c)$$

If we implement the visibility processor so that it loads ten chains of 2,000 object processors in parallel, the image can be computed in:

$$c = 30 * (16384 * 3 + 9 * 20,000/10) = 2,014,560 \text{ cycles}$$

This increases the clock period to $\approx$ 495 nsec. The object processor can now be implemented in technologies which are slower than ECL gate arrays. I expect that the technology of choice will be bulk CMOS. A CMOS implementation will offer two major advantages. First, heat dissipation is greatly

reduced over an ECL gate array implementation, and second, many more object processors can be realized on a die.

Weinberg estimated that his processors require about 1,500 gate equivalents. Today's best gate array technology implements about 10,000 gates on a die. About six objects processors could be fabricated on a gate array today. We would expect that an object processor would occupy less silicon area when implemented in bulk CMOS. Custom MOS designs require less space than gate array implementations because they route signals more efficiently and implement logic and state more efficiently. Today's bulk CMOS probably would be able to implement ten processors per die.

Moore [MOORE79] suggests that the number of devices per die will double every two years. Thus, it should be possible to fabricate forty processors on a die by the early 1990's. This level of integration would allow a 2,000 processor chain to be implemented on a single printed circuit board with 50 integrated circuits. The entire 20,000 processor chain would only require 500 integrated circuits and could easily be implemented with only ten printed circuit boards.

In his dissertation, Weinberg describes how to implement a pixel tiler in ECL logic that can tile and filter a 64 by 64 subpixel tile in 23 nsec. His design makes extensive use of parallelism, pipelining and table lookup methods. With 495 nsec. to tile each pixel, implementing the pixel tiler will not push technological limits.

Figure 6.7 illustrates the overall visibility processor architecture. Visible polygons arrive from the clipping subsystem. The polygons need not be sorted by virtual processor. The object preprocessor fractures polygons into trapezoids and loads the trapezoids into the object processors. The object preprocessor also constructs the trapezoid property table (TBT). The pixel tiler generates pixels in yx order and transmits these pixel values to the frame store where they are stored for display. The frame store is double buffered so that one frame is being displayed while the other is being constructed.

With early 1990's technology, it would seem reasonable to expect that the visibility processor could be implemented with less than twenty printed circuit boards. I expect the object processors to occupy ten printed circuit boards and expect that the object preprocessor and pixel tilers could be implemented on just a few additional printed circuit cards. If we assume that a visibility processor can be implemented with sixteen printed circuit boards, then the entire ANIMAC-1 visibility subsystem could be constructed with 256 printed circuit cards which would most likely be housed in two racks.

Although the visibility subsystem is only a part of the ANIMAC-1 system, it will require considerably less space than the equivalent portion of today's real-time simulation engines and will offer performance well above that offered by today's systems.

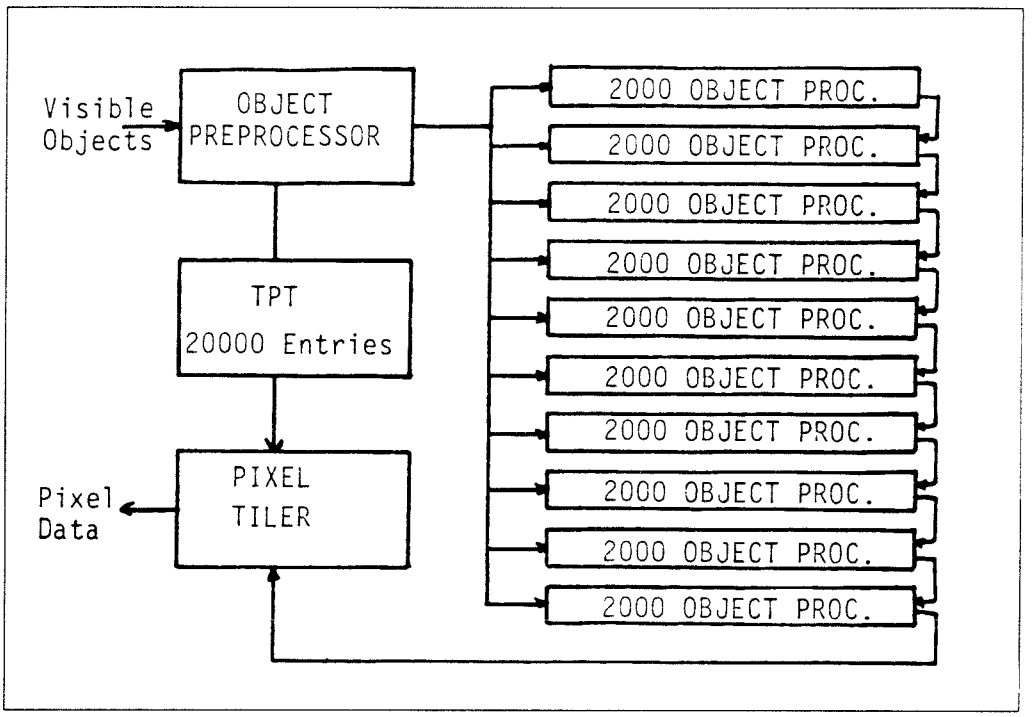


Figure 6.7: The ANIMAC-1 visibility processor architecture.

6.2 Implementing the ANIMAC-2 System

The ANIMAC-2 system extends the ANIMAC-1 system to produce shadowing effects in real-time. The addition of shadowing effects substantially increases the amount of hardware needed to implement the visibility processor. This should not be surprising since the software simulation spent most of its time performing shadowing computations.

Chapter 4 illustrated that shadowing requires interprocessor communications. That chapter developed the ANIMAC shadowing algorithm which divided image generation into three separate tasks. *Visible surface determination* determines which surfaces are visible, *local shadowing* determines whether pixels lie in shadows cast by objects within a visibility processor, and *foreign shadowing* determines whether pixels lie in shadows cast by objects in other visibility processors.

Foreign shadowing is implemented by the visibility processors with the shadow map algorithm which requires only a nearest neighbor processor interconnection scheme. Figure 6.8 illustrates the interprocessor communication links required in the ANIMAC-2 visibility subsystem architecture. The lines

between processors represent interprocessor communications channels. Each processor communicates with its eight nearest neighbors using horizontal, vertical and diagonal channels. Processors on the array boundaries must communicate with processors on the opposite side of the array; the figure illustrates that the channels form four tori, one each in the horizontal and vertical directions, and two in the diagonal directions.

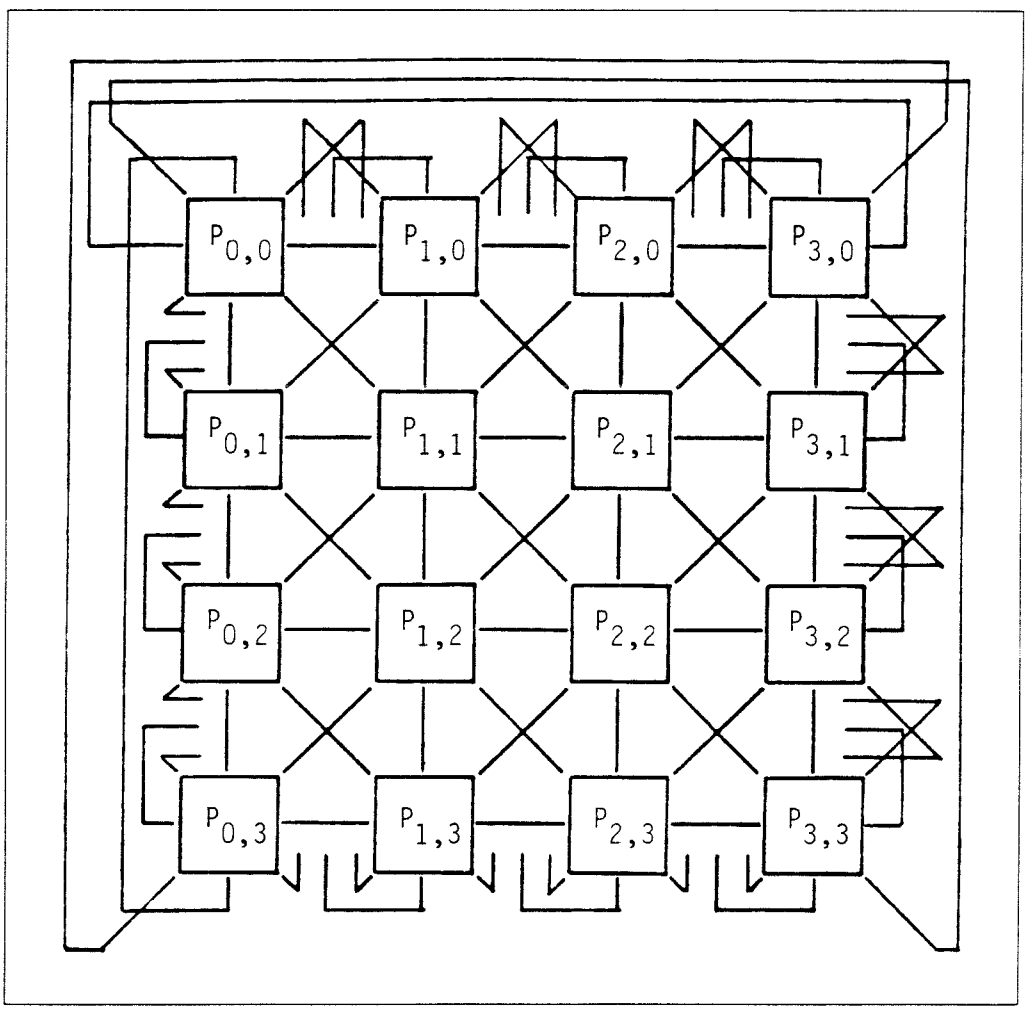


Figure 6.8: The ANIMAC-2 visibility subsystem requires interprocessor communications. Lines represent channels between a processor and its eight neighbors.

The ANIMAC-1 visibility subsystem was capable of producing a visible surface image of a scene within one frame time. The ANIMAC-2 visibility subsystem is unable to produce a shadowed scene within one frame time. Instead, the ANIMAC-2 system requires three frame times to compute a shadowed image. The ANIMAC-2 system can be built to update the image at slower than real-time (10 Hz) or may three-way interleave its visibility subsystem to achieve real-time performance. The majority of this chapter's discussion will focus upon implementing a visibility subsystem that can compute a shadowed frame in three frame periods since the problem of three-way interleaving such a system poses no new technological problems.

Generating shadowed images requires: (1) computing a visible surface image, (2) computing local shadowing effects, and (3) computing foreign shadowing effects. Table 6.1 illustrates the temporal sequence in which activities must be scheduled to generate frame i. The table also illustrates how three-way interleaving may be used to compute subsequent frames $i+1$ and $i+2$. Activities are subscripted to indicate the frame to which they contribute.

τ_1	τ_2	τ_3	τ_4	τ_5
LSM_i	CSM_i	Shad_i		
Vis_i		Shad_i		
	Local_i	Shad_i		
	LSM_{i+1}	CSM_{i+1}	Shad_{i+1}	
	Vis_{i+1}		Shad_{i+1}	
		Local_{i+1}	Shad_{i+1}	
		LSM_{i+2}	CSM_{i+2}	Shad_{i+2}
		Vis_{i+2}		Shad_{i+2}
			Local_{i+2}	Shad_{i+2}

Table 6.1: Task scheduling for the ANIMAC-2 Visibility Subsystem.

Table 6.1 illustrates that the foreign shadowing algorithm dictates system timing. Foreign shadowing must be broken down into three distinct computational phases. First, local shadow maps (LSMs) must be computed. After LSMs have been computed, composite shadow maps (CSMs) are created by merging data from LSMs. Finally, CSMs are sampled to determine whether a visible surface lies in a shadow cast by a foreign object. The ANIMAC-2 allocates a frame time to each of these three tasks which are denoted as LSM_i, CSM_i and Shad_i.

Visible surfaces require only one frame time to compute and are denoted in the table as Vis_i. Visible surfaces are computed at the same time as LSMs

because they must be computed prior to local shadowing and shadow map sampling.

The ANIMAC-2 implements local shadowing with a shadow volume algorithm. This algorithm requires the visible surface image to have already been computed. Therefore, the local shadowing task, Local_i, must follow Vis_i.

Table 6.1 shows that no two instances of a task are ever scheduled concurrently. This suggests that three-way interleaving may be implemented without replicating processing units if appropriate buffering is implemented. Interprocessor communication occurs only during the CSM_i task which suggests that the interprocessor communication channels need not be replicated for three-way interleaving.

Figure 6.9 presents a detailed functional diagram of the ANIMAC-2 visibility processor architecture. The processor is divided into five major subsystems. The visible surface processor (VSP) uses a processor per object architecture similar to the one employed by the ANIMAC-1. The local shadowing processor (LSP) uses a new processor per object architecture which will be discussed later. The local shadow map processor (LSMP) uses a processor per pixel architecture to create each LSM. The composite shadow map processor (CSMP) uses a simple merging processor to create each CSM. The illumination processor (IP) determines the color of a subpixel taking into account whether the subpixel needs to be shadowed.

Figure 6.9 illustrates the visibility processor receiving polygons from the clipping subsystem over three links. The clipping subsystem clips visible polygons against each virtual processor's clipping space. These visible polygons are received by the VSP and LSMP over different links. The clipping subsystem also clips shadow volume polygons against each virtual processor's clipping space. Shadow volume polygons are received by the LSP.

Visibility processors communicate with each other over their interprocessor communication channels. Each virtual processor communicates over eight virtual channels. Figure 6.9 shows these virtual channels being multiplexed onto eight physical channels by the channel multiplexor.

The visibility processor also contains several memories which are shared between subsystems. The subpixel buffer (SPBUF) is used to store information about visible surfaces on a subpixel basis. Information is stored in the SPBUF by the VSP and accessed by the LSP and the IP.

The local shadow buffer (LSBUF) contains a bit for each subpixel. This entry indicates whether the subpixel lies in a locally cast shadow. The LSBUF is written to by the LSP and read from by the IP.

The composite shadow maps (CSM) are illustrated as part of the CSMP. Each CSM is implemented as twelve QSMs. CSMs are written by the CSMP and inspected by the IP.

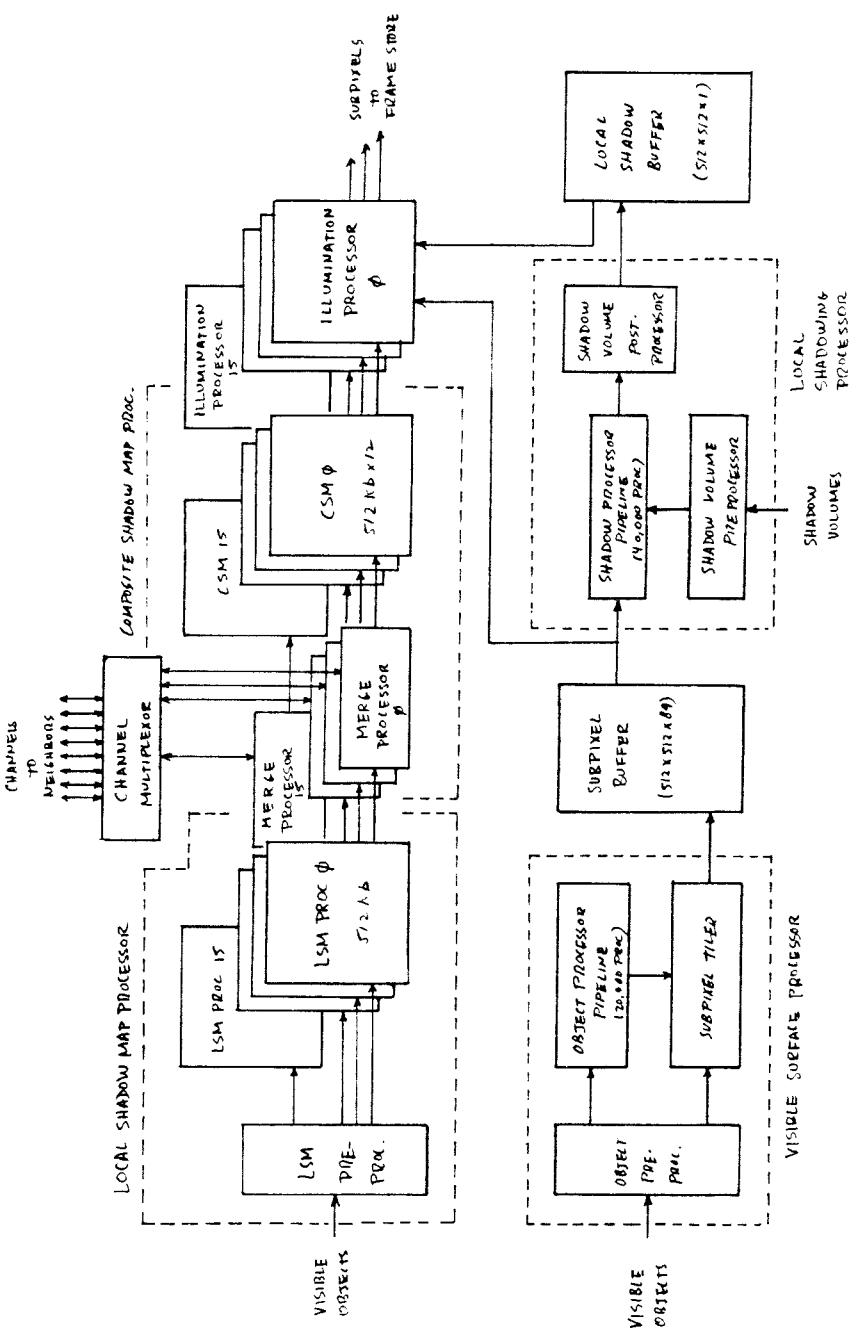


Figure 6.9: A detailed overview of the ANIMAC-2 visibility processor architecture.

The following sections discuss the implementation of the visibility processor subsystems.

6.2.2 Visible Surface Processor

The ANIMAC-2 visible surface processor uses a processor per object architecture similar to the one used in the ANIMAC-1 system. The object preprocessor and the object processors are identical. The pixel tiler has been changed. Instead of computing the final color for a pixel, it now tiles subpixels and stores these subpixel values in a subpixel buffer (SPBUF).

The processor per object architecture has previously been discussed in some detail. This discussion will focus on the design of the pixel tiler and subpixel buffer.

6.2.2.1 Pixel Tiling Processors

The pixel tiler receives a trapezoid identifier (TID) from the object processor pipeline every clock cycle (495 nsec.). Several TIDs may arrive for each pixel that has to be tiled. The previous discussion assumed that the pixel tiler must on the average consider three TIDs per pixel. Each tiled pixel requires the tiling of some number, s, of subpixels. Since the pixel tiler may have to tile a pixel for each TID received, it must be capable of generating s subpixels every clock period. The ANIMAC-2 architecture tiles each pixel on a four by four subpixel grid, thus sixteen subpixels must be computed every 495 nsec.

The ANIMAC-2 pixel tiler has been designed to tile subpixels differently than Weinberg's original pixel tiler. Weinberg's pixel tiler tiled a two by two pixel region around the pixel center. He assumed that a visible surface's depth and color were constant over this region. The ANIMAC-2 pixel tiler differs in that it need only compute the subpixels within a one by one pixel region around the pixel center. So that subpixels may be shadowed correctly, the ANIMAC-2 pixel tiler does not assume constant depth and color over the pixel region.

The ANIMAC-2 pixel tiler is implemented as a pipelined system. Figure 6.10 illustrates that the pixel tiler consists of a trapezoid intersection processor (TIP) and four row processors (RP). The trapezoid intersection processor receives a TID from the object processor pipeline and clips the TID's trapezoid against the current pixel's y extents. This produces another trapezoid which is passed onto the first row processor.

Each row processor is responsible for the tiling of four subpixels. RPs compute, for each subpixel, trapezoid depth and surface normal vectors. RPs operate by receiving trapezoids from their left neighbor and tiling their subpixels. Each row processor strips its scan-line from the trapezoid and passes

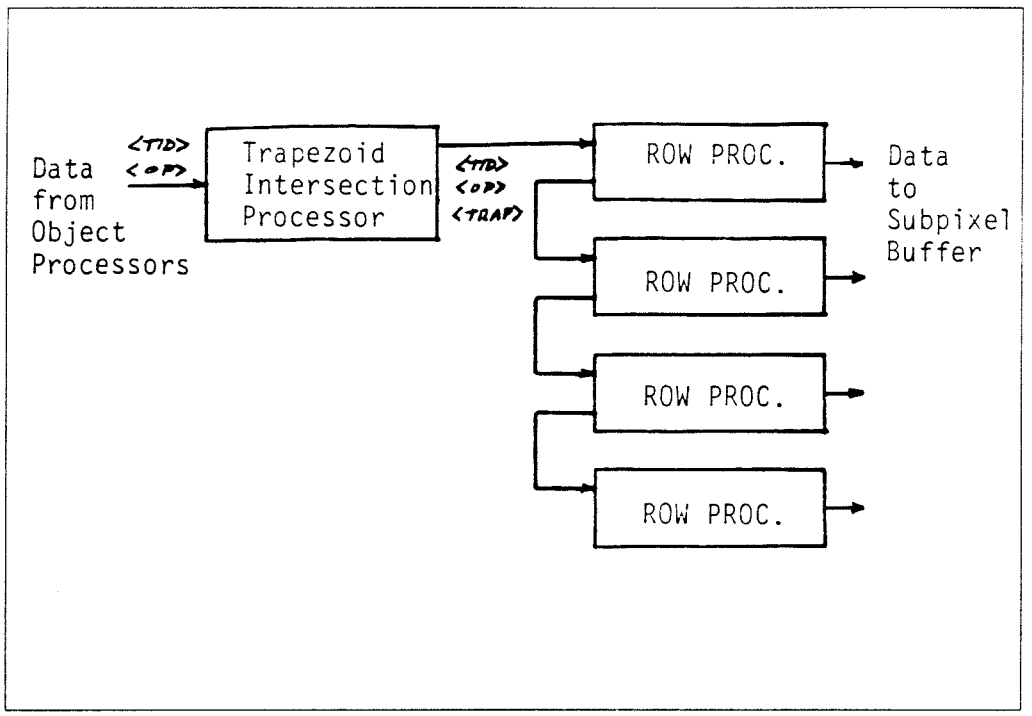


Figure 6.10: The ANIMAC-2 pixel tiler architecture.

this new trapezoid to the RP to its right. Each RP can store its four subpixels in the SPBUF memory.

The TIP operates by receiving a TID and a opcode from the object processor pipeline. The opcode indicates what type of operation is occurring. During scan-conversion, the opcodes indicate that the TID should be associated with a new pixel or with the current pixel. Processors keep track of the current pixel by updating a pixel counter when new pixel opcodes arrive.

To determine the y extents of the TID's trapezoid, the TIP first retrieves information describing the trapezoid from the trapezoid property table (TPT). A trapezoid's boundaries are described to subpixel resolution with a fixed point representation. Surface normals vectors are also computed using a fixed point representation. The following values are stored in the TPT and describe a

trapezoid.

y_t	Top scan-line y value
y_b	Bottom scan-line y value
x_l	Left-most x value on y_t
x_r	Right-most x value on y_t
δx_l	Left edge $\partial x/\partial y$
δx_r	Right edge $\partial x/\partial y$
z	Depth at (x_l, y_t)
δz_p	Change in depth per pixel, i.e. $\partial z/\partial x$
δz_l	Change in depth per scan-line
N	Normal vector at (x_l, y_t)
δN_l	Change in N per scan-line
δN_p	Change in N per pixel, i.e. $\partial N/\partial x$
$\delta\delta N_p$	Change in N_p per scan-line

A rectangular region surrounding the current pixel can be described with its x and y edge coordinates as:

p_t	Top subpixel scan-line y value
p_b	Bottom subpixel scan-line y value
p_l	Left subpixel edge x value
p_r	Right subpixel edge x value

The intersection of a trapezoid and a rectangular pixel region can be computed by intersecting the trapezoid with the pixels y extents. Since the trapezoid is visible at the current pixel, we know that $y_t \geq p_b$. If $y_t < p_t$ we

compute another trapezoid which is clipped to y_t as:

$$
\begin{aligned}
y_t' &:= \min(y_t, p_t); \\
y_b' &:= y_b; \\
x_l' &:= x_l + \delta x_l(y_t - p_t); \\
x_r' &:= x_r + \delta x_r(y_t - p_t); \\
\delta x_l' &:= \delta x_l; \\
\delta x_r' &:= \delta x_r; \\
z' &:= z + \delta z_l(y_t - p_t); \\
\delta z_p' &:= \delta z_p; \\
\delta z_l' &:= \delta z_l; \\
N &:= N + \delta N_l(y_t - p_t); \\
\delta N_p' &:= \delta N_p + \delta\delta N_p(y_t - p_t); \\
\delta N_l' &:= \delta N_l; \\
\delta\delta N_p' &:= \delta\delta N_p;
\end{aligned}
$$

Only the boundary coordinates, depth, and normal vector are modified during this computation. These variables can be computed more efficiently by regrouping as:

$$
\begin{aligned}
y_t' &:= \min(y_t, p_t); \\
\Delta y &:= y_t - p_t; \\
x_l' &:= x_l + \delta x_l \Delta y; \\
x_r' &:= x_r + \delta x_r \Delta y; \\
z' &:= z + \delta z_l \Delta y; \\
N_1' &:= N_1 + \delta N_{l_1} \Delta y; \\
N_2' &:= N_2 + \delta N_{l_2} \Delta y; \\
N_3' &:= N_3 + \delta N_{l_3} \Delta y; \\
\delta N_{p_1}' &:= N_{p_1} + \delta\delta N_{p_1} \Delta y; \\
\delta N_{p_2}' &:= N_{p_2} + \delta\delta N_{p_2} \Delta y; \\
\delta N_{p_3}' &:= N_{p_3} + \delta\delta N_{p_3} \Delta y;
\end{aligned}
$$

This computation requires nine multiplications, ten additions, and one comparison to be performed every 495 nsec. The longest computation path

requires the sequential computation of two additions and one multiplication. Figure 6.11 illustrates how this computation can be implemented in hardware.

The different computations have to be computed to different precisions. We know that any trapezoid is constrained to lie in the portion of the subpixel space associated with its virtual processor. For the ANIMAC systems, a virtual processor occupies a 32 by 32 pixel region. Each pixel is tiled on a 4 by 4 subpixel grid. Thus each virtual processor region occupies a 128 by 128 subpixel region which implies that the difference Δy can be represented as a signed 8 bit integer.

The trapezoid is clipped against y_t only if $\Delta y > 0$. This implies that each of the multiplies requires multiplying an n bit signed integer by a 7 bit unsigned integer. Both δx_l and δy_l can be represented as 15 bit signed integers. If z is represented as a 24 bit unsigned integer, then δz_l needs to be represented as a 32 bit signed integer. Normal vectors can be represented with 16 bit signed integers which suggests that δN_l and δN_p be represented with 23 bit signed integers. $\delta\delta N_p$ requires additional precision and should be represented with a 30 bit signed integer.

Thus the longest computation path involves the computation of z' which requires an 8 bit subtraction, a 32 by 8 bit multiplication and a 24 bit addition. The multiplication dominates the computation time. A 32 by 8 bit multiplication can be implemented with two 16 by 8 bit multiplications and one 24 bit addition. Commercially available 16 by 16 multipliers can produce a 32 bit product in 75 nsec. The computation of $z + \delta z_l \Delta y$ can easily be computed in 225 nsec. which is much faster than the required computation time of 495 nsec. This extra 270 nsec. can be used to fetch the trapezoid values from the TPT. The TPT can be implemented with MOS dynamic RAMs.

Figure 6.12 illustrates the structure of the row processors. The row processor consists of five subcomponents. The trapezoid computation processor (TCP) strips the row processor's subscan-line from the trapezoid and passes the new trapezoid onto the next row processor. The x range processor (XRP) computes the intersection of the trapezoid with the row processor's subscan-line and passes a description of this intersection onto the subpixel processors (SPP). Each SPP computes the trapezoid's depth and surface normal at the subpixel and retains these values if the depth is less than the subpixel's current depth. SPPs are also responsible for storing the SPP values in the subpixel buffer memory (SPBUF).

When the TCP receives a trapezoid, it strips its scan-line from the trapezoid and passes the remaining trapezoid and opcode onto the next row processor. The TCP determines whether its scan-line, y, intersects the trapezoid by checking if $y_b \leq y \leq y_t$. If $y > y_t$, the row processor passes the trapezoid

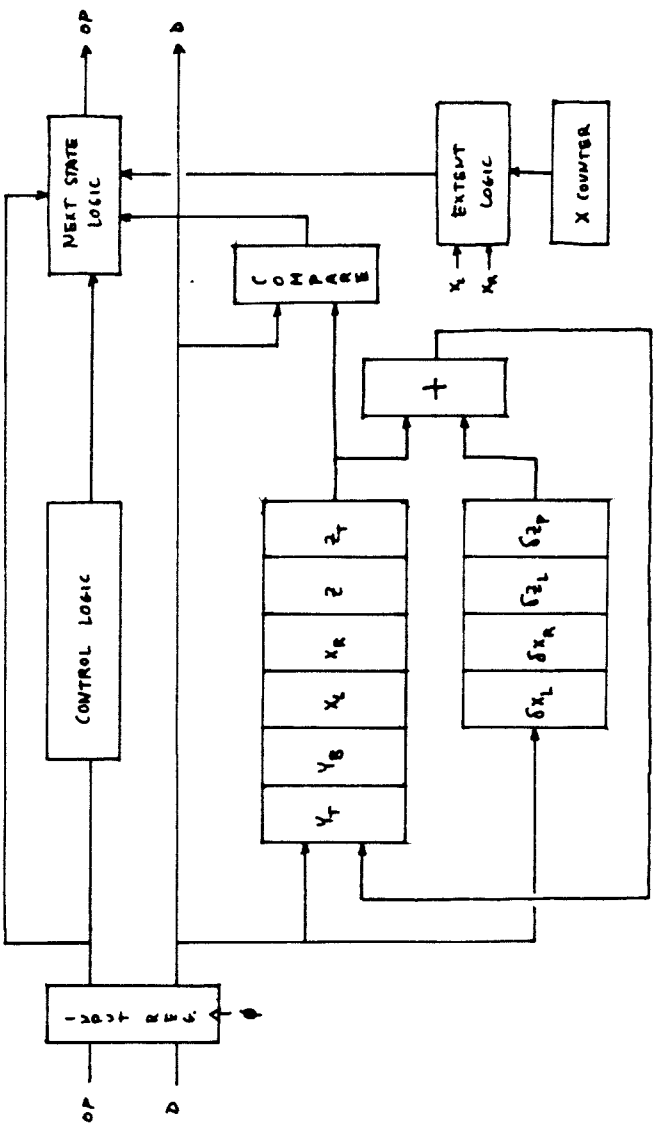


Figure 6.11: One possible implementation of the trapezoid intersection computation.

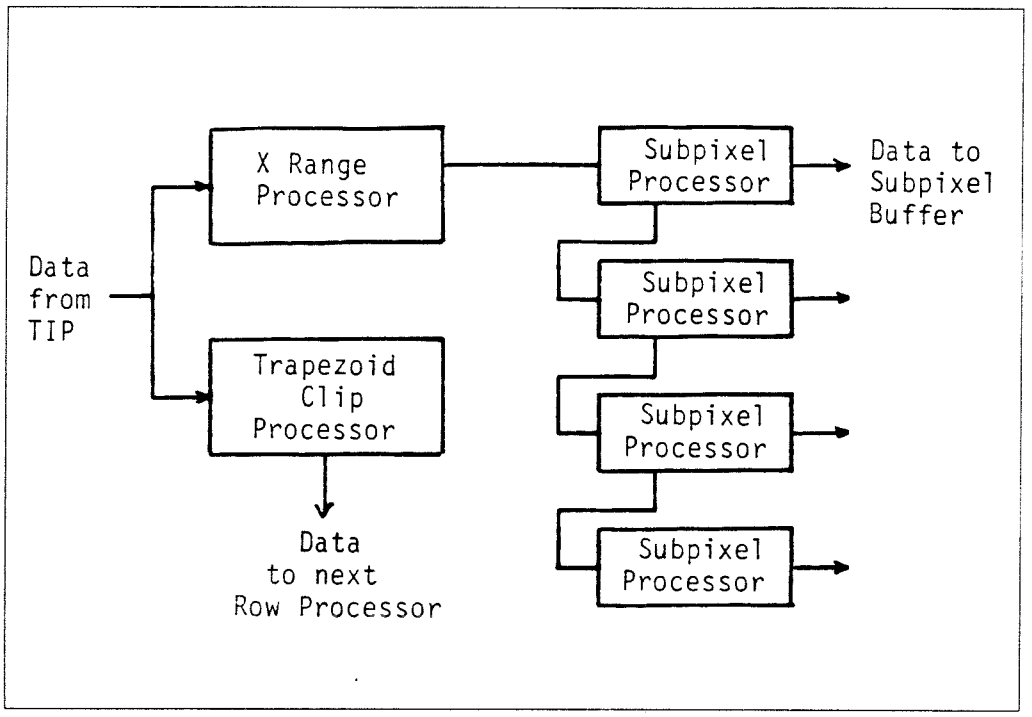


Figure 6.12: The Row Processor can be implemented in a pipelined fashion with six specialized processors.

to the next processor unmodified, otherwise it removes a scan-line from the trapezoid by modifying:

$$
\begin{aligned}
y_t' &:= y_t - 1; \\
x_l' &:= x_l + \delta x_l; \\
x_r' &:= x_r + \delta x_r; \\
z' &:= z + \delta z_l; \\
N_1' &:= N_1 + \delta N_{l_1}; \\
N_2' &:= N_2 + \delta N_{l_2}; \\
N_3' &:= N_3 + \delta N_{l_3}; \\
\delta N_{p_1}' &:= \delta N_{p_1} + \delta\delta N_{p_1}; \\
\delta N_{p_2}' &:= \delta N_{p_2} + \delta\delta N_{p_2}; \\
\delta N_{p_3}' &:= \delta N_{p_3} + \delta\delta N_{p_3};
\end{aligned}
$$

This computation only requires ten additions and can easily be implemented in well under 495 nsec.

The XRP determines the x extents of the trapezoid on the row processor's subscan-line. It computes three values which are passed along to the subpixel processors. These values are used to indicate whether the current scan-line is within the trapezoid and which subpixel processors are within the trapezoid. These values are:

s_y	Subpixel scan-line intersects trapezoid (1-bit)
s_{x_l}	Left-most active subpixel (2-bits)
s_{x_r}	Right-most active subpixel (2-bits)

These values are computed as:

$$\begin{aligned} s_y &:= (y = y_t \wedge y \geq y_b); \\ s_{x_l} &:= \max(x_l, p_l); \\ s_{x_r} &:= \min(x_r, p_r); \end{aligned}$$

These three values along with the trapezoids TID, z, δz_p, N, and δN_p are passed to the subpixel processors. The pixel depth and surface normal must be correctly computed for the first pixel if $x_l < p_l$ as:

$$\begin{aligned} z' &:= z + \delta z_p(x_l - p_l); \\ N_1' &:= N_1 + \delta N_{p_1}(x_l - p_l); \\ N_2' &:= N_2 + \delta N_{p_1}(x_l - p_l); \\ N_3' &:= N_3 + \delta N_{p_1}(x_l - p_l); \end{aligned}$$

These computations require five additions, four multiplications and four comparisons. These operations can easily be computed in the required 495 nsec. with today's technology.

Each subpixel processor receives these values from the processor to its left. The SPP compares the subpixel depth with its subpixel depth and updates its visible trapezoid if the new trapezoid is closer.

The SPP maintains information about the surface which is currently visible at its subpixel. This information consists of the subpixel depth, trapezoid TID, and surface normal vector. The SPP potentially updates its visible surface every clock period. It receives information from the preceding processor and first checks whether its subpixel lies in the interval $[s_{x_l}, s_{x_r}]$. If it is in this interval, the SPP attempts to update its visible surface and modifies the values of z and N that it passes onto the next SPP.

The SPP updates its visible surface by comparing the arriving subpixel depth with the stored subpixel depth. If the arriving depth is smaller, the SPP stores the arriving depth, TID, and surface normal vector.

The SPP also must update the values of z and N. These values are computed as:

$$z' := z + \delta z_p;$$
$$N_1' := N_1 + \delta N_{p_1};$$
$$N_2' := N_2 + \delta N_{p_2};$$
$$N_3' := N_3 + \delta N_{p_3};$$

These four additions plus the conditional update of the visible surface can be performed in much less than the 495 nsec. The remaining time can be used to store the subpixel information in the SPBUF when a new pixel is to be tiled.

The subpixel buffer (SPBUF) stores visible surface information at the subpixel resolution. These subpixels are later checked to see if they lie in shadow. Finally, subpixel colors are computed by the illumination processor (IP) and stored in the frame store for viewing.

Each SPBUF entry contains entries for subpixel depth (24 bits), surface identifier (17 bits), and surface normal vector (48 bits) requiring a total of 89 bits of storage per subpixel. The SPBUF must be capable of storing sixteen subpixel entries every 495 nsec. The SPBUF can be implemented with 200 nsec. memory if it is organized so that eight subpixels can be written simultaneously. This would require a data buss that is 712 bits wide. Narrower data busses could be implemented by utilizing faster memories.

6.2.3 Local Shadowing Processor

The ANIMAC-2 system implements a local shadowing algorithm fashioned after the shadow volume algorithm [CROW77A]. This implementation makes use of a new processor per object architecture developed by the author. This architecture resembles Weinberg's visible surface architecture [WEINBE82] but instead of determined which surface is visible at a point, this architecture determines whether points on visible surfaces lie in shadow. In the ANIMAC-2, these points are the subpixels computed by the visible surface processor.

Figure 6.9 illustrates the local shadowing processor per object architecture. The clipping subsystem creates shadow volumes and clips the shadow volume's faces (shadow polygons) against each virtual processor's clipping volume. The clipping subsystem delivers shadow polygons to the shadow volume preprocessor (SVP) ordered so that a shadow volume's shadow polygons arrive as a group.

The SVP fractures the shadow polygon into trapezoids and loads these trapezoids into the shadow polygon processors so that all of a shadow volume's trapezoids are loaded into adjacent processors. These adjacent processors are collectively referred to as a *shadow volume region.*

After a shadow polygon processor has been loaded, its state consists of geometrical information that describes the trapezoid's boundaries and depth, and two additional bits of information which are stored as processor flags. The F flag bit indicates whether the trapezoid faces towards or away from the viewer. The P flag bit indicates that the processor is the last processor in a shadow volume region.

The local shadowing processor pipeline operates by passing opcodes and data down the pipeline. An opcode and datum are presented to the first shadow polygon processor by the shadow volume preprocessor and are passed from processor to processor. Under certain conditions, a processor may alter the opcode but it can never alter the datum.

The opcodes consist of instructions to initialize the processors, to update scan-line variables, and to shadow a subpixel. Subpixels are shadowed by propagating the subpixel's depth down the processor pipeline. Information about the subpixel's shadowing state is encoded within the opcode using four shadowing states. The shadowing states are:

$\langle U \rangle$	Shadowing is undetermined.
$\langle B \rangle$	The subpixel is behind a frontfacing trapezoid.
$\langle I \rangle$	The subpixel is in front of a backfacing trapezoid.
$\langle S \rangle$	The subpixel is in shadow.

As the subpixel depth propagates down the pipeline, each processor incrementally computes the depth of its trapezoid at the current subpixel and compares its depth with the subpixel depth. The subpixel's shadowing state is then modified depending upon the results of the depth comparison and the processor's state. When the subpixel is shifted out of the pipeline, the shadow volume post processor observes the shadowing state and records the subpixel's shadowing state in the local shadow buffer (LSB).

This architecture implements the shadow volume algorithm through a sequence of simple shadowing state transitions. A subpixel enters a shadow volume region in one of two shadowing states. The shadowing state may be unknown $\langle U \rangle$ or in shadow $\langle S \rangle$. Since the subpixel must enter a shadow volume region in these two shadowing states, it also must exit the region in one of these two states. Shadowing states may make the transition from $\langle U \rangle$ to $\langle S \rangle$ within a shadow volume region.

The shadow volume algorithm implies that a subpixel lies in shadow if the subpixel lies in the interior of a shadow volume. If the scene environment is constructed from convex polygons, the test to decide if a point lies inside a shadow volume is simplified to checking whether the point lies behind a frontfacing shadow polygon and in front of a backfacing shadow polygon. Table 6.2 shows that simple shadowing state transitions can be used to implement this decision.

Pixel Shadowing State Transitions				
	F = Frontfacing		F = Backfacing	
Pixel State	$z_t < z_p$	$z_t \geq z_p$	$z_t \leq z_p$	$z_t > z_p$
$\langle U \rangle$	$\langle B \rangle$	$\langle U \rangle$	$\langle U \rangle$	$\langle I \rangle$
$\langle B \rangle$	$\langle B \rangle$	$\langle B \rangle$	$\langle B \rangle$	$\langle S \rangle$
$\langle I \rangle$	$\langle S \rangle$	$\langle I \rangle$	$\langle I \rangle$	$\langle I \rangle$
$\langle S \rangle$	$\langle S \rangle$	$\langle S \rangle$	$\langle S \rangle$	$\langle S \rangle$

Table 6.2: Pixel Shadowing State Transitions. z_p represents the subpixel depth. z_t represents the trapezoid's depth.

A subpixel's shadowing state must be restored to either $\langle U \rangle$ or $\langle S \rangle$ by the last shadow polygon processor in a shadow region, i.e. when $P = 1$. Table 6.3 illustrates a shadowing state transition that is applied by processors with $P = 1$ after the shadowing state transition of Table 6.2. These two shadowing state transitions can be combined into one state transition that depends upon the flag bits F and P and the results of a depth comparison.

Pixel Restoration State Transitions	
Pixel State	Next State
$\langle U \rangle$	$\langle U \rangle$
$\langle B \rangle$	$\langle U \rangle$
$\langle I \rangle$	$\langle U \rangle$
$\langle S \rangle$	$\langle S \rangle$

Table 6.3: Pixel states are restored to either the $\langle U \rangle$ or $\langle S \rangle$ state by the last processor in a shadow volume region.

The idea of using a state transition function to implement the shadow volume algorithm was also used in the author's software implementation described in Chapter 5. The processor per object implementation is simpler than the software implementation because the scene model is constructed from convex polygons. A state transition technique exactly like the one used by the software implementation can be implemented in hardware to handle shadow

volumes cast by non-convex polygons. The software shadowing state transition requires that polygons be tiled carefully. The shadowing state transition proposed here allows the shadow polygon processors to tile shadow polygons without worrying about silhouette and interior edges.

Operation of the shadow polygon pipeline can be broken down into three phases. During the load phase, trapezoidal information is loaded into the shadow polygon processors. During the shadowing phase, processors incrementally determine whether their trapezoid is visible at the current subpixel and if so, perform the shadowing state transitions of Tables 6.2 and 6.3. During the scan-line update phase, processors update trapezoidal values that need to be updated for each new scan-line.

A description of the shadow polygon processor behavior during these three phases requires first describing how trapezoids are represented. Trapezoids are conveniently described with the nine values recommended by Weinberg. All values must be described to the resolution of the subpixel image. These values are:

y_t	Top scan-line y value
y_b	Bottom scan-line y value
x_l	Left-most x value on y_t
x_r	Right-most x value on y_t
δx_l	Left edge $\partial x/\partial y$
δx_r	Right edge $\partial x/\partial y$
z	Depth at (x_l, y_t)
δz_p	Change in depth per subpixel, ie. $\partial z/\partial x$
δz_l	Change in depth per scan-line

Figure 6.12 illustrates the shadow polygon processor architecture at the register level. This implementation is similar to the one proposed by Weinberg. An opcode and a datum arrive at the processor's input port. The *control logic* interprets the opcode to effect the instruction. Three classes of instructions are encoded in the opcode. The *Load* instructions are used to initialize the processor's state. The *Inc* instructions are used to perform scan-line updates of y_t, y_b, x_l, x_r, and z. The *Pix* instructions determine whether a subpixel should be shadowed. The four shadowing states are encoded in the four *Pix* instructions. Table 6.5 lists the opcodes and describes their functions.

Table 6.5 indicates that none of the instructions modify the datum d' and only the *Load* instructions and three of the *Pix* instructions modify the opcode op'. The *Load* instructions stores datum bits in the various registers. The five *Inc* instructions use the processor's adder to perform scan-line updates to values stored in the registers. The four *Pix* instructions possibly modify the

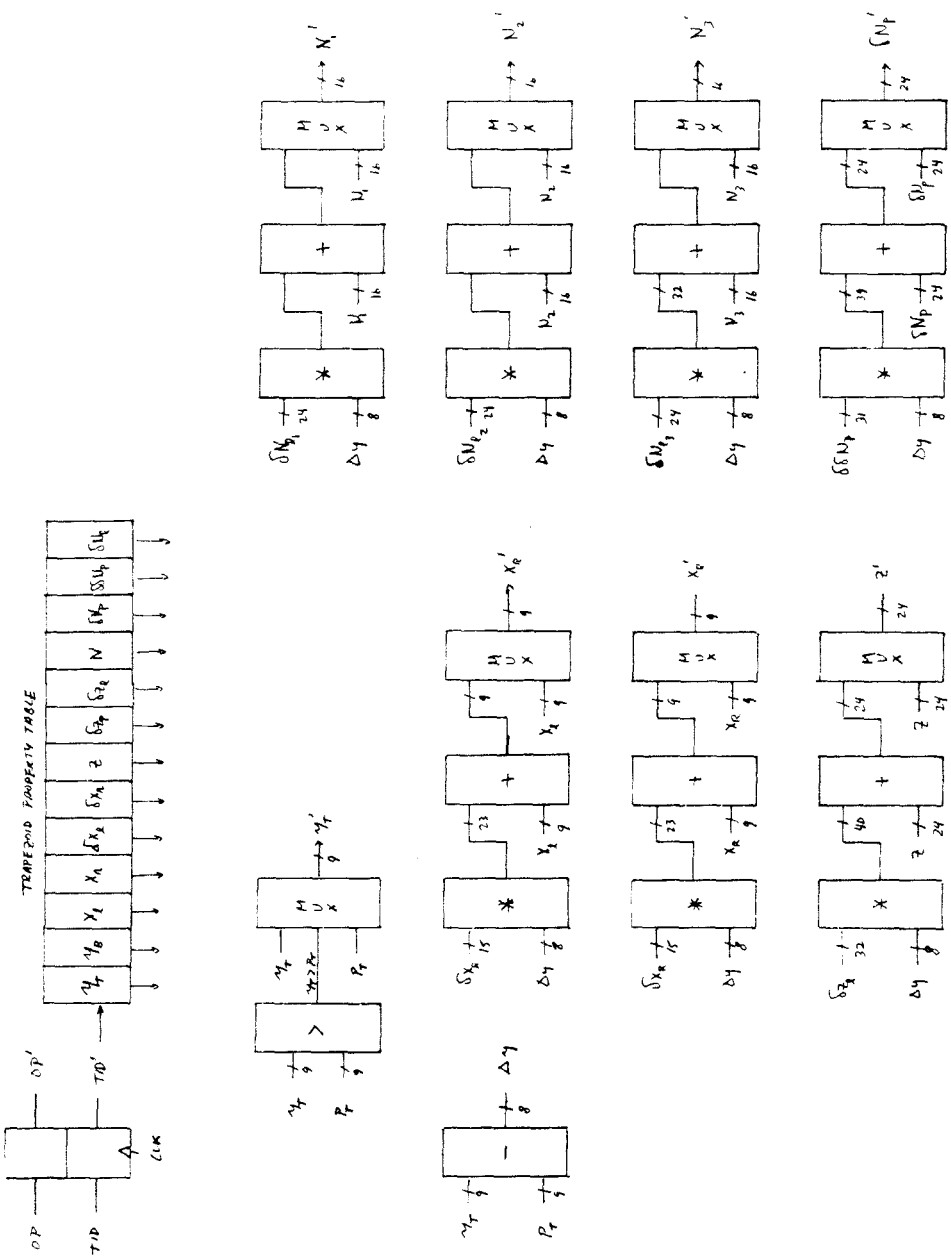


Figure 6.12: Shadow Polygon Processor architectural detail.

NOP	Do Nothing
$op' \leftarrow$ NOP	$d' \leftarrow d$

Reset	Reset Processors and Enable Loading
$I \leftarrow 1$	
$T \leftarrow 0$	$B \leftarrow 0$
$op' \leftarrow$ Reset	$d' \leftarrow d$

LoadY	Load Y and Flag Registers
if $(I = 1)$	
$y_t \leftarrow d\langle 0{:}8 \rangle$	$y_b \leftarrow d\langle 9{:}17 \rangle$
$F \leftarrow d\langle 18 \rangle$	$P \leftarrow d\langle 19 \rangle$
$op' \leftarrow$ NOP	$d' \leftarrow d$
else	
$op' \leftarrow$ LoadY	$d' \leftarrow d$

LoadX	Load X Registers
if $(I = 1)$	
$x_l \leftarrow d\langle 0{:}8 \rangle$	$x_r \leftarrow d\langle 9{:}17 \rangle$
$op' \leftarrow$ NOP	$d' \leftarrow d$
else	
$op' \leftarrow$ LoadX	$d' \leftarrow d$

LoadXL	Load Left Edge Register
if $(I = 1)$	
$\delta x_l \leftarrow d\langle 0{:}17 \rangle$	
$op' \leftarrow$ NOP	$d' \leftarrow d$
else	
$op' \leftarrow$ LoadXL	$d' \leftarrow d$

LoadXR	Load Right Edge Register
if $(I = 1)$	
$\delta x_r \leftarrow d\langle 0{:}17 \rangle$	
$op' \leftarrow$ NOP	$d' \leftarrow d$
else	
$op' \leftarrow$ LoadXR	$d' \leftarrow d$

Table 6.5: Shadow Polygon Processor opcodes and their functional descriptions.

LoadZP	Load δz_p Register
if $(I = 1)$	
$\delta z_p \leftarrow d\langle 0{:}31\rangle$	
$op' \leftarrow$ NOP	$d' \leftarrow d$
else	
$op' \leftarrow$ LoadZP	$d' \leftarrow d$

LoadZ	Load Z Register
if $(I = 1)$	
$z \leftarrow d\langle 0{:}23\rangle$	
$op' \leftarrow$ NOP	$d' \leftarrow d$
else	
$op' \leftarrow$ LoadZ	$d' \leftarrow d$

LoadZL	Load δz_l Register
if $(I = 1)$	
$\delta z_l \leftarrow d\langle 0{:}31\rangle$	
$op' \leftarrow$ NOP	$d' \leftarrow d$
else	
$op' \leftarrow$ LoadZL	$d' \leftarrow d$
$I \leftarrow 0$	

IncXL	Scan-line update x_l
$x \leftarrow 0$	
$x_l \leftarrow x_l + \delta x_l$	$L = 0$
$op' \leftarrow$ IncXL	$d' \leftarrow d$

IncXR	Scan-line update x_r
$x_r \leftarrow x_r + \delta x_r$	$R = 0$
$op' \leftarrow$ IncXR	$d' \leftarrow d$

IncZ	Scan-line update z
$z \leftarrow z_t \leftarrow z + \delta z_l$	
$op' \leftarrow$ IncZ	$d' \leftarrow d$

IncYT	Scan-line update y_t
$y_t \leftarrow y_t - 1$	$T \leftarrow (y_t = 0) \vee T$
$op' \leftarrow$ IncYT	$d' \leftarrow d$

Table 6.5 (cont.): Shadow Polygon Processor opcodes and their functional descriptions.

IncYB	Scan-line update y_b
$y_b \leftarrow y_b - 1$	$B \leftarrow (y_b = 0) \vee B$
$op' \leftarrow$ IncYB	$d' \leftarrow d$

PixU	Determine Subpixel Shadowing
$L \leftarrow (x_l = x) \vee L$	$R \leftarrow (x_r = x) \vee R$
if (L)	
$z_t \leftarrow z_t + \delta z_p$	
$x \leftarrow x + 1$	
$op' \leftarrow$ Pix?	$d' \leftarrow d$

PixI	Determine Subpixel Shadowing
$L \leftarrow (x_l = x) \vee L$	$R \leftarrow (x_r = x) \vee R$
if (L)	
$z_t \leftarrow z_t + \delta z_p$	
$x \leftarrow x + 1$	
$op' \leftarrow$ Pix?	$d' \leftarrow d$

PixB	Determine Subpixel Shadowing
$L \leftarrow (x_l = x) \vee L$	$R \leftarrow (x_r = x) \vee R$
if (L)	
$z_t \leftarrow z_t + \delta z_p$	
$x \leftarrow x + 1$	
$op' \leftarrow$ Pix?	$d' \leftarrow d$

PixS	Determine Subpixel Shadowing
$L \leftarrow (x_l = x) \vee L$	$R \leftarrow (x_r = x) \vee R$
if (L)	
$z_t \leftarrow z_t + \delta z_p$	
$x \leftarrow x + 1$	
$op' \leftarrow$ PixS	$d' \leftarrow d$

Table 6.5 (cont.): Shadow Polygon Processor opcodes and their functional descriptions.

opcode if the subpixel is inside the processor's trapezoid and update certain trapezoidal values.

The implementation of the *Load* and *Inc* instructions is straightforward and can be implemented with a small control logic PLA. The interesting computations take place during the *Pix* instructions. Pixel shadowing state transitions are implemented by the *Next State Logic* unit based upon inputs from the *Extent Logic*, *Control Logic*, *Depth Comparator*, and the trapezoid state flags.

Seven one-bit state flags are needed by the shadow polygon processor. Two of these seven flags are set by the *Load* instructions. These two flags are the F flag, which indicates whether the trapezoid is front or backfacing, and the P flag, which indicates whether the processor is at the end of a shadow volume region. Another flag, the I flag, is used during processor loading.

Two other flag bits are updated once per scan-line by the *Inc* instructions. These flags are the T flag, which indicates that the current scan-line is equal to or less than y_t and the B flag, which indicates that the current scan-line is equal to or less than y_b.

The remaining two flags are updated for each subpixel during the *Pix* instructions. These flags are the L flag, which indicates whether the current subpixel x coordinate is equal to or greater than x_l, and the R flag, which indicates that the current subpixel x coordinate is greater than x_r.

The *Extent Logic*, illustrated in Figure 6.13, determines the values of L and R. During each scan-line update, the x-counter is zeroed and the L and R flags are reset. The x-counter is incremented by each *Pix* instruction and maintains the current x pixel coordinate. The L flag indicates that the current pixel is to the left of the trapezoid's left edge. The L flag is set when $x_l = x$. The R flag indicates that the current pixel is to the right of the trapezoid's right edge and is set when $x_r = x$.

Three of the *Pix* instructions require that the incoming subpixel depth be compared with the trapezoid's depth. Depths are represented as 24-bit unsigned integers. The depth comparator produces two exclusive outputs, *Less* and *More*. *Less* indicates that the subpixel depth is less than the trapezoid's depth while *More* indicates the subpixel depth is greater than the trapezoid depth.

The *Pix* instructions also require that the trapezoid depth value be updated if the current subpixel is inside the trapezoid. The adder always computes the sum of z and δz_p which is stored in z if the L flag bit is set.

Shadowing state transitions are implemented by the *Next State Logic*. This state transition logic can be implemented in a small PLA or with discrete logic. The next state logic, illustrated in Figure 6.14, computes the *Alter* signal which indicates that the opcode is one of $\{PixU, PixI, PixB\}$ and that the current subpixel is inside the trapezoid. The next state logic may modify the opcode's shadowing state field $op\langle 0{:}1\rangle$ only if *Alter* is asserted. *Alter* is computed as:

$$M \wedge T \wedge \neg B \wedge L \wedge \neg R$$

M is produced by the *control logic* and indicates that the opcode is one of $\{PixU, PixI, PixB\}$. $T \wedge \neg B$ indicates that the current subpixel is within the

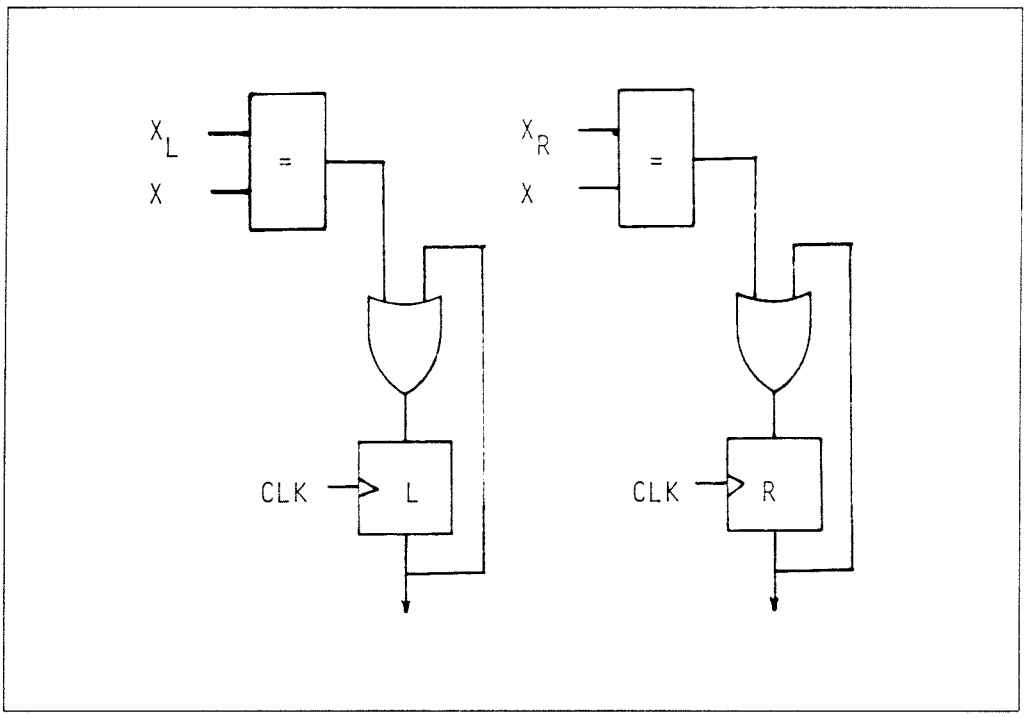


Figure 6.13: The *Extent Logic* determines the values of the L and R flag bits for each subpixel.

trapezoid's y extents. $L \wedge \neg R$ indicates that the current subpixel is within the trapezoid's x extents.

Since the majority of shadow polygon processor instructions are *Pix* instructions, the time required to perform the *Pix* operations determines the overall performance of the local shadowing processor. The previous discussion revealed that during a *Pix* instruction, the polygon processor must perform three computations. It must compare two 24-bit depth values, add two 32-bit values, and perform the shadowing state transition. Since the comparison and addition are performed in parallel, the polygon processor requires a clock period long enough to perform the addition and the shadowing state transition.

The ANIMAC-2 system requires that the local shadowing processor operate fast enough that it can determine local shadowing for n_s subpixels and n_t shadow polygon trapezoids. The value of n_s is determined by the image resolution, the number of physical processors, N_p, and the subsampling rate s. For the ANIMAC-2, n_s is equal to 262,144.

The value of n_t has not yet been determined. Earlier we decided that the visibility processors had to be designed to handle 6,250 frontfacing polygons

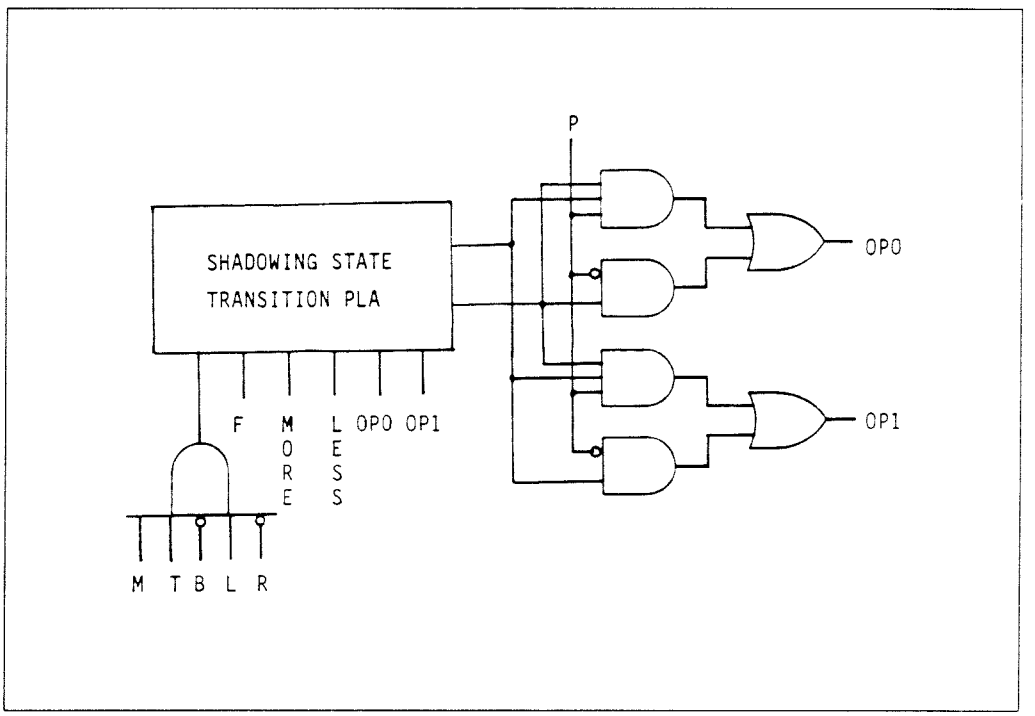


Figure 6.14: The *Next State Logic* performs the shadowing state transitions when necessary.

per frame time. For purposes of casting shadow volumes, these polygons can be considered to be triangles which cast shadow volumes that are triangular prisms. Each triangular prism is made from three quadrilaterals and two triangles which together require eleven trapezoids. Each shadow volume would require eleven trapezoids. Since shadow volumes do not need to be cast for backfacing polygons,

$$n_t = 6,250 * 11 = 68,750 \text{ trapezoids}$$

The total number of clock cycles required to compute a frame, c_f, is the sum of the loading time, the processing time, and the time required to drain the processor pipeline and can be written as:

$$c_f = 8 * n_t + n_s + n_t$$

where the constant 8 is the number of instructions required to initialize a processor. The total number of clock cycles per second, c, is found by multiplying c_f by the frame rate.

$$c = 30 * (8 * n_t + n_s + n_t)$$

Substituting in the value of n_t, we find that:

$$c = 30 * (8 * 68,750 + 262,144 + 68,750) = 26,426,820 \text{ cycles}$$

which suggests that the shadow polygon processor pipeline must operate with a 37 nsec. clock period. Noticing that most of the clock cycles are used loading the processor pipeline and draining it, suggests that the clock period can be increased by decreasing the number of trapezoids, n_t.

Crow has suggested several ways for decreasing the number of shadow polygons that need to be considered. We consider his suggestion that a shadow volume may be cast from a closed polyhedron's silhouette instead of from each of the polyhedron's polygons. Some analysis shows this to be a worthwhile suggestion.

Given a closed convex polyhedron composed of n triangular faces, we wish to determine how many trapezoids, t, are required to construct a shadow volume. To simplify the discussion, consider a spherical polyhedron with faces of similar area.

The shadow volume cast by a closed polyhedron consists of a front face which is constructed from the polyhedron's frontfacing polygons, side faces which are formed as projections of the polyhedron's silhouette edges, and a back face. The silhouette of the polyhedron consists of $\sqrt{\pi n}$ edges, each which casts a quadrilateral shadow volume side face. The front face is composed of $n/2$ triangles while the back face can be constructed as a polygon with $\sqrt{\pi n}$ vertices. This results in:

$$t = 3\sqrt{\pi n} + 2(n/2) + \sqrt{\pi n} - 1 = 4\sqrt{\pi n} + n - 1$$

The value of n_t depends upon the number of triangles distributed to a visibility processor. Earlier we determined that each visibility processor must be designed to handle 6,250 front facing triangles or 13,000 triangles in all. Table 6.4 tabulates the number of trapezoids to represent shadow volumes

Silhouette Shadow Volumes			
n	m	t	%
4	3250	55833	(81%)
9	1444	42265	(61%)
16	812	35207	(51%)
25	520	30913	(45%)
36	361	27991	(41%)
49	265	25871	(38%)

Table 6.4: Polyhedron silhouettes can be used to reduce the number of trapezoids needed to tile the shadow volume faces. The percentages indicate the reductions over the 68,750 trapezoids normally required to tile the shadow polygon faces.

for a scene composed of 13,000 triangles. These triangles are clustered as m polyhedra of n triangles each.

Table 6.4 suggests that silhouette shadow volume casting can be used to reduce the number of trapezoids by more than 50%. These results can be used to determine how many shadow polygon processors are needed in the local shadowing processor. The rest of this discussion will be based upon the assumption that we need 40,000 shadow polygon processors. This corresponds to a scene constructed of polyhedra with about ten faces each. We do not expect the scene to be uniform but suggest that 40,000 trapezoids is a conservative estimate for two reasons. First, polyhedra which are to appear smoothly shaded will be modeled with at least ten faces, and second, the scene will contain many polyhedra with more than ten faces, which further reduces the number of needed trapezoids.

The number of shadow polygon processor clock cycles can be recomputed for the new value of n_t as:

$$c = 30 * (8 * 40{,}000 + 262{,}144 + 40{,}000) = 18{,}664{,}320 \text{ cycles}$$

which requires a clock period of 53.5 nsec. The clock period can be lengthened by loading shadow polygon processors in parallel. If the pipeline of 40,000 processors can be loaded in ten parallel chains, the number of clock cycles becomes:

$$c = 30 * (8 * (40{,}000/10) + 262{,}144 + 40{,}000) = 10{,}024{,}320 \text{ cycles}$$

which requires a clock period just under 100 nsec.

It seems reasonable to expect that the shadow polygon processor can be designed to perform a 32-bit addition and the shadowing state transition within

100 nsec. in a CMOS technology. If the system implementor is willing to further constrain the shadowing environment, the number of shadow polygon processors, and the clock rate, can be further reduced. This constraint consists of only marking certain objects as shadow casting. Only objects marked as shadow casting need to generate shadow volumes. The software implementation discussed in Chapter 5 used this strategy.

Since it seems reasonable to expect that the shadow polygon processor can be constructed in the near future to operate at the required speed, the ANIMAC-2 system utilizes the full 40,000 shadow polygon processors. These processors require approximately the same amount of silicon area as the object processors. Therefore, it seems reasonable to expect to be able to fabricate 40 shadow polygon processors on a single die by the early 1990's.

This level of integration results in a local shadowing processor that is constructed of 1,000 shadow polygon integrated circuits organized into ten subsystems of 100 packages each. The local shadowing processor requires twice as many integrated circuits as the visible surface processor and probably will require twice the rack space. It would seem reasonable to expect that the visible surface processor, subpixel memory, local shadowing processor, and local shadow memory could be fabricated on about 50 printed circuit boards.

6.2.4 Local Shadow Map Processor

The local shadow map processor (LSMP) is responsible for generating local shadow maps (LSMs) for each of the sixteen virtual processors. The LSMP consists of a preprocessor and sixteen Bit-Plane processors. The preprocessor receives visible polygons from the clipping engine and prepares them for the Bit-Plane processors. Bit-Plane processors tile each polygon in their local shadow map. One Bit-Plane processor is associated with each of a processor's sixteen virtual processors.

Polygons arrive from the clipping subsystem tagged with a virtual processor identifier (VPID). Vertices arrive in viewing space coordinates. The preprocessor is responsible for transforming polygons from viewing space into ESM space. This transform is implemented as:

$$x_{esm} := ax_v + by_v + cz_v + d;$$
$$y_{esm} := ex_v + fy_v + gz_v + h;$$

The ANIMAC-2 design assumes that each visibility processor will receive 6,250 polygons from the clipping subsystem during each frame time. These polygons were triangles before being clipped against the virtual processor's clipping volume. Clipping against a plane can add one vertex to a polygon.

Each polygon is clipped against six planes which means that each triangle could end up having nine vertices when it arrives at the LSMP.

The simulations run in Chapter 3 showed that the two dimensional spatial subdivision of the image space resulted in little polygon fracturing. Even if polygons are fractured, the odds of a polygon having to be clipped against more than one plane is small because the vast majority of polygons are small.

The design of the LSMP is based on the assumption that at most one half of the 6,250 polygons will have been clipped against one plane. The total number of vertices arriving from the clipping subsystem, n_v, is:

$$n_v = (3,125 * 3) + (3,125 * 4) = 21,875$$

The performance criteria for the LSMP is established by n_v. This particular choice for n_v implies that the LSMP must handle a vertex every 1.52 usec.

The preprocessor must perform six floating point multiplies and six floating point adds every 1.52 usec. This computation only requires 7.9 million floating point operations per second (MFLOPS). Weitek [WEITEK83] currently produces 32-bit floating point multipliers and ALUs that operate at 8 MFLOPS when pipelined. The preprocessor is easily realizable with today's technology.

ESM coordinate space vertices are routed to the Bit-Plane processors. Each Bit-Plane processor is associated with a virtual processor and only tiles that virtual processor's polygons in its LSM. Bit-Plane processors select their polygons by inspecting the polygon VPIDs.

If we design to handle worst case performance, each Bit-Plane processor must be capable of tiling all 6,250 polygons in one frame time. Bit-Plane processors are implemented using a processor per pixel architecture. Since LSMs are large, ie. 0.5 Mb, it would be impractical to implement the Bit-Plane processors with a processor per object architecture since the clock rate would be excessively high. Processor per pixel architectures operate at a clock rate that is relatively independent of the number of pixels and therefore work well for this task.

The processor per pixel architecture used by the Bit-Plane processor is an adaptation of Fuchs's Pixel-Plane design. Fuchs's processors tile convex polygons, determining pixel depth, and coloring by evaluating the equation $aX + bY + c$ *en masse*. The reader should refer to the discussion in Chapter 3 for details on Fuchs's architecture.

Fuchs's machine must evaluate $aX + bY + c$ once for each of the polygon's edges to determine which pixels are inside the polygon. Four additional evaluations compute depth and pixel coloring. The Bit-Plane architecture only

needs to determine which pixels lie inside a polygon. This allows the Bit-Plane architecture to process polygons faster than the Pixel-Plane architecture.

Our earlier discussion suggested that the average polygon has 3.5 edges and thus requires 3.5 evaluations with the Bit-Plane architecture versus 7.5 for the Pixel-Planes architecture. Fuchs's group estimates that current Pixel-Plane chips are capable of scan-converting 1,000 quadrilaterals per frame time [POULTO85]. We expect that the Bit-Plane architecture can scan-convert at least twice as many polygons per frame time. Thus, it would seem that a Bit-Plane architecture implemented today could scan-convert in excess of 2,000 polygons per frame time. Future design and technological improvements are likely to allow Bit-Plane architectures to create LSMs at the rates demanded by the ANIMAC-2 architecture.

Each Bit-Plane processor utilizes half a million pixel processors. Clearly, many pixel processors must be fabricated on a die to make this economical to construct. Table 6.6 indicates how many integrated circuits will be needed to construct a 512 Kb LSM processor for various levels of integration.

Processors/Chip	Chips
256	2048
512	1024
1024	512
2048	256
4096	128
8096	64
16384	32

Table 6.6: Relationship between the number of Bit-Plane processors per die and the number of integrated circuits required to implement a 512 Kb shadow map.

The Bit-Plane architecture probably becomes economical to implement when 2048 pixel processors can be fabricated on a die. This level of integration implies that the Bit-Plane processor could fit on a single printed circuit card using surface mount technology. Achieving this level of integration is the only technological obstacle preventing the LSMP from being built today. Both the LSMP preprocessor and the Bit-Plane preprocessor are realizable with today's technology.

Fuchs's group reports fabricating a 64 pixel processor Pixel-Planes chip [POULTO85]. Figure 6.15 illustrates the Pixel-Plane pixel processor. The pixel processor consists of a small ALU which implements two bit serial adders and some control logic plus four registers for storing pixel state information. The Z register stores the pixel depth. The F register stores the sum

of $Ax + By + C_1 + C_2$. The I and P registers store pixel intensity values. Each of these four registers requires 24 bits of storage. We quickly realize that the size of a processor is limited by the space required to store 96 bits of information. Fuchs's reports that the Pixel-Planes chip uses 70% of its silicon area to implement storage, 20% to implement the bit serial adder-multiplier trees that compute $Ax + C_1$ and $Ax + C_2$, and 10% to implement the pixel processor ALUs.

Figure 6.16 illustrates the Bit-Plane pixel processor architecture. Since the Bit-Plane processor only needs to tile the polygon, the Z, F, I, and P registers are not needed. Furthermore, since depth comparisons are never performed, the ALU can be implemented with a single serial adder. Only two bits of state are required. The E bit indicates that the processor is enabled, i.e. currently inside all of a polygons edges, and the T bit indicates that the pixel has been tiled. The entire Bit-Plane processor should be implemented in less space than Fuchs's Pixel-Plane ALU.

The number of Bit-Plane processors that can be fit on a die with today's technology can be estimated from Fuchs's numbers. The Bit-Plane architecture eliminates the 70% of Fuchs's silicon area that was used for storage. Since we have eliminated 70% of the silicon area from Fuchs's chip, This suggests that a 64 processor Bit-Plane chip would require about 30% of the area of Fuchs's die. It doesn't seem unreasonable to expect to be able to fabricate 256 Bit-Plane pixel processors on a die with today's technology.

Fabricating the desired 2,048 processors on a die only requires eight times as many devices per die. If the density of components continues to double every two years, it should be feasible to fabricate chips of this complexity in the early 1990's.

The Bit-Plane processor's clock rate has not yet been discussed. If $Ax + By + C$ is to be computed to n bits precision, the time required to process an edge is n clock periods. The ANIMAC-2 LSMs require that n be computed to 22 bits. The clock period, τ, can be computed as:

$$\tau = \frac{\tau_f}{22 n_v}$$

where τ_f is the frame time of one thirtieth of a second and n_v is the number of vertices that must be handled during a frame time. Substituting ANIMAC-2 values, we find:

$$\tau = \frac{1}{30 * 22 * 21,875} = 69 \text{ nsec.}$$

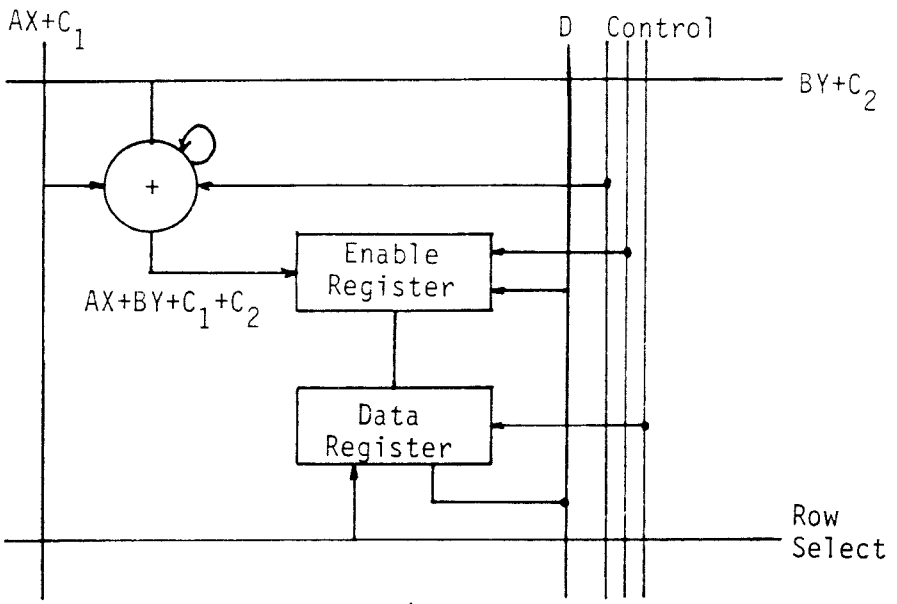


Figure 6.15: Fuchs's Pixel-Plane processor requires two serial adders and four 24-bit registers.

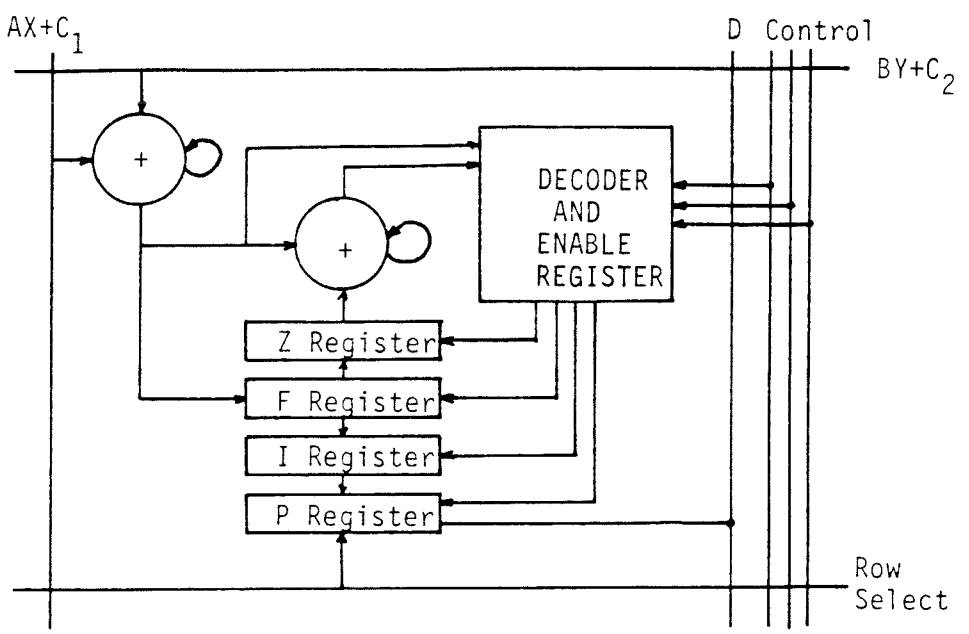


Figure 6.16: The Bit-Plane processor requires only one serial adder and two bits of state information.

The only computation that has to be performed during this 69 nsec. is a serial add and updating the ALU's E state bit. This should be easily implementable even with today's technology. Fuchs's reports that the Pixel-Planes chip is designed to run at 10 MHz, just a bit slower than the Bit-Plane chip.

6.2.5 Composite Shadow Map Processor

Figure 6.9 illustrates that the ANIMAC-2 visibility processor uses a merge processor (MP) to construct each CSM. Since each ANIMAC-2 visibility processor manages sixteen virtual visibility processors, sixteen merge processors are used to construct the sixteen CSMs.

The ANIMAC-2 virtual processors form their CSMs as the union of up to twelve quadrant shadow maps (QSMs). Each of these QSMs is computed in neighboring processors. Chapter 4 discussed how these twelve QSMs are formed and mentioned that all twelve of these QSMs are passed to a virtual processor during the creation of a processor's four QSMs.

The computation of a virtual processor's four QSMs is totally independent, so each merge processor computes the four QSMs in parallel. I first discuss how merge processors can be used to construct QSMs and the amount of interprocessor communications that is required. Later, I discuss the total amount of interprocessor communications that is required to compute all four QSMs.

A merge processor creates a QSM by unioning its LSM with the QSMs of three of its neighbors. A merge processor communicates with its neighbors over virtual channels. Figure 6.17 illustrates how each MP communicates with its neighboring MPs. The figure shows eight channels connecting merge processor m_{xy} with its eight neighbors. During the QSM merging process, a merge processor receives data from three of its neighbors and transmits data to three of its neighbors. During the creation of the QSMs, the interprocessor communications network can be broken down into four separate networks as illustrated in Figure 6.18. Each channel in this figure is a unidirectional channel. The vertical and horizontal channels are utilized by all four QSM computations while the diagonal channels are only used for two of the QSM computations.

The merge processor's primary task is to communicate with other processors. The first part of this section will analyze the amount of communication that is necessary between processors. The second part will discuss an algorithm that the merge processor might implement to perform this communication and create a QSM.

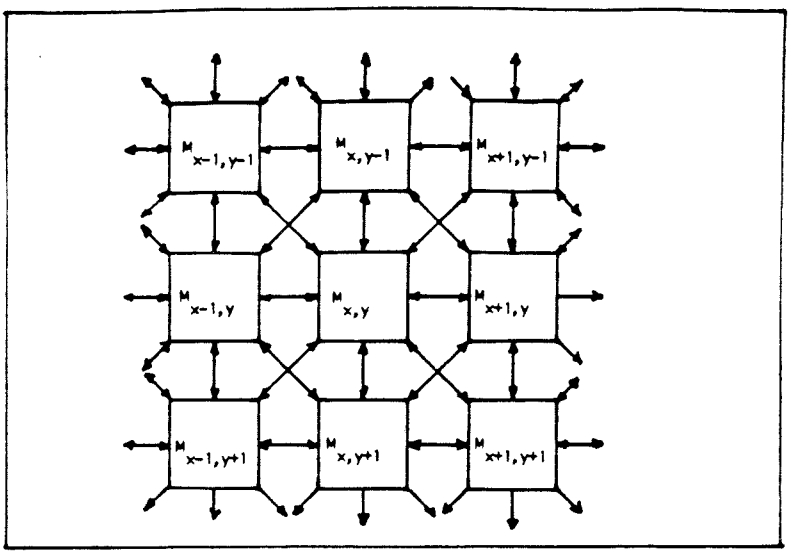


Figure 6.17: Virtual processors communicate with their eight neighbors over virtual channels.

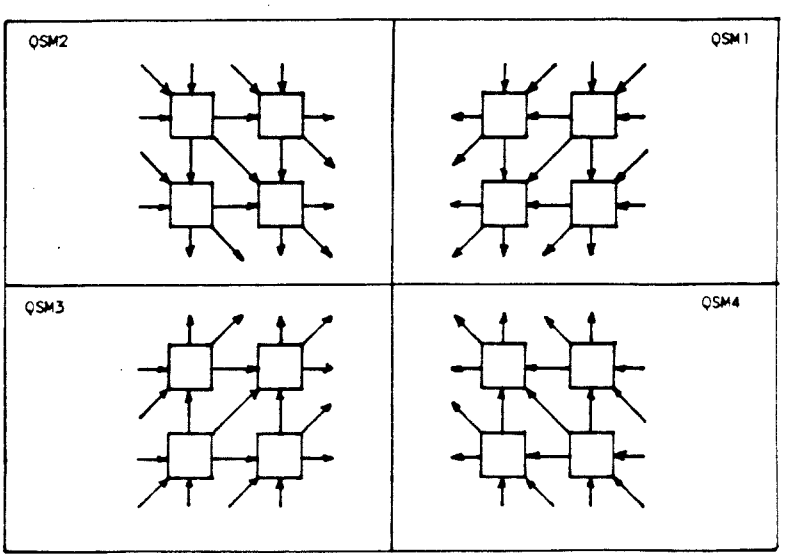


Figure 6.18: Computation of the four QSMs requires can be split into four separate computations, each with its own interprocessor connectivity.

Interprocessor communication can be studied by analyzing how information propagates through the interprocessor communication network. This analysis will allow us to compute the rate at which data must be transmitted across each channel and how much of a channel's capacity is actually utilized.

The amount of information that passes between processors depends solely upon the size of the shadow maps. Each virtual processor implements local and composite shadow maps of the same size. For the ANIMAC-2 system, each LSM and QSM requires 500 Kb of storage. During the merge operation, each processor sends the union of its LSM and QSM to three of its neighbors. The union of a processors LSM and QSM bit maps is another 500 Kb bit map. Thus, computing a QSM requires the transmission of 500 Kb across each channel during a frame time.

Transmitting 500 Kb every frame time suggests that the minimum channel rate is 15 Mb/sec. If the channel is not fully utilized, higher channel rates will be needed. We present an analysis of the channel rates needed to create QSMs by analyzing three algorithms for creating QSMs.

The first QSM creation algorithm is identical to the sequential QSM algorithm proposed in Chapter 4 and implemented in Chapter 5 and is called the Raster Sequential algorithm. Figure 6.19 illustrates a directed graph that is used to compute one QSM. Nodes represent virtual processors and edges represent interprocessor communication channels. The QSM computation is initiated by the root node sending the union of its LSM and QSM to its children. Each child processor forms its QSM by unioning the three shadow maps that it receives with its LSM. QSMs received from other processors are saved for future use as the processor's CSM. After the processor has formed its QSM, it transmits it to its children.

The time, τ, required to create all of the shadow maps depends upon the maximum depth of the graph, d. Since a shadow map must be received by a processor before it can transmit its shadow map, the total communications time for this algorithm is:

$$\tau = \frac{s_{csm} d}{C_R}$$

where, s_{csm} represents the number of bits that must be transmitted over each channel, and C_R represents the channel bit rate. We notice that the maximum depth of the graph is $w+h-1$, where w is the width of the processor array and h is the height of the processor array. The previous equation can be rewritten as:

$$\tau = \frac{s_{csm}(w+h-1)}{C_R}$$

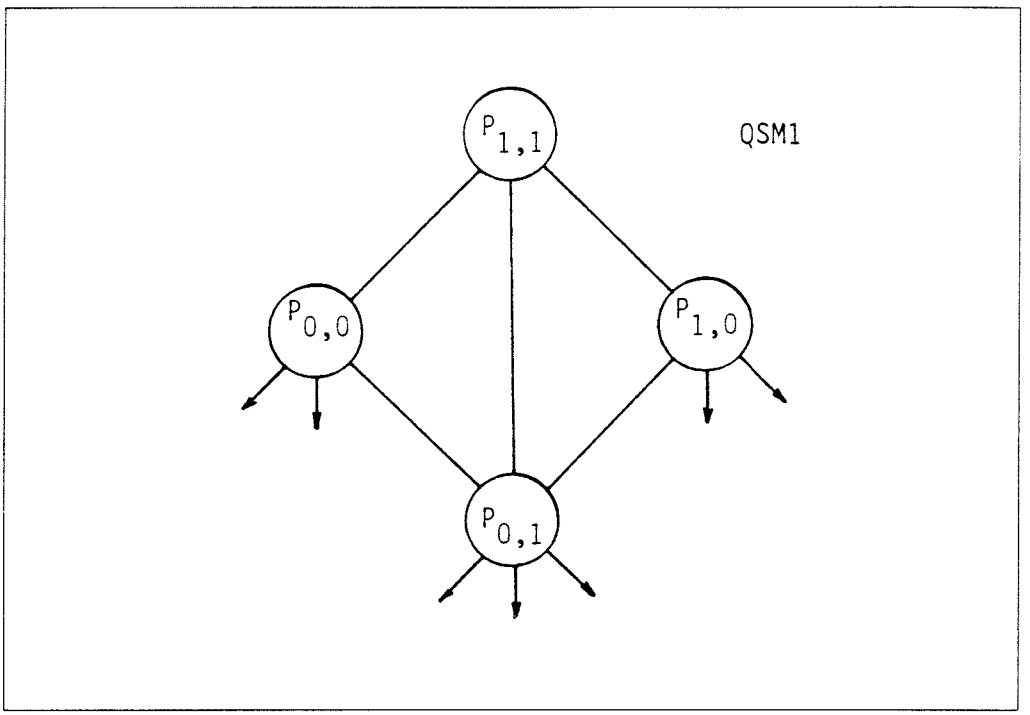


Figure 6.19: A directed graph can be used to create a QSM.

Which can be rewritten to solve for C_R as:

$$C_R = \frac{s_{csm}(w + h - 1)}{\tau}$$

Substituting ANIMAC-2 values for s_{csm}, w, h, and τ, we find the channel bit rate to be:

$$C_R = 524,288 * (16 + 16 - 1) * 30 \approx 487 \text{ Mb/sec.}$$

Thus, Raster Sequential merging of shadow maps requires virtual channel rates in excess of 487 Mb/sec. This algorithm is very inefficient when compared to the minimum channel rate that we earlier computed to be 15 Mb/sec.

The Pixel Sequential algorithm creates QSMs more efficiently than the Raster Sequential algorithm. Figure 6.20 helps explain how this algorithm operates. QSM creation can be pipelined by computing the QSM pixels that fall along a ray. All QSM pixels can be created by raster scanning the ray across the ESM. This algorithm can be parallelized by scanning multiple rays

across the ESM. A separate ray may be cast to scan out each QSM that directly depends upon the ESM. This method requires time proportional to the number of pixels in a QSM plus the propagation delay, ϵ, through the interprocessor network. This can be written as:

$$\tau = \frac{s_{csm} + \epsilon}{C_R}$$

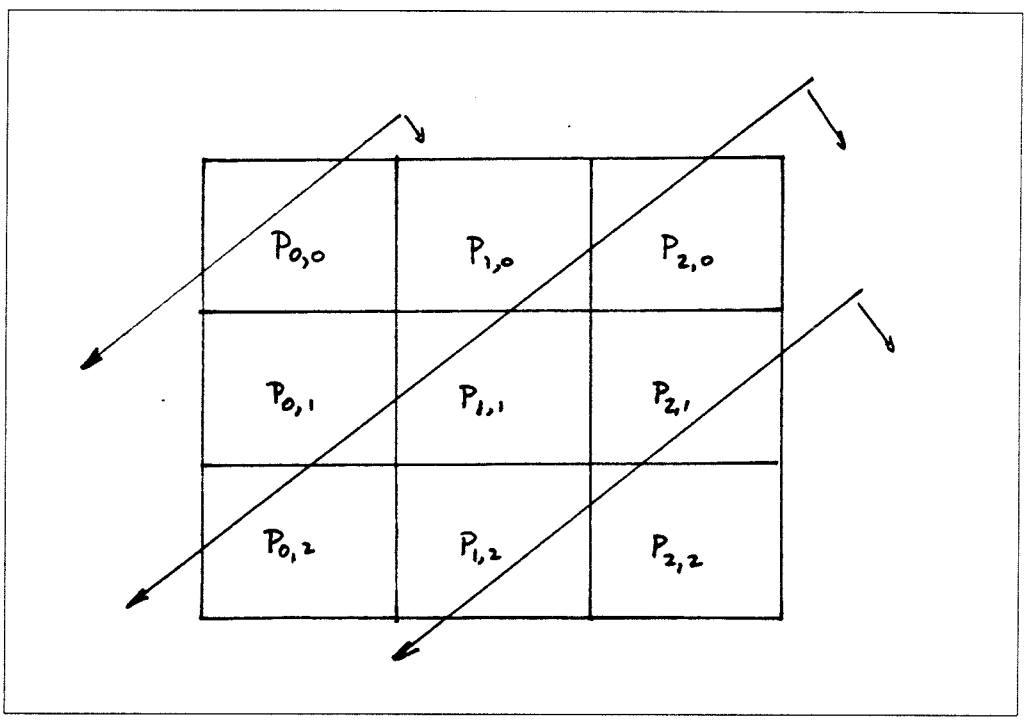


Figure 6.20: QSMs can be merged in a pixel parallel manner. The ray illustrates the current pixel that is being computed. During the computation, the ray raster scans the QSM.

Substituting ANIMAC-2 values into the equation for τ, and rewriting to solve for C_R, we find:

$$C_R = (524,288 + 0) * 30 \approx 15 \text{ Mb/sec.}$$

The Pixel Sequential algorithm performs at the minimum channel bit rate when the propagation delay ϵ is small. Note that ϵ can be quite long before it starts to have a substantial influence upon C_R.

The two CSM creation algorithms both operate similarly. Each algorithm operates by sending a portion of a processor's shadow map to its dependent processors, which in turn do the same. We see two extreme behaviors when we (1) send the entire shadow map, and (2) send only one pixel of the shadow map. We can generalize this algorithm by specifying that a processor sends a packet of p bits to each of its dependent processors. Each of these dependent processors updates a portion of its shadow map and sends that portion in a packet to its dependents. Updating a portion of a shadow map requires time and we model this time as δ which is measured in sec./bit. We can model the time required by this algorithm as:

$$\tau = \frac{s_{csm}}{p}\left(\frac{p}{C_R} + \delta p\right) + (w + h - 2)\left(\frac{p}{C_R} + \delta p\right)$$

The first term represents the number of packets times the time required to send and process a packet. The second term represents the time required to drain the pipeline. We can rewrite this equation to solve for channel rate C_R. We find:

$$C_R = \frac{(s_{csm} + p(w + h - 2))}{\tau - dp(s_{csm} + p(w + h - 2))}$$

Substituting ANIMAC-2 values into the equation for C_R, we find:

$$C_R = \frac{(524,288 + 30 * p)}{\frac{1}{30} - 524,288 * d * p - 30 * p}$$

Table 6.7 tabulates values of C_R for various packet lengths and delay times. We notice that for reasonable packet sizes and delays, the channel bit rate can be kept below 20 Mb/sec. The table also shows that this model produces the same bit rates that were previously calculated for the Raster Sequential and Pixel Sequential algorithms. The Pixel Sequential algorithm is modeled by $p = 1$ while $p = 524288$ models the Raster Sequential algorithm.

The delay term mandates how fast pixels must be accessed from the the LSM and written into the QSM memory. If data is accessed from the LSM and QSM memories at 16 bits per cycle, a 10 nsec./bit delay will require less than 20 Mb/sec. This delay implies that 16 bits of data must be processed in 160 nsec. This suggests that the LSM and QSM memory be accessed in a little under 160 nsec. The memory access time can be made longer by increasing the packet size. Notice that if $p = 1024$, we can still use a 10 nsec./bit delay and run under 20 Mb/sec. This combination of packet size and bit delays suggest accessing 1024 bits every 10 usec.

Channel Bit Rates (Mb/sec)						
p	$\delta = 0$	$\delta = 0.5$	$\delta = 1$	$\delta = 5$	$\delta = 10$	$\delta = 50$
1	15.73	15.85	15.98	17.07	18.67	73.67
16	15.74	15.87	15.99	17.09	18.68	73.96
1024	16.65	16.79	16.93	18.16	19.98	99.41
4096	19.42	19.61	19.80	21.50	24.09	663.8
16384	30.47	30.95	31.43	35.95	43.83	
65536	74.71	77.61	80.74	119.3	295.4	
131072	133.7	143.3	154.3	403.3		
262144	251.7	287.9	336.3			
524288	487.6	644.8	951.6			

Table 6.7: Interprocessor channel bit rates computed for various packet lengths, p, and bit delays δ (nsec./bit).

Figure 6.9 suggests that virtual channels can be multiplexed onto channels between physical processors. The virtual to physical processor mapping assures that a virtual processor's neighbor resides in one of its physical processor's neighbors. Figure 6.8 illustrated how physical channels wrap around to form a torus in the x, y, and diagonal directions.

The amount of communications that is required on a physical channel depends upon the virtual channel bit rate, C_R, the ratio of virtual processors to physical processors, σ, and the number of QSMs which must be computed. Earlier it was mentioned that four QSMs are computed in parallel using the processor interconnect illustrated in Figure 6.17. Figure 6.21 illustrates the bandwidth required on each channel to compute the four QSMs in parallel. The vertical and horizontal channels must handle $4C_R$ bits per second while the diagonal channels only need to handle $2C_R$. Each of these channels is illustrated as two unidirectional channels requiring half the bandwidth.

If $C_R = 20$ Mb/sec, each of the eight unidirectional horizontal and vertical channels must handle 40 Mb/sec while the eight diagonal channels need to handle 20 Mb/sec. The ANIMAC-2 maps sixteen virtual processors onto each physical processor. Thus the amount of information that flows between physical processors must be multiplied by 16. We find that the horizontal and vertical physical channels must handle 640 Mb/sec while the diagonal channels need only 320 Mb/sec.

The processor intercommunication demands are high but not excessive. Today's fiber-optic technology can transmit data in excess of 565 Mb/sec. [ELECTR85]. A processor could implement each of its unidirectional channels with a 320 Mb fiber-optic link. The eight bidirectional interprocessor channels would require sixteen fiber-optic strands. The interprocessor communication

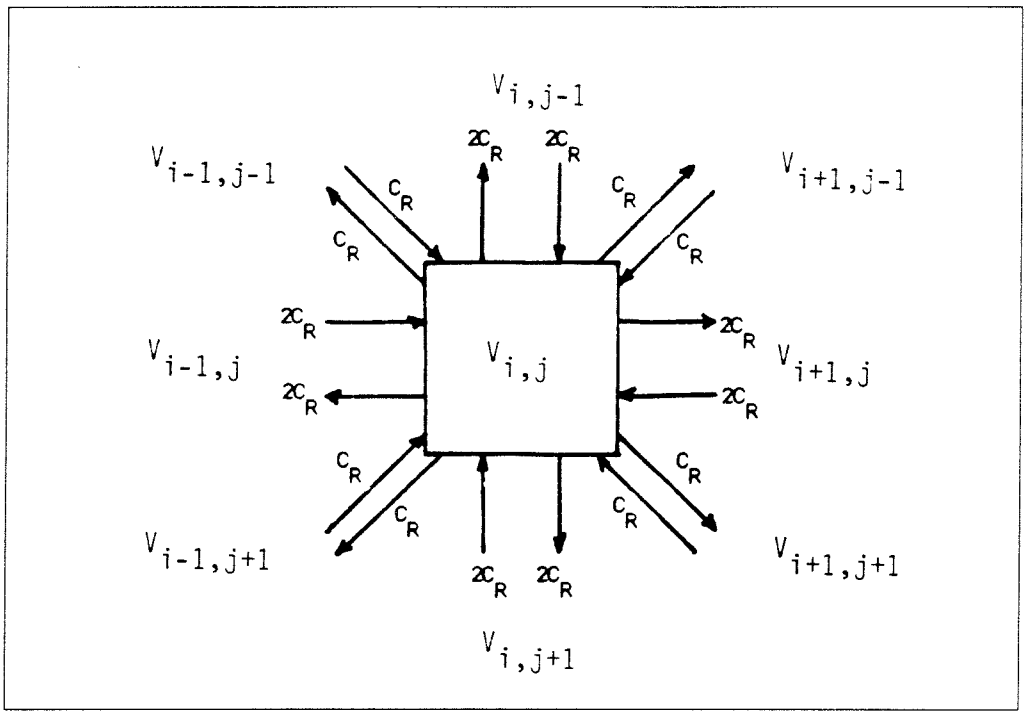


Figure 6.21: Bandwidth requirements for each of a virtual processor's channels.

may be the most difficult part of the ANIMAC-2 system to implement. Although expensive, it is technologically feasible to construct the interprocessor communication channels.

6.2.6 Illumination Processor

Figure 6.9 illustrates that each ANIMAC-2 visibility processor implements an illumination processor (IP) for each of its sixteen virtual processors. These illumination processors read subpixel data from the subpixel buffer (SPBUF) and determine what color values to assign to the subpixels. Subpixel color values are stored in the frame store where they are spatially filtered and presented to the viewer.

The IP discussed in this section implements a Gouraud [GOURAU71] illumination model. The color of a subpixel is described by three primary color

intensities, I_R, I_G, and I_B which represent red, green and blue respectively. These values are computed as:

$$I_R = I_{a_R} + I_{d_R}(N \cdot L)\phi$$
$$I_G = I_{a_G} + I_{d_G}(N \cdot L)\phi$$
$$I_B = I_{a_B} + I_{d_G}(N \cdot L)\phi$$

The I_a terms represent the amount of ambient light which is added into every pixel. The I_d terms represent the amount of diffuse light which can be added into each pixel. The actual amount of diffuse light is attenuated by two factors. The $N \cdot L$ term effects a Lambert Law light intensity function and the ϕ term attenuates the subpixel's intensity when it is in shadow.

The illumination processor must determine subpixel colors for all of a virtual processor's subpixels. In the ANIMAC-2 system, a virtual processor has a 32 by 32 screen area. Since each pixel is tiled with 16 subpixels, an illumination processor must color 16,384 subpixels every frame time. This corresponds to computing a new color every 2.06 usec. This section will show that the computations can be pipelined so that the IP can operate on a 2 usec clock.

The IP must perform several tasks. First, it must retrieve the subpixel data from the SPBUF and LSBUF. Second, it must determine foreign shadowing by inspecting the composite shadow map. Finally, it must compute the subpixel's color.

Figure 6.22 illustrates the illumination processor architecture as a pipeline of processes. The subpixel access processor (SAP) retrieves the subpixel data from the SPBUF and the local shadowing bit from the LSBUF. The viewing transform processor (VTP) transforms the image space subpixel coordinates back into viewing space coordinates. The CSM transform processor (CTP) transforms the subpixel's viewing space coordinates into CSM coordinates. The CSM inspection processor (CIP) inspects the CSM to determine whether the subpixel lies in foreign shadow. The color processor (CP) retrieves values for I_a and I_d from a color property table (CPT) which is indexed by the object's identifier. The Lambert processor (LP) determines the dot product $N \cdot L$. The subpixel color processor (SCP) uses all of this information to color the subpixel.

6.2.6.1 Subpixel Access Processor

The SAP retrieves a subpixel's data from the SPBUF and LSBUF. This requires one read access per 2 usec. to be performed by each of the sixteen SAPs. Thus the SPBUF and LSBUF must be able to deliver a subpixel's data every

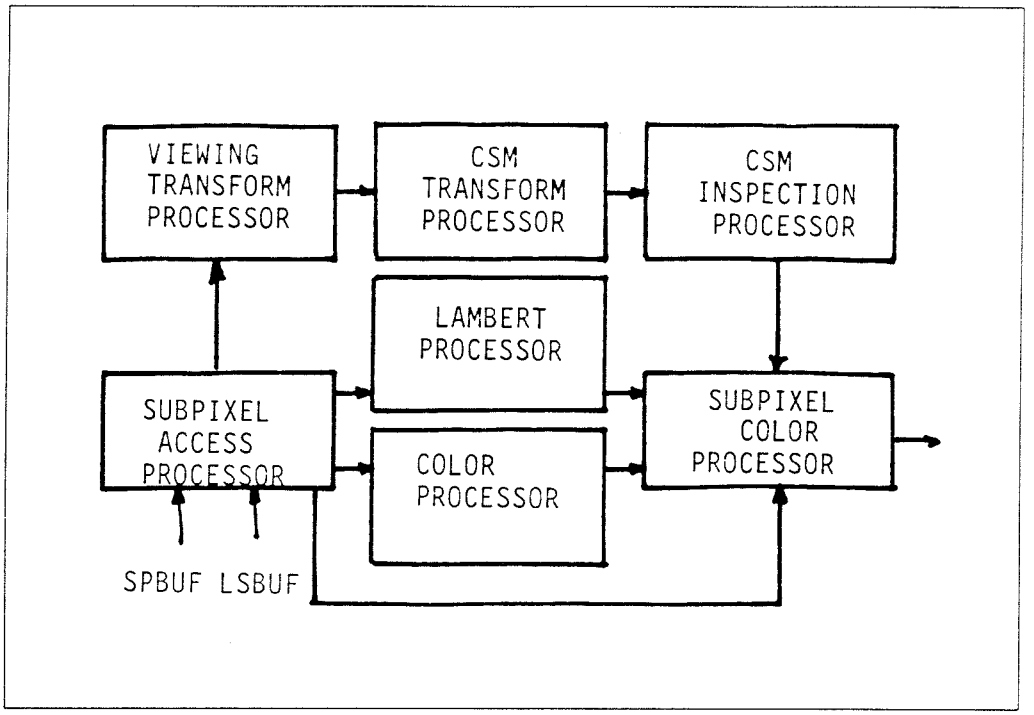


Figure 6.22: The Illumination Processor can be implemented as a pipeline of specialized processors.

125 nsec. An earlier discussion suggested that the SPBUF and LSBUF are accessed at faster rates by the VSP and LSP and thus, the SAP imposes no additional demands upon these memory systems.

The SAP retrieves the subpixels depth, z, object id, id, and surface normal vector, N, from the SPBUF. The local shadowing bit, s_l, is retrieved from the LSBUF. The subpixel's depth is passed onto the VTP. The object identifier is passed to the CP. The normal vector is passed to the LP and the local shadowing bit is passed to the SCP.

6.2.6.2 Viewing Transform Processor

The viewing transform processor implements a transform similar to T_{pv}^{-1} which was discussed in Chapter 5. Given a subpixel with image space coordinates (x, y, z), viewing space coordinates (x_v, y_v, z_v) are computed by equations of

the form:

$$z_v = \frac{k_1}{zk_2 - k_3}$$
$$x_v = z_v(k_4 x - k_5)$$
$$y_v = z_v(k_6 y - k_7)$$

where $k_1, \ldots, k_7$ are floating point constants that are determined from the projection and virtual processor region.

The computation of (x_v, y_v, z_v) requires three floating point additions, three floating point multiplications and one floating point divide. The total computation can be performed in the allotted 2 usec. with a floating point ALU capable of operating at 3.5 MFLOPS.

6.2.6.3 CSM Transform Processor

The CTP receives the viewing space subpixel coordinates (x_v, y_v, z_v) and transforms them into CSM space coordinates, (x_{csm}, y_{csm}). These coordinate are computed as:

$$x_{csm} := ax_v + by_v + cz_v + d;$$
$$y_{csm} := ex_v + fy_v + gz_v + h;$$

The LTP computation requires six floating point multiplies and six floating point additions and can be computed with a floating point ALU that operates at 6 MFLOPS.

6.2.6.4 CSM Inspection Processor

The CIP receives the integer CSM subpixel coordinates and determines if that subpixel lies in foreign shadow. The CIP checks if x_{csm} and y_{csm} are within the LSM array bounds. If the point is in bounds, the CIP computes the array indices of the subpixel and the eight CSM pixels surrounding the subpixel and samples these nine CSM pixels. The CIP determines that the subpixel lies in foreign shadow only if all nine CSM pixels are set. Foreign shadowing is indicated with the s_f bit which is passed to the SCP.

The CIP computation requires two range comparisons, nine array index computations and nine CSM accesses. All arithmetic can be performed with integer arithmetic. The two range comparisons can be performed quickly with four comparisons. Each array index requires one multiplication and one addition. None of these operations requires time even near the allotted 2 usec.

The CSM accesses can be arranged so that all nine accesses have a full 2 usec. Depending upon how the CSM is organized, the nine accesses require nine, three or one memory read operation. Under worst case considerations,

the CSM must be read nine times during each 2 usec. which indicates that the CSM must have a 220 nsec. read cycle time.

6.2.6.5 Lambert Processor

The Lambert processor (LP) computes the dot product $N \cdot L$. L is the light source direction vector and can be considered a constant. A new N arrives with each subpixel. Both N and L are represented with 16-bit fixed point numbers. $N \cdot L$ is computed as:

$$N \cdot L = \frac{(N_1 L_1 + N_2 L_2 + N_3 L_3)}{\sqrt{N_1^2 + N_2^2 + N_3^2}}$$

The division can be implemented as a multiplication with the reciprocal of the magnitude of N. Table lookup can be used to evaluate the square root function and to determine the reciprocal. None of the operations required for the LP computation requires more than 2 usec.

6.2.6.6 Color Processor

The color processor uses the objects *id* to retrieve ambient and diffuse colors from the color property table (CPT). These colors, I_a and I_d are stored for each triangle as 16-bit fixed point numbers. The CPT stores six values for each of the 100,000 triangles and is accessed by all sixteen color processors. It must be implemented with an effective read cycle time of 125 nsec.

6.2.6.7 Subpixel Color Processor

The subpixel color processor receives $N \cdot L$, I_a, I_d, and shadowing indicators s_l and s_f. It computes the subpixels color by computing:

$$I_R = I_{a_R} + I_{d_R}(N \cdot L)\phi$$
$$I_G = I_{a_G} + I_{d_G}(N \cdot L)\phi$$
$$I_B = I_{a_B} + I_{d_G}(N \cdot L)\phi$$

The shadowing attenuation factor ϕ is either 1 or some constant less than one if the subpixel lies in shadow which can easily be determined as $s_l \vee s_f$.

The entire computation can be computed with five integer multiplications and three integer additions in well under 2 usec.

6.3 Conclusions

This chapter has illustrated how the techniques developed in this thesis can be used to construct two real-time computer graphics systems. The author has

also presented two new VLSI architectures that are useful for certain computations. One of these used an object per processor architecture to compute local shadowing effects. The other used a simplified processor per pixel architecture to create local shadow maps.

The two systems illustrated in this chapter would be difficult to implement with today's technology. Discussions in the chapter illustrated that these systems will become realizable in the not too distant future. The ANIMAC-1 visibility subsystem will become economically realizable when about 40 object processors can be fabricated upon a single die. The ANIMAC-2 visibility subsystem will be economically realizable when: (1) 40 object processors can be fabricated upon a single die, (2) when 2,048 Bit-Plane pixel processors can be fabricated upon a single die, and (3) when several hundred megabit channels become relatively inexpensive. All of these conditions should occur during the early 1990's.

Implementing the ANIMAC-2 system requires roughly 80,000 integrated circuits to implement its local and foreign shadowing calculations. This seems like a huge price to pay for shadows but when compared to the increased time demands of the software implementation, this amount of hardware is readily justified. Still, it is interesting to compare the number of integrated circuits that would be required if shadowing was implemented solely as a local process. A few definitions will simplify the discussion.

N_v	The number of virtual processors (regions)
N_o	The number of triangles in the scene
η	The expected parallel efficiency
s_{esm}	The size of the ESM
k_t	The number of shadow volume trapezoids per triangle

The number of integrated circuits required to implement the ANIMAC shadowing algorithm can be computed by first determining the number of Bit-Plane processors required for the LSMs and the number of shadow volume processors required for the LSP.

LSMs require a number of bits, N_{lsm}, that depends upon the size of the ESM and the number of processors. N_{lsm} can be computed as:

$$N_{lsm} = \begin{cases} 0, & \text{if } N_v = 1; \\ s_{esm}\sqrt{N_v}, & \text{otherwise.} \end{cases}$$

The local shadowing processors require a number of processors, N_{svol}, which is independent of the number of processors and depends only on the

number of objects, the expected parallel efficiency, and the ratio of shadow volume trapezoids to objects. N_{svol} can be computed as:

$$N_{svol} = k_t \frac{N_o}{\eta}$$

The number of integrated circuits can be computed given two ratios which describe the number of processing elements that can be fit on a die:

k_1 The number of Bit-Plane processors per die

k_2 The number of shadow volume processors per die

With these definitions, N_A, the number of integrated circuits required to implement the ANIMAC shadowing computations can be written as:

$$N_A = \frac{s_{esm}\sqrt{N_v}}{k_1} + \frac{k_t N_o}{k_2 \eta}$$

If instead of using the ANIMAC shadowing algorithm, each processor implements the local shadowing algorithm for all potential shadow casting objects, N_L integrated circuits will be required, where:

$$N_L = \sqrt{N_v} \frac{k_t N_o}{k_2 \eta}$$

The behavior of N_A and N_L, as functions of N_v, determines which algorithm requires the least integrated circuits for a given N_v, holding constant all other variables. N_v', the number of processors for which $N_A = N_L$ can be solved for as:

$$N_v' = \left(\frac{k_1 k_t N_o}{k_1 k_t N_o - s_{esm} k_2 \eta} \right)^2$$

Substituting in ANIMAC-2 values, we find:

$$N_v' = \left(\frac{2048 \cdot 6.4 \cdot 50000}{2048 \cdot 6.4 \cdot 50000 - 8000000 \cdot 40 \cdot 0.5} \right)^2 \approx 1.75$$

The computed value of N_v' would require some 21,120 integrated circuits to be used for shadowing computations. Clearly, N_v' is so small that it suggests that having each processor compute only local shadow effects is only feasible for a uniprocessor under these constraints. N_v' also depends upon our choice of N_o, η, s_{esm}, k_t, k_1, and k_2 and would need to be evaluated for different environments.

The ANIMAC-2 system requires some 78,500 integrated circuits to implements its shadowing computations. A similar system which computed only local shadows would require about 256,000 integrated circuits. This example very vividly illustrates the savings introduced by the ANIMAC shadowing algorithm.

7

Conclusions and Future Work

The work presented in this dissertation can be summarized as contributions to four specific research areas. These include understanding scene composition, utilizing parallelism in real-time computer graphics, developing a shadowing algorithm for a parallel environment, and utilizing VLSI architectures to implement high performance real-time computer graphics systems.

The analysis of scene composition led to several interesting findings. The most interesting discovery was that the spatial non-uniformity of scenes can be characterized by an asymptotic value. This finding suggests that as more and more processors are used to render a scene, the distribution of objects to processors starts to look self-similar. Other findings typically support commonly held opinions about scene composition and are reported herein since the literature is relatively void of such observations.

This thesis proposes that parallelism can be utilized to build very high performance real-time computer graphics engines by assigning processors to portions of the image space. Simulation results suggest that one of the best ways of assigning processors to image space regions is to perform a two-dimensional division of the image space and assign several regions to each processor. These regions are assigned to processors in a way that maps regions topologically far apart onto a processor and maps neighboring regions onto neighboring processors.

The ANIMAC shadowing algorithm proposes that shadowing computations be divided into two separate processes. One process determines shadows cast by objects within a processor's viewing space subvolume and as such need

only rely on data local to the processor. The second process determines shadows cast by objects in other processors. This process can be implemented so that interprocessor communications is limited to a processor's eight neighbors.

The ANIMAC shadowing algorithm is also well suited to software implementations since it allows image creation to be subdivided into tasks of manageable proportions. Its structure allows it to be easily ported to, and make good use of, today's general purpose multiprocessors.

Several VLSI architectures have been proposed for computer graphics applications. This thesis suggests ways of utilizing these and new VLSI architectures to construct computer graphics systems whose aggregate performance exceeds the performance attainable from any one of these VLSI architectures.

These systems are complex, the most sophisticated will probably require some 100,000 integrated circuits when it becomes realizable in the 1990's. However, this level of complexity is not out of line when compared to today's real-time computer graphics systems.

The work presented in this dissertation suggests several areas for future work. The work on scene analysis was performed on a very small sample space and studied relatively few of the variables that contribute to scene composition. Much more work is needed in this area and will hopefully lead to a statistical model of scene composition that can be used to analyze algorithms and to drive simulators.

The tesselated regular two-dimensional spatial subdivision techniques appear to offer relatively high parallel efficiencies. Irregular spatial subdivision techniques appear to offer higher efficiencies but depend upon being able to determine an optimal image space partitioning. These irregular spatial subdivision architectures deserve further attention. It would be interesting to study whether near optimal partitionings can be determined without requiring intensive computation.

Shadow Maps have been implemented using bit map representations and suffer from sampling problems related to their discrete representation. Shadow Maps may be implemented in other representations that maintain more accurate descriptions of shadow boundaries. Future research might develop shadow map representations which are much more accurate than bit maps yet can be implemented efficiently.

Appendix A

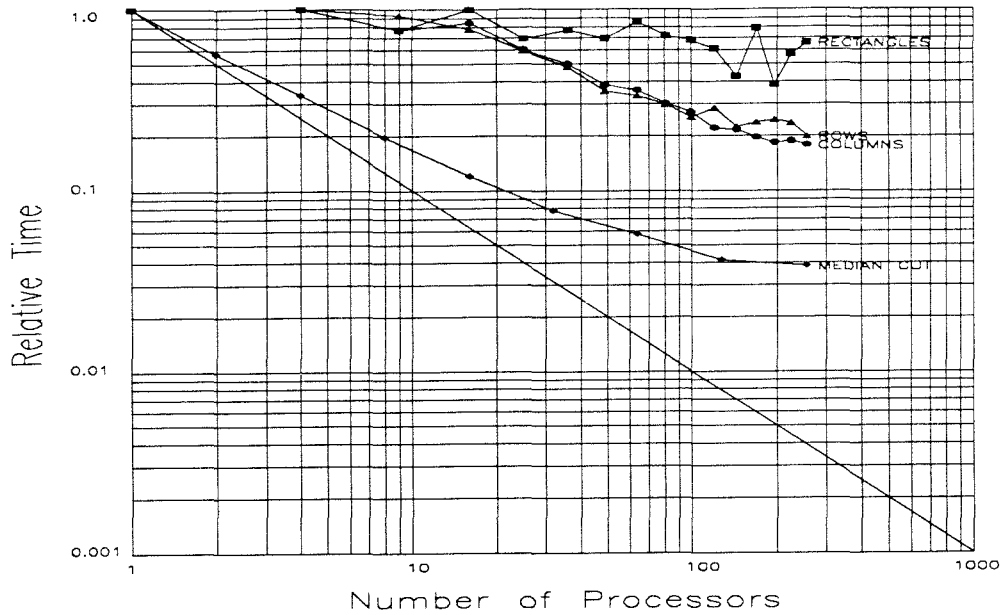
Simulation Results

This appendix contains data from simulations of various spatial subdivision strategies discussed in Chapter 3. Figures $A.1$–$A.17$ correspond to nonvirtual spatial subdivision techniques. Figures $A.18$–$A.34$ correspond to the virtual spatial subdivision techniques.

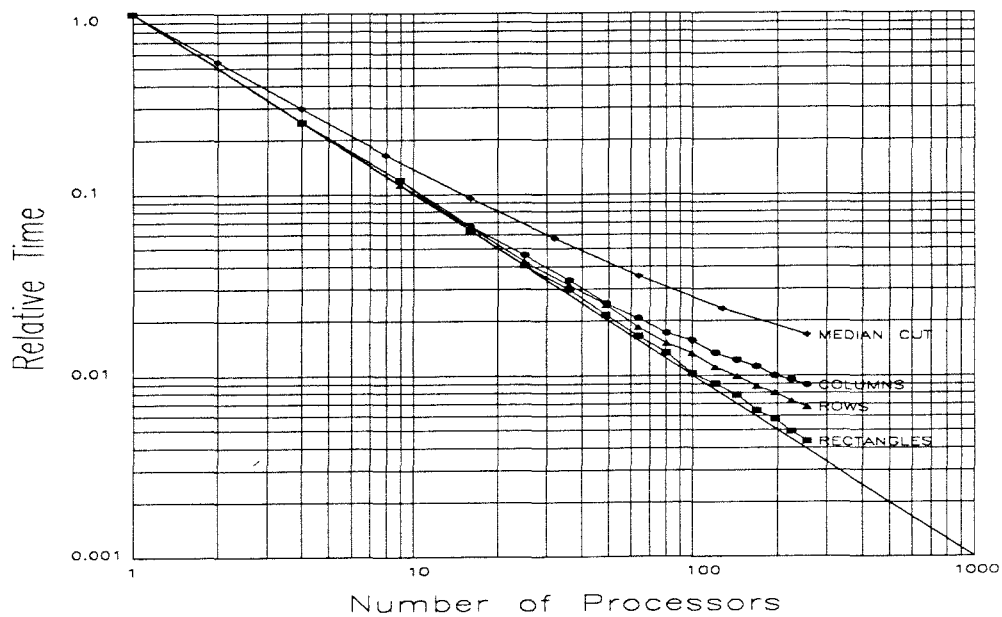
Each set of figures is plotted similarly. The nonvirtual figures consist of four subfigures. The first subfigure is a photograph of the scene that was used to drive the simulator. The second subfigure illustrates the four methods that were used to subdivide the image space and is illustrated for configurations of sixteen processors. The third subfigure is a plot of performance for the cost metric which corresponds to latency. The fourth subfigure plots performance for the cost metric which corresponds to throughput.

The virtual plots each contain six subfigures. These six subfigures plot performance for the six different spatial subdivision techniques. These are *rows*, *columns*, *rectangles*, *virtual rows*, *virtual columns*, and *virtual rectangles*.

Figure A.1: X-Wing Fighter (xwf0)



File: xwf0.sta, 994 Polygons
Time = Max(n)



File: xwf0.sta, 994 Polygons
Time = Avg(n)

Figure A.1: X-Wing Fighter (xwf0) *(cont.)*

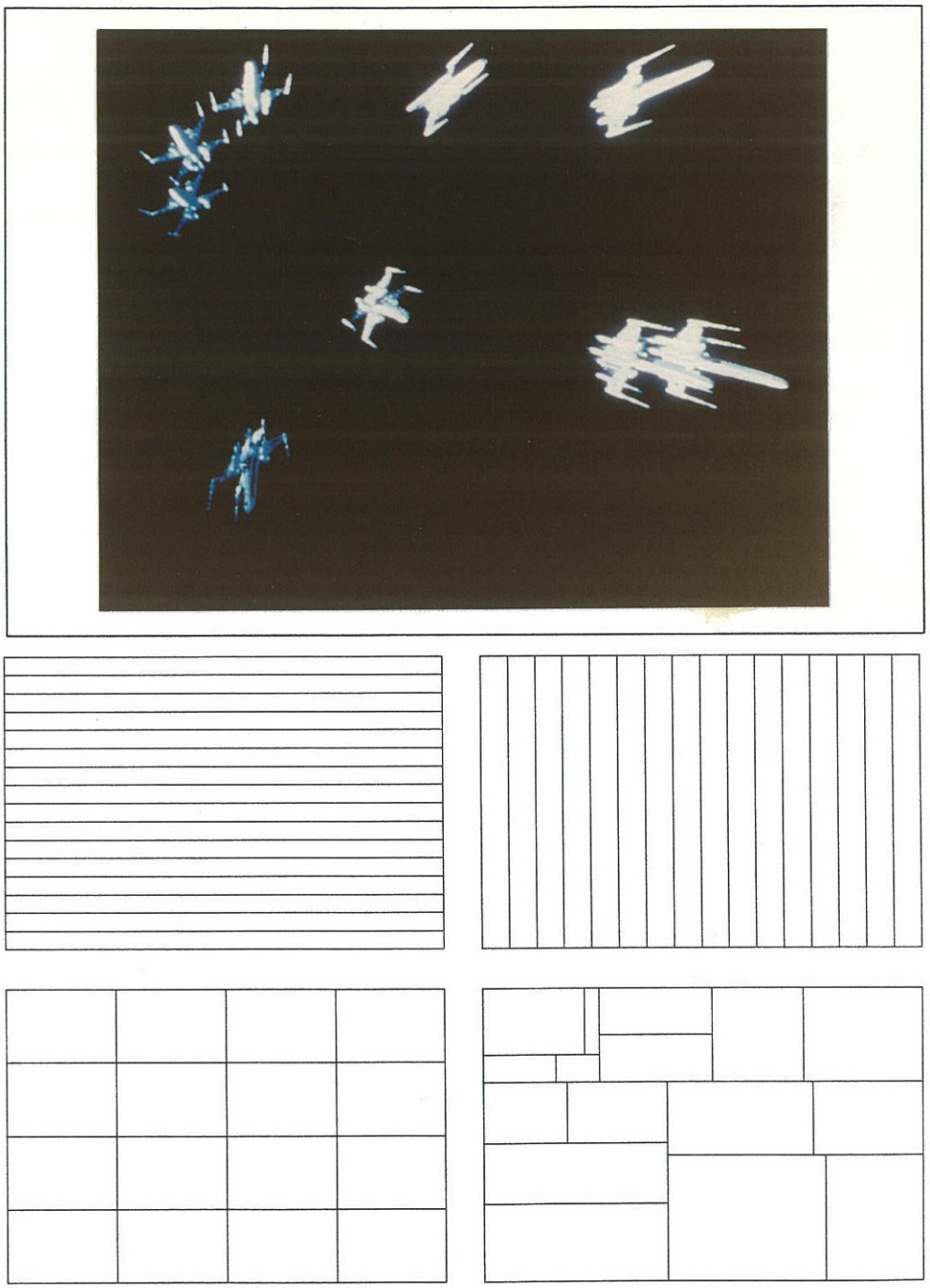
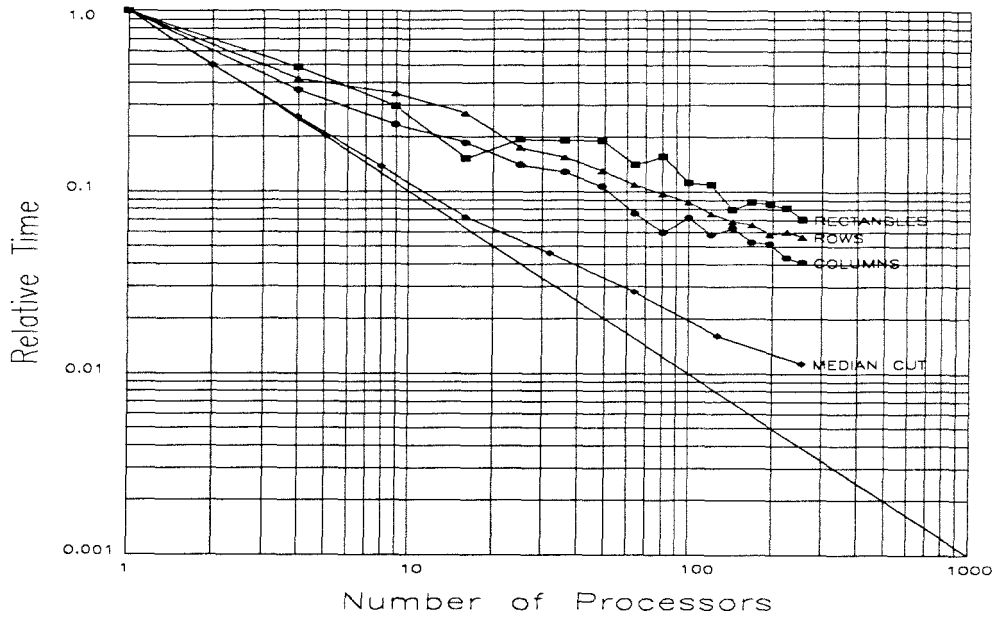


Figure A.2: X-Wing Fighter (xwf1)



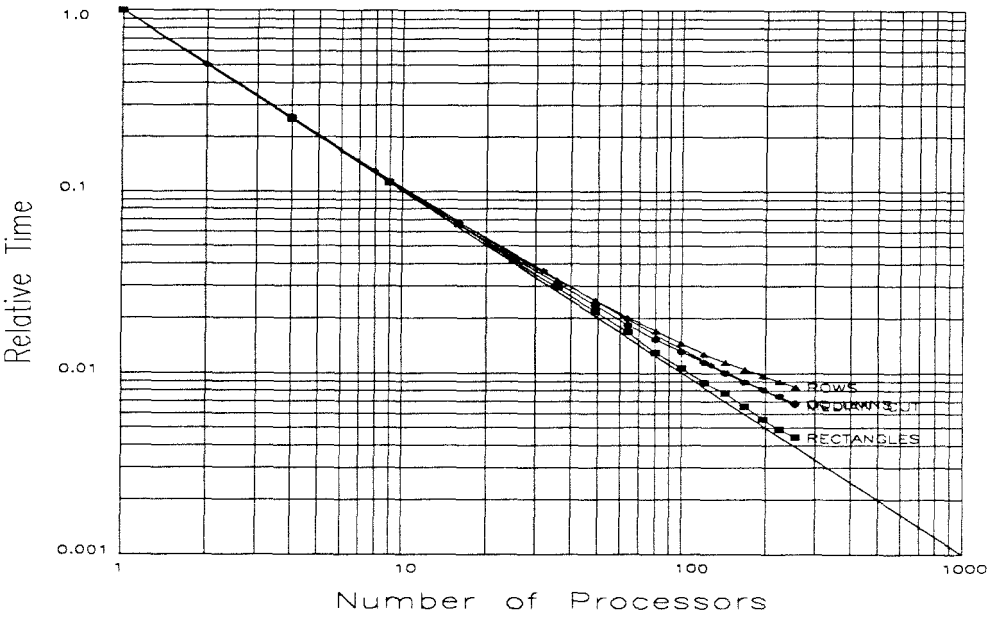


Figure A.2: X-Wing Fighter (xwf1) *(cont.)*

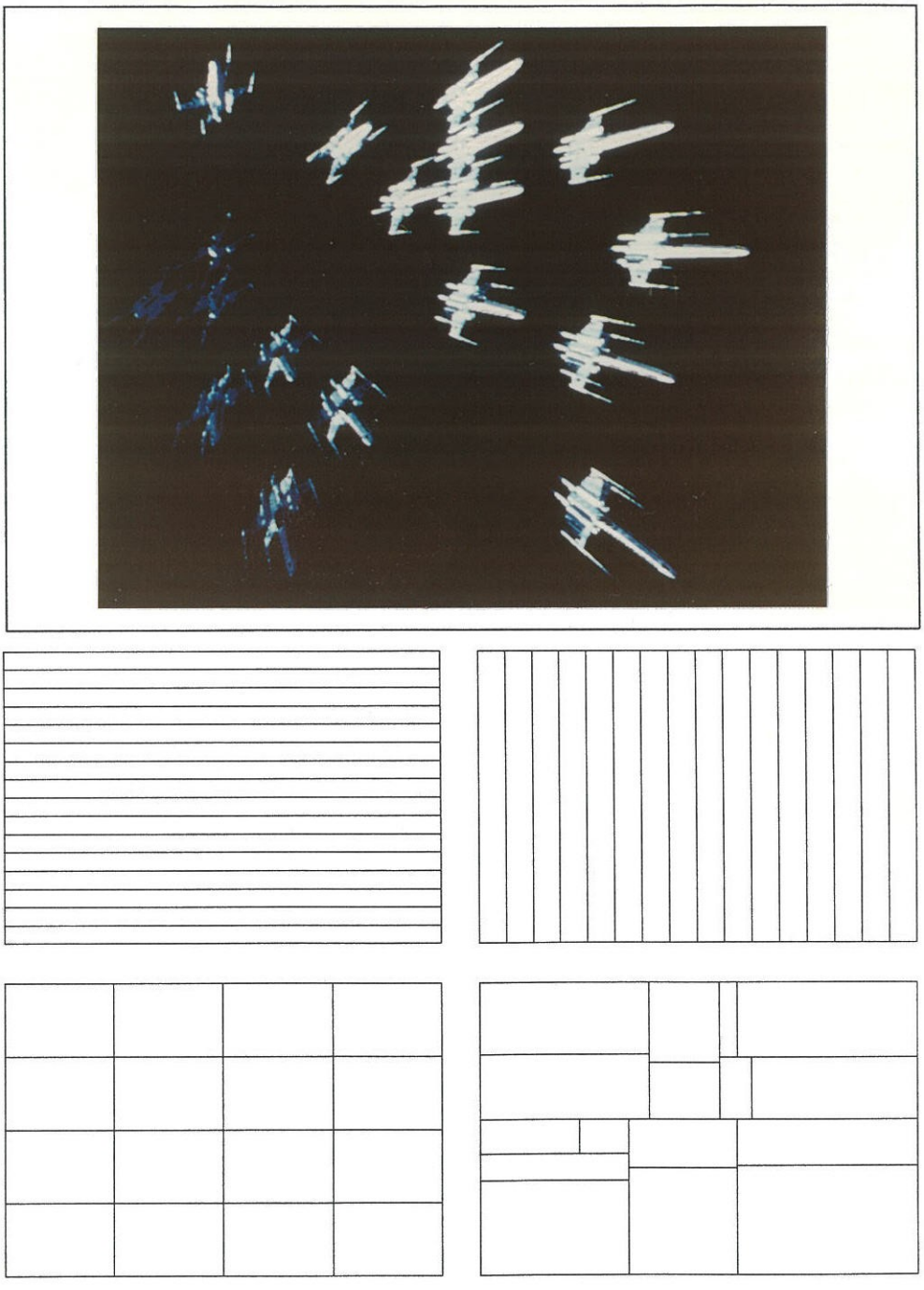
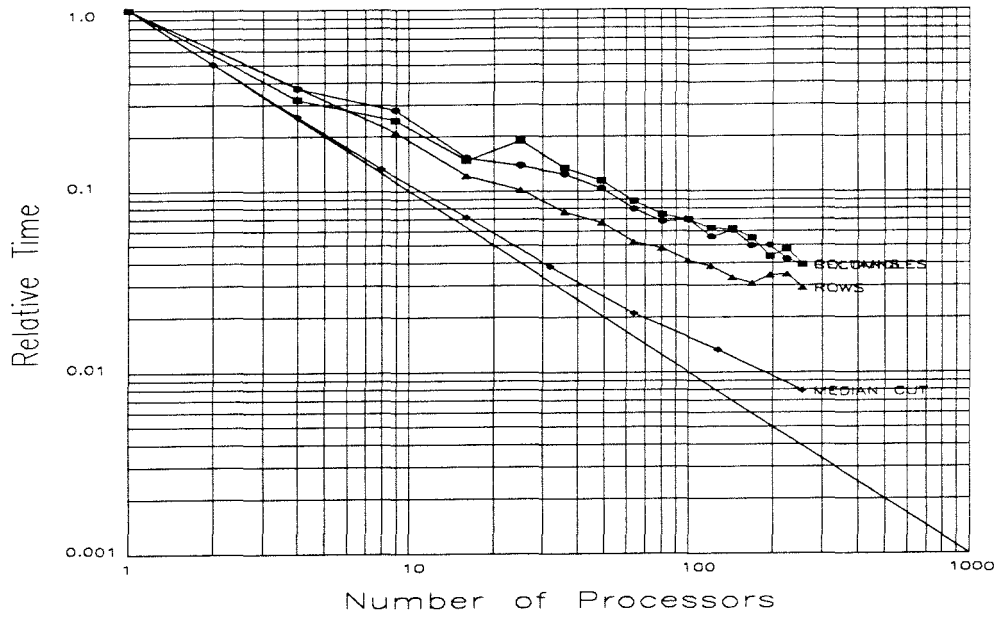


Figure A.3: X-Wing Fighter (xwf2)



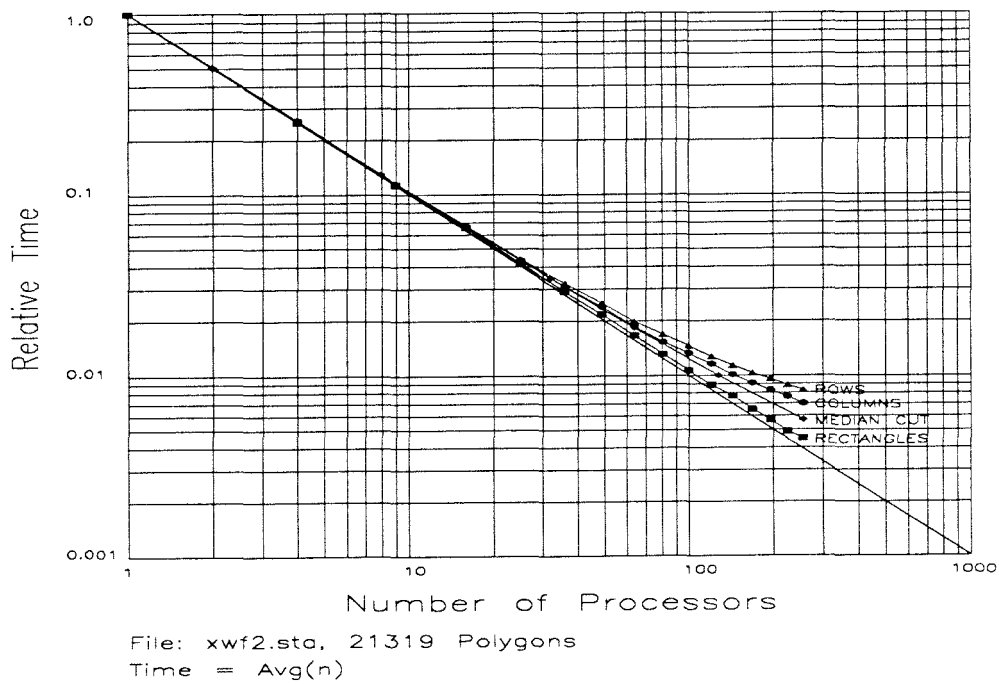


Figure A.3: X-Wing Fighter (xwf2) *(cont.)*

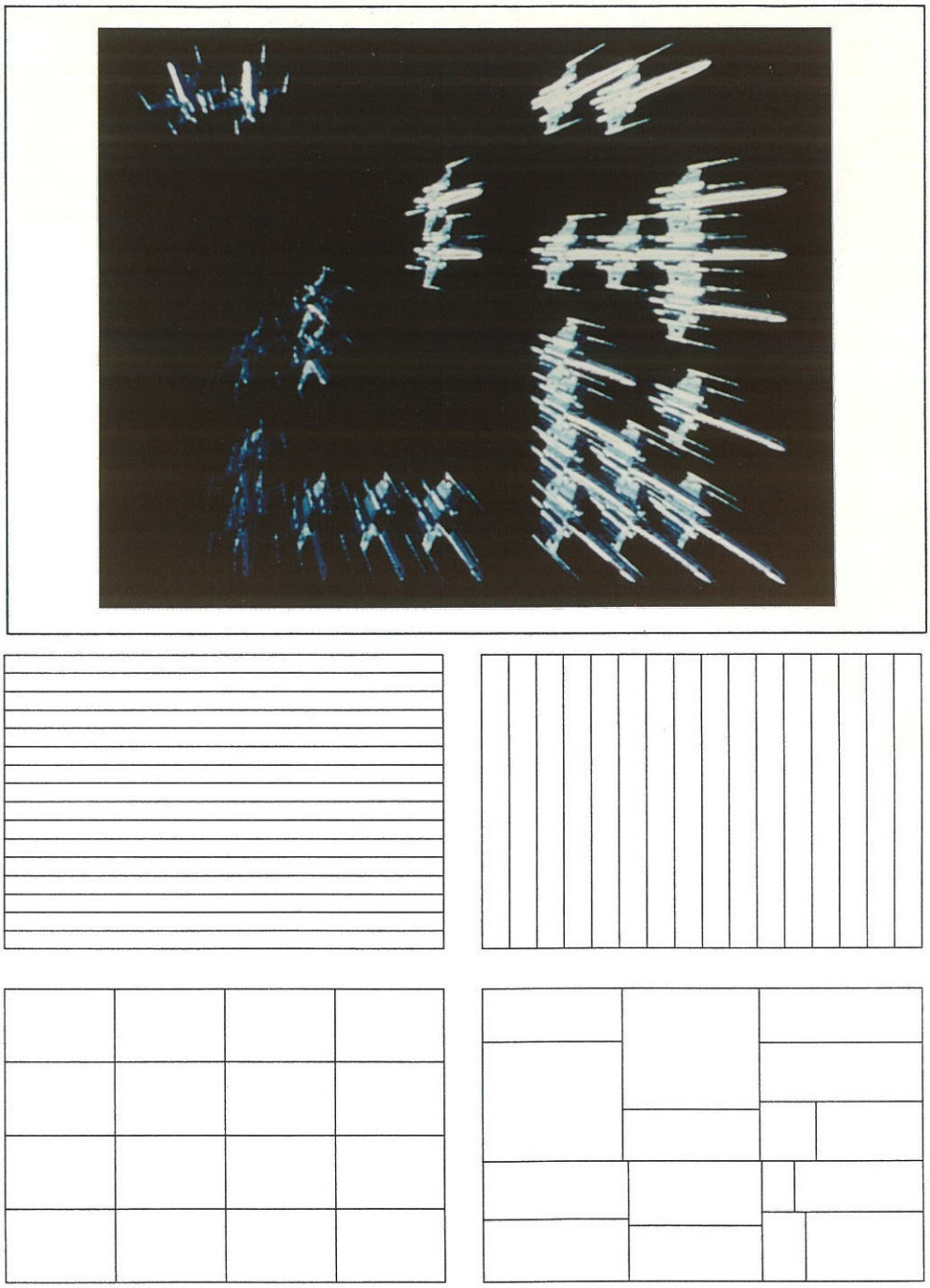
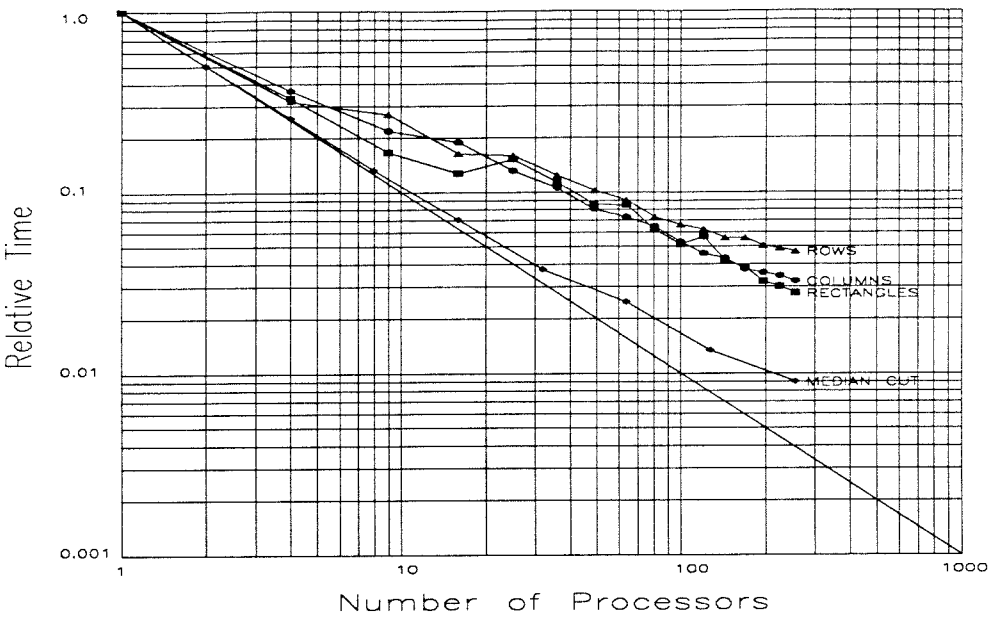


Figure A.4: X-Wing Fighter (xwf3)



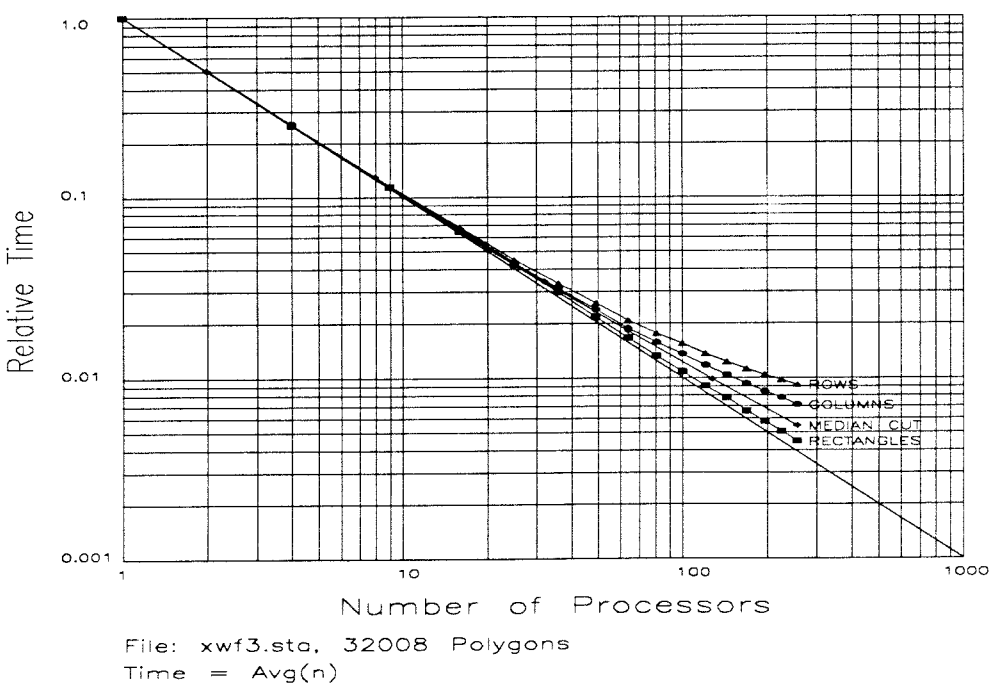


Figure A.4: X-Wing Fighter (xwf3) *(cont.)*

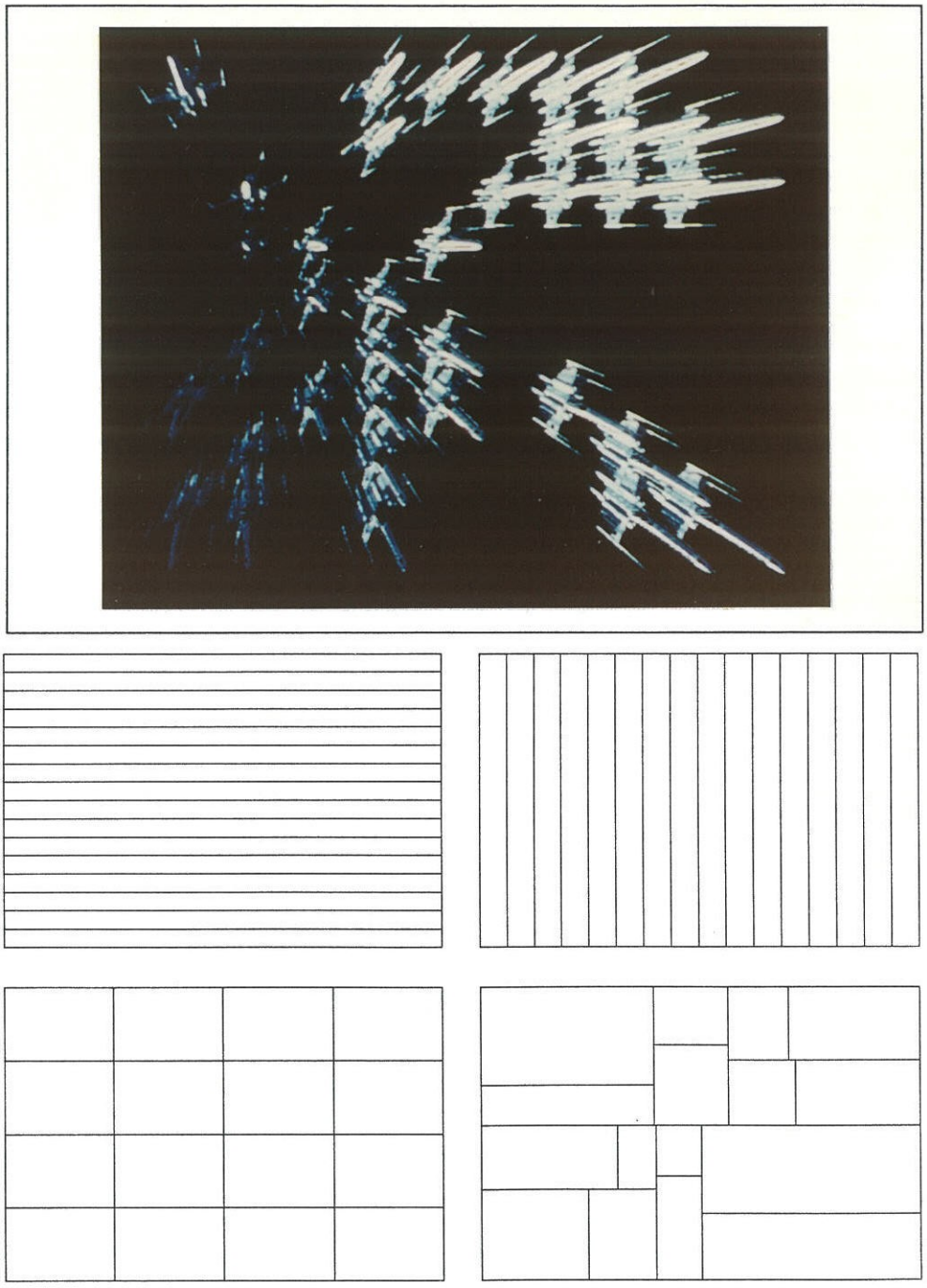
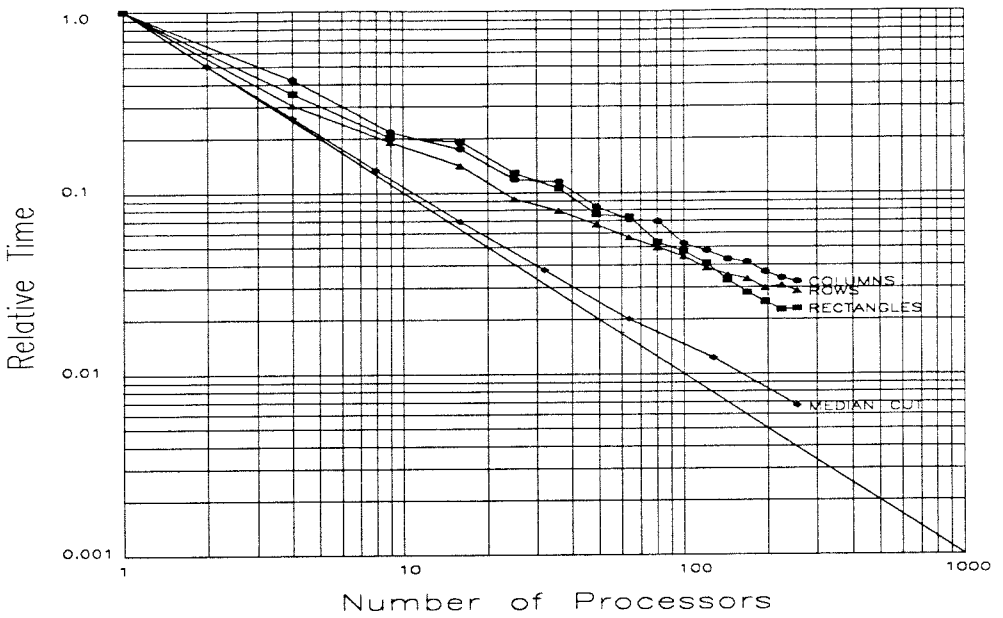
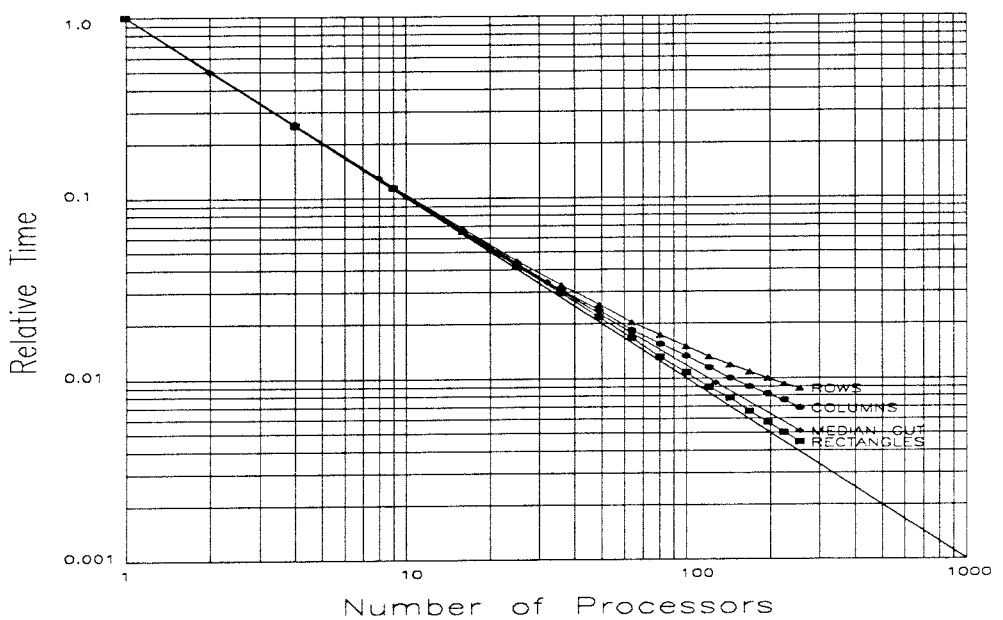


Figure A.5: X-Wing Fighter (xwf4)



File: xwf4.sta, 43146 Polygons
Time = Max(n)



File: xwf4.sta, 43146 Polygons
Time = Avg(n)

Figure A.5: X-Wing Fighter (xwf4) *(cont.)*

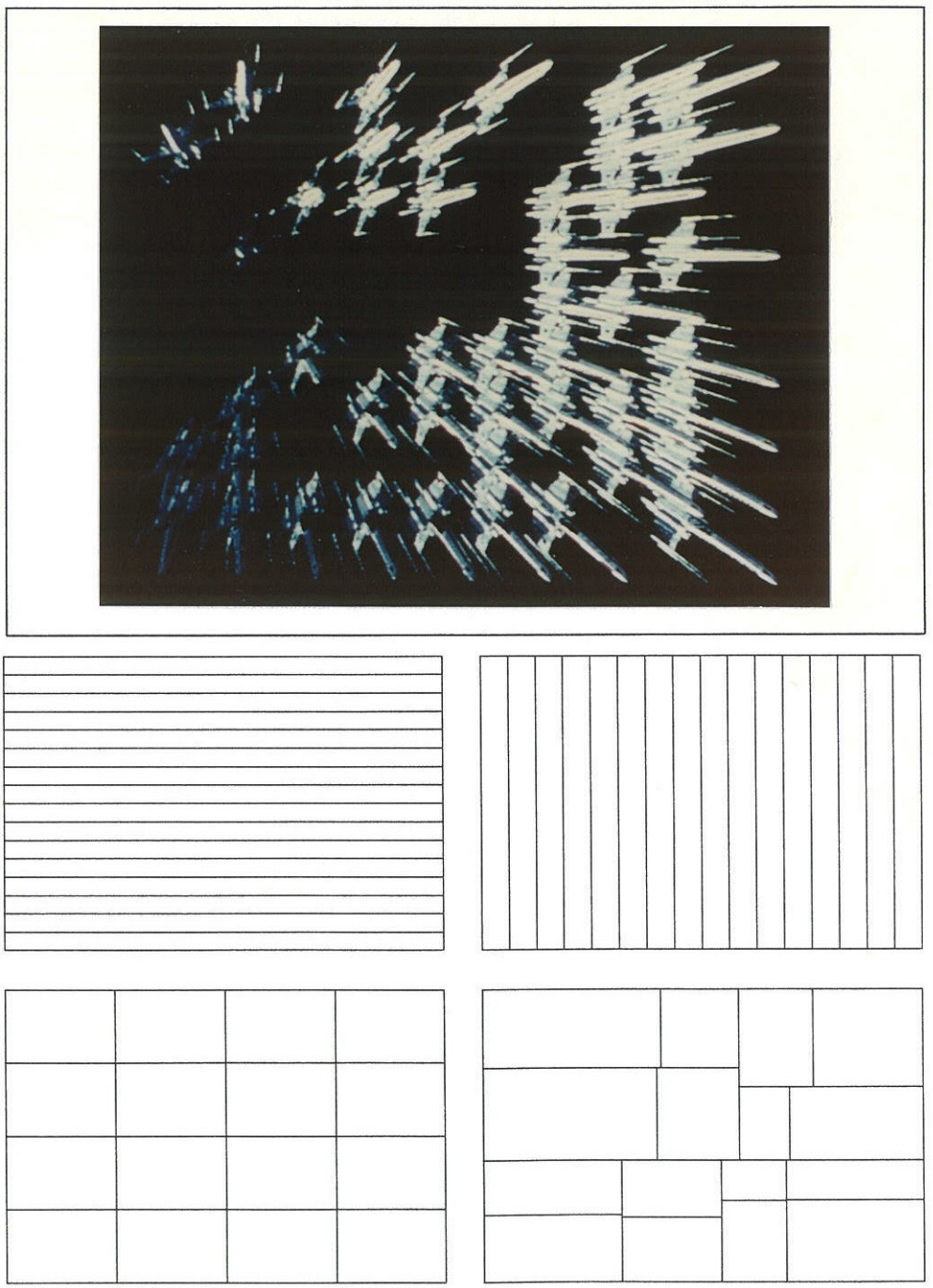
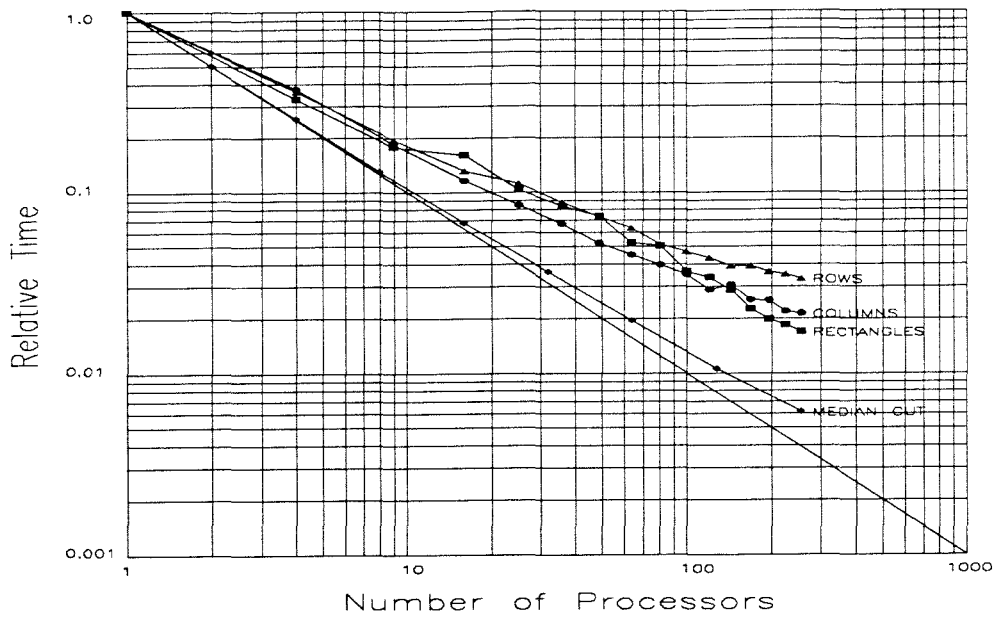


Figure A.6: X-Wing Fighter (xwf5)



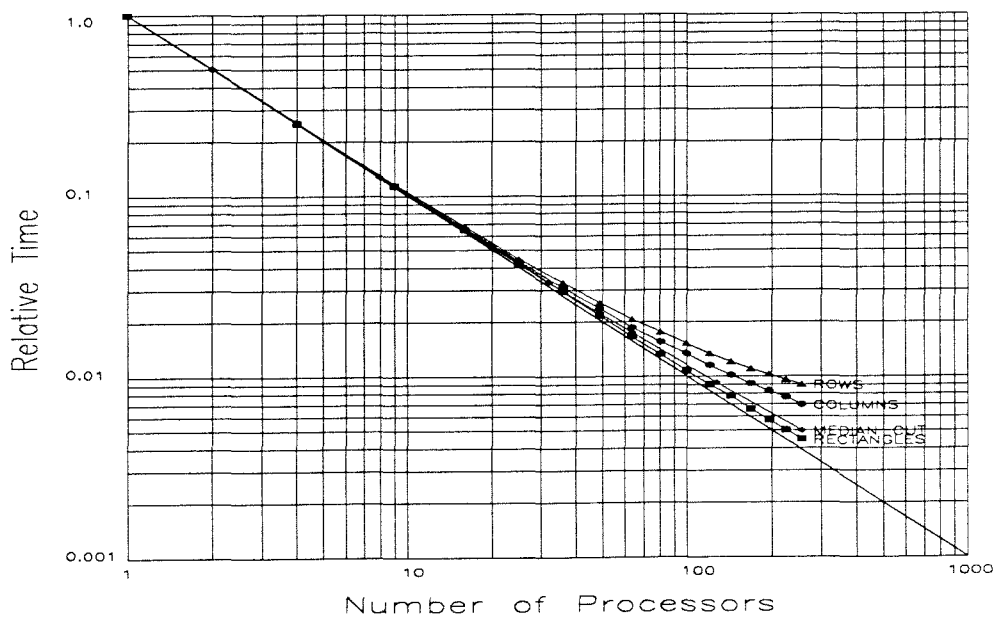


Figure A.6: X-Wing Fighter (xwf5) *(cont.)*

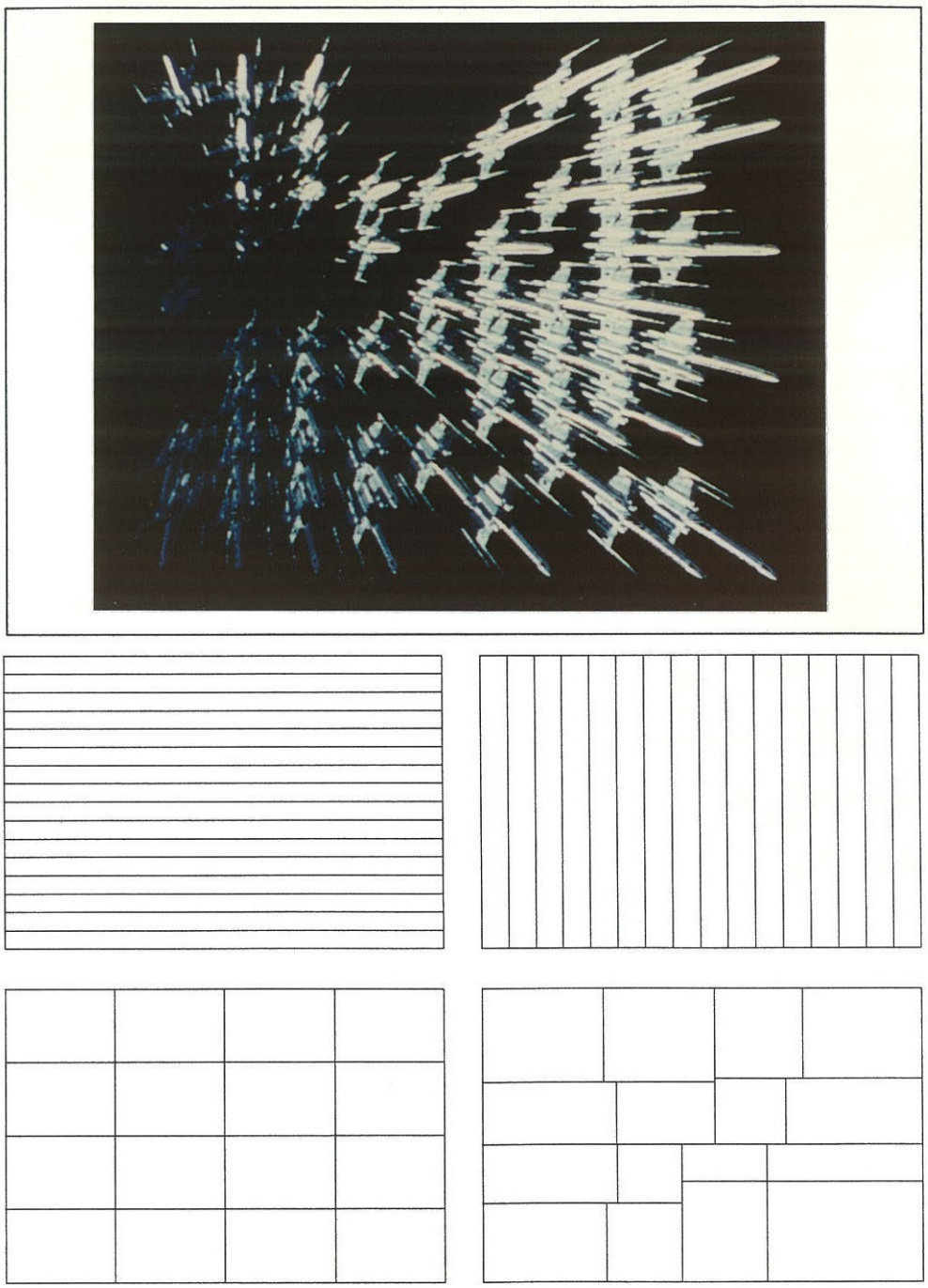
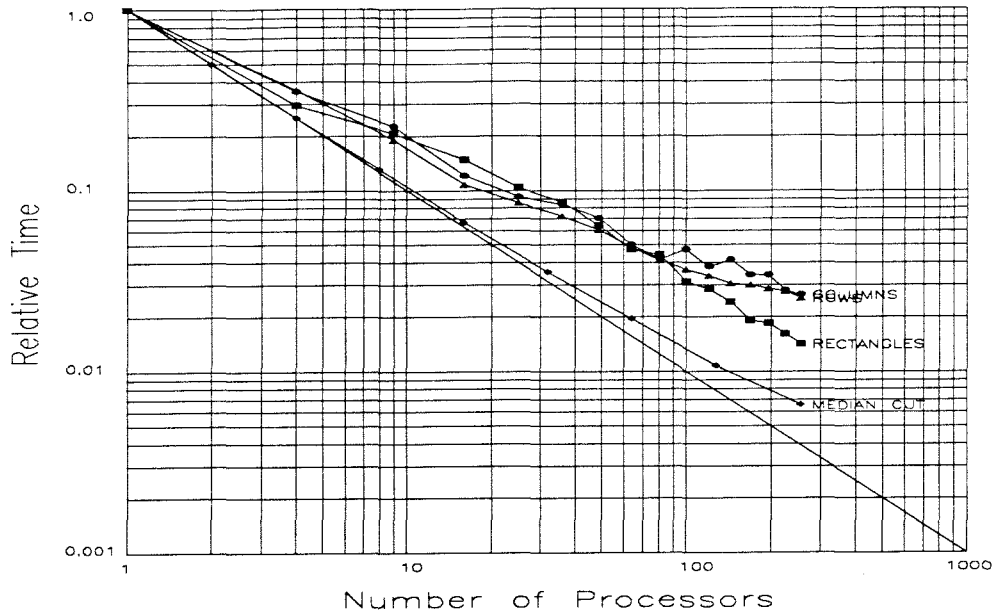


Figure A.7: X-Wing Fighter (xwf6)



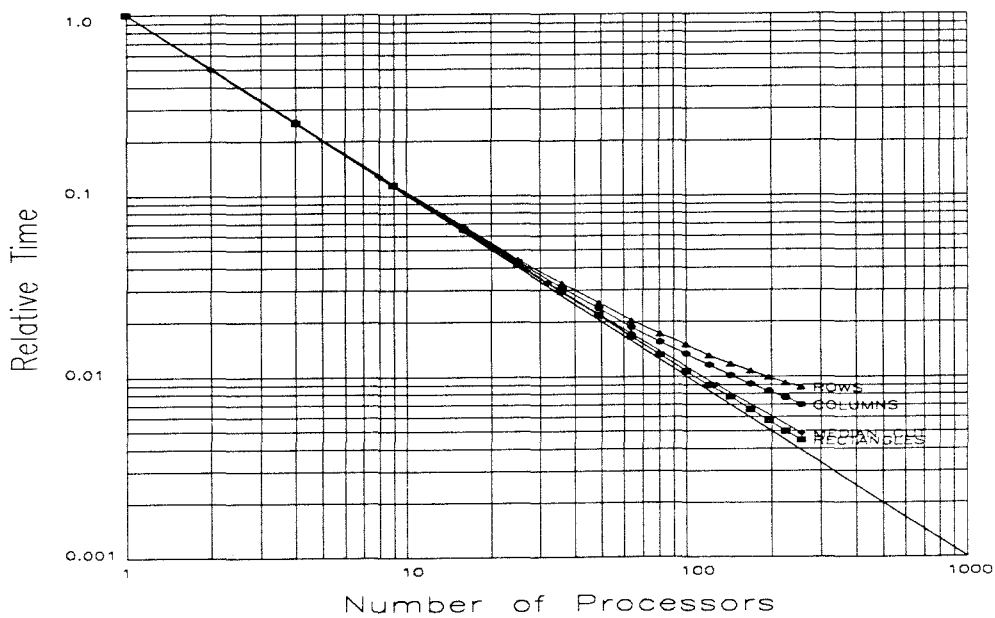


Figure A.7: X-Wing Fighter (xwf6) *(cont.)*

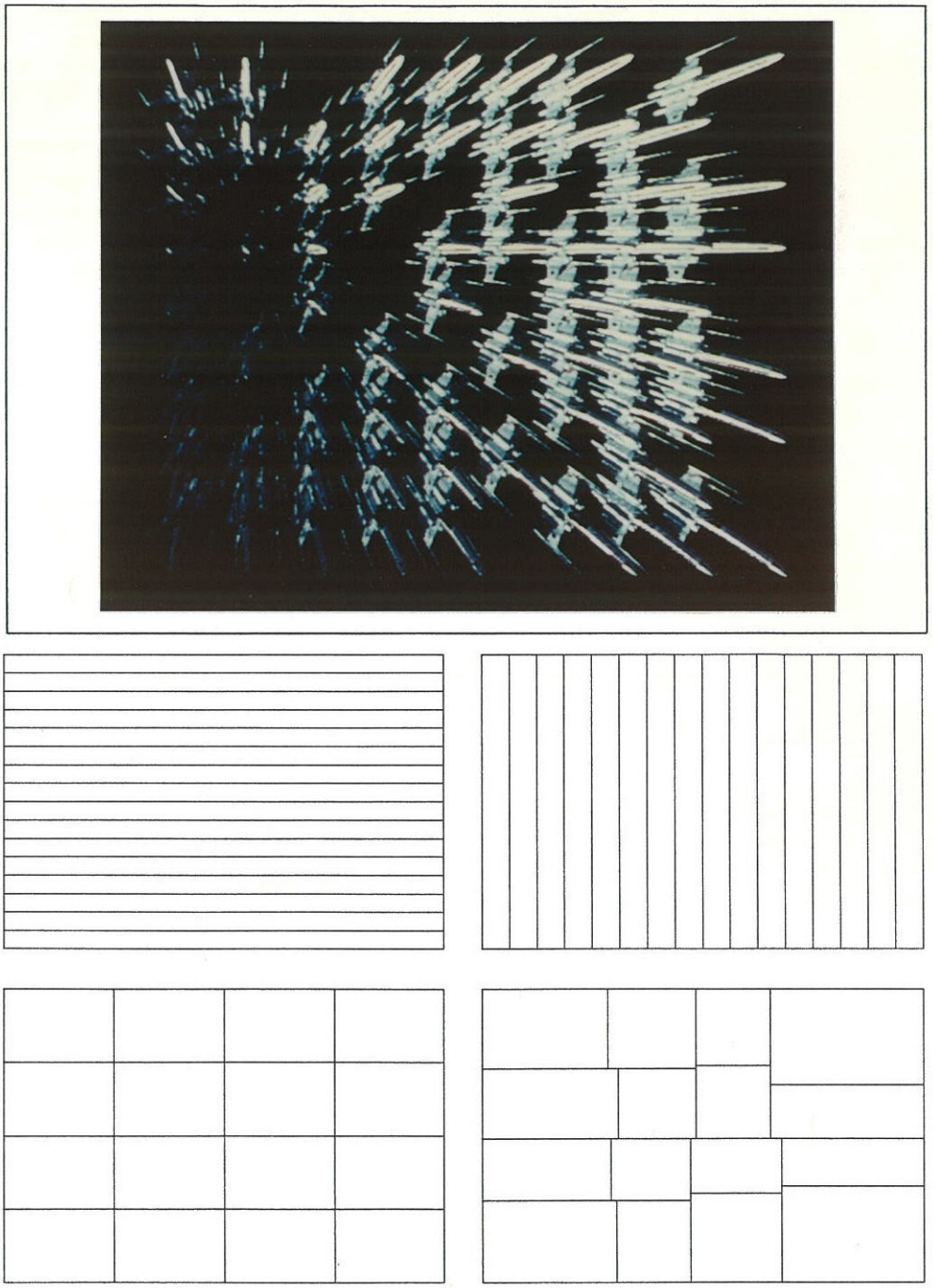
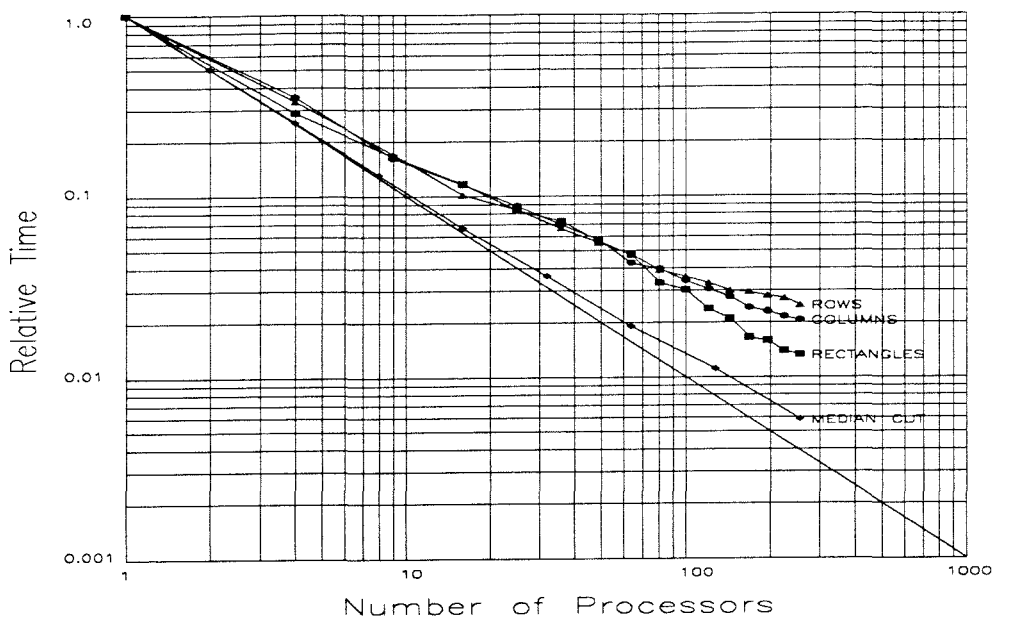


Figure A.8: X-Wing Fighter (xwf7)



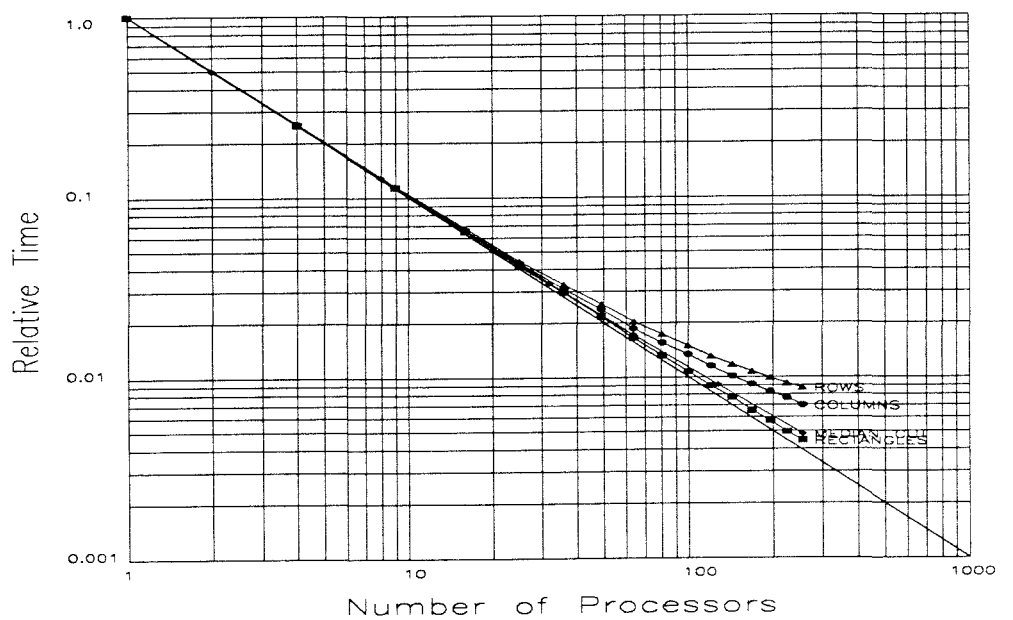


Figure A.8: X-Wing Fighter (xwf7) *(cont.)*

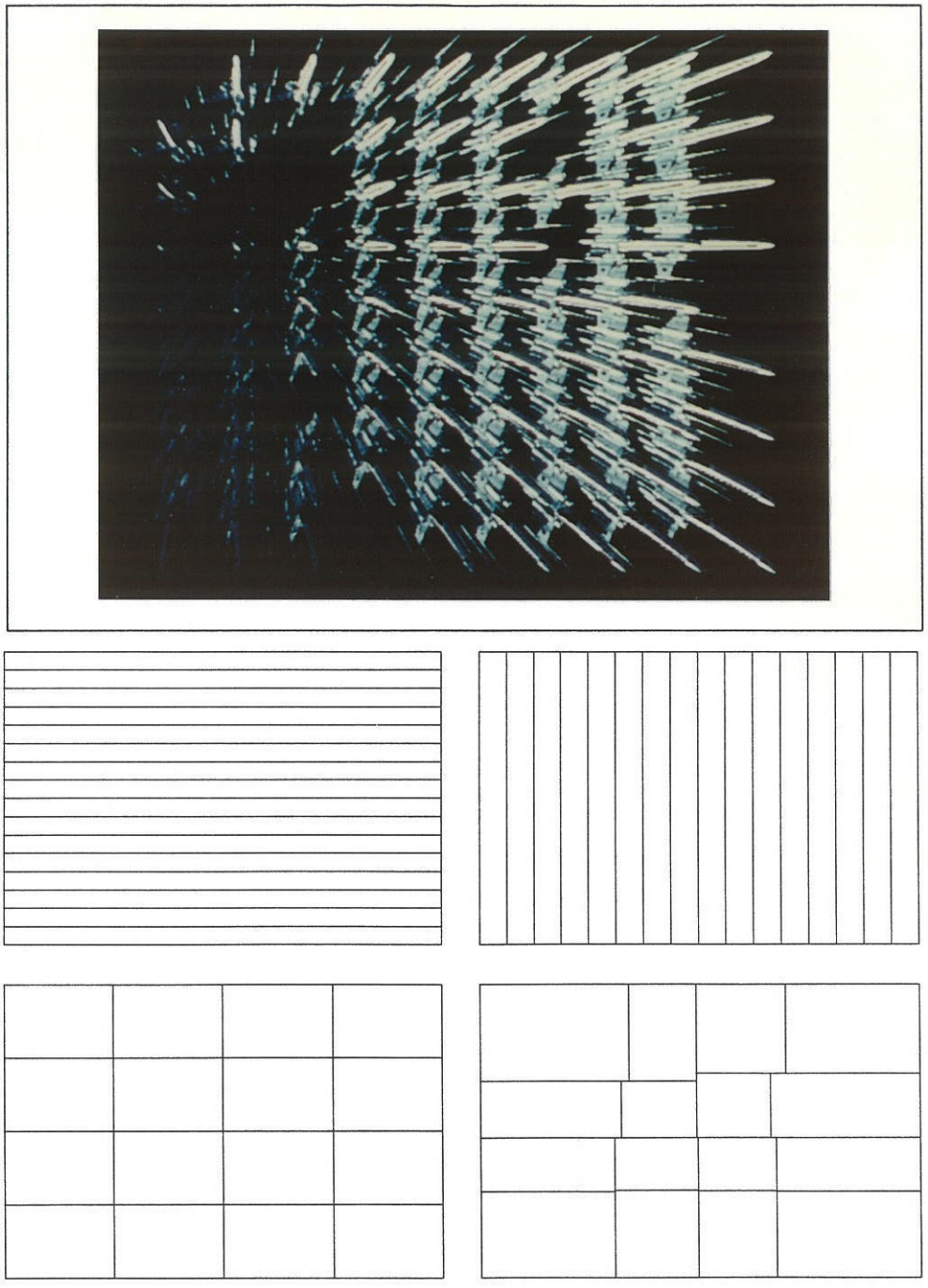
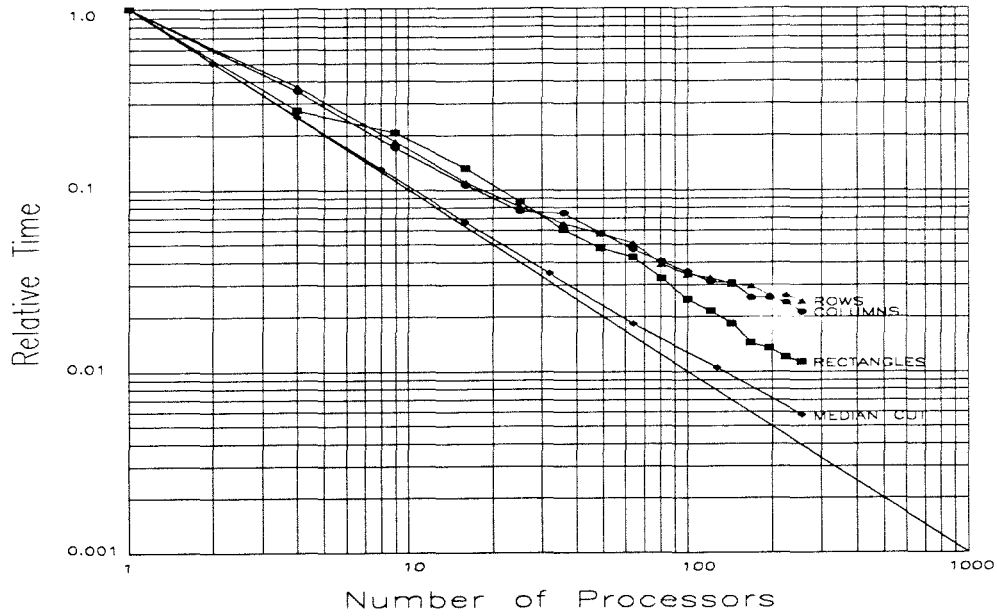
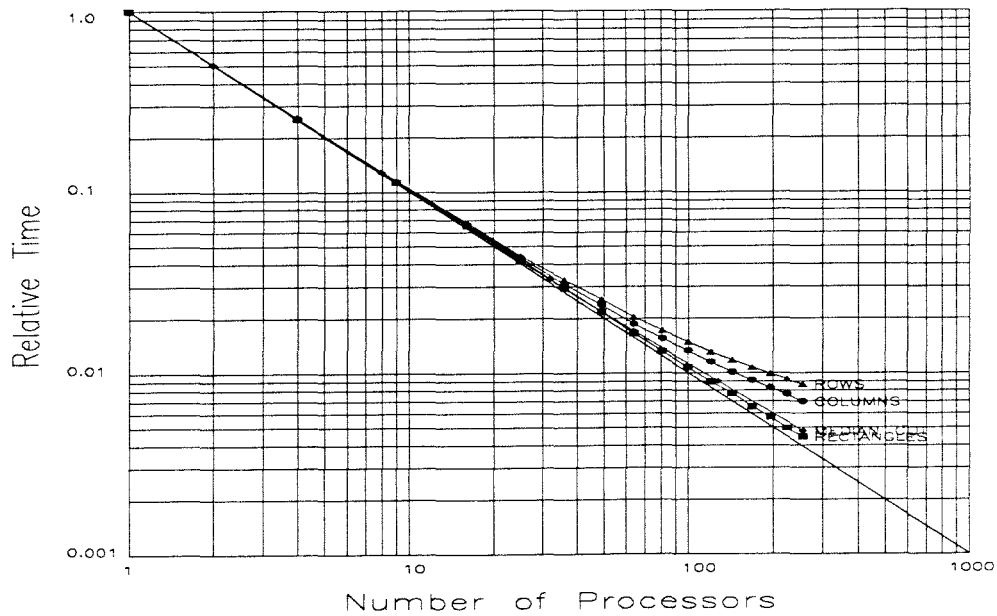


Figure A.9: X-Wing Fighter (xwf8)



File: xwf8.sta, 85464 Polygons
Time = Max(n)



File: xwf8.sta, 85464 Polygons
Time = Avg(n)

Figure A.9: X-Wing Fighter (xwf8) *(cont.)*

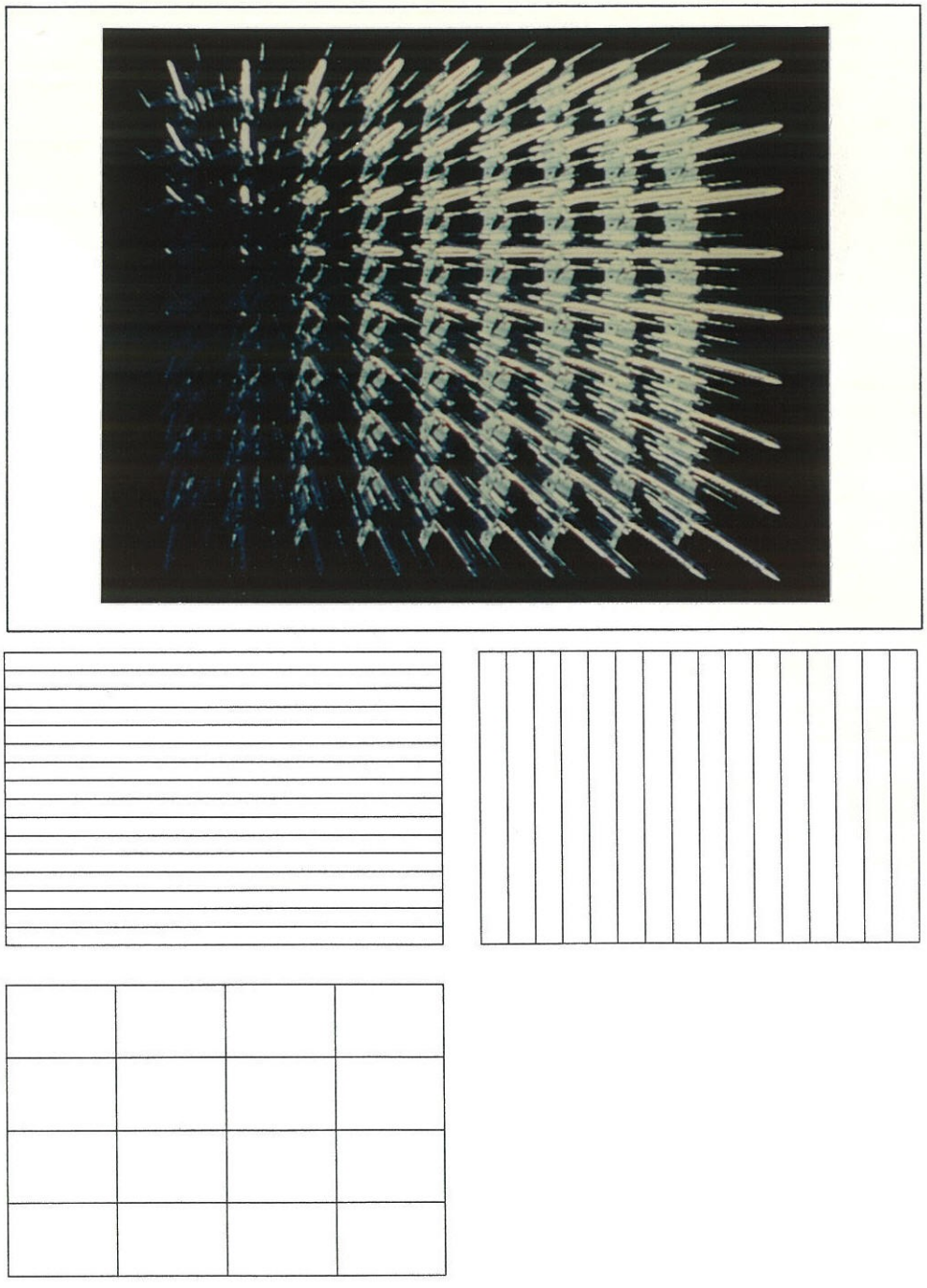
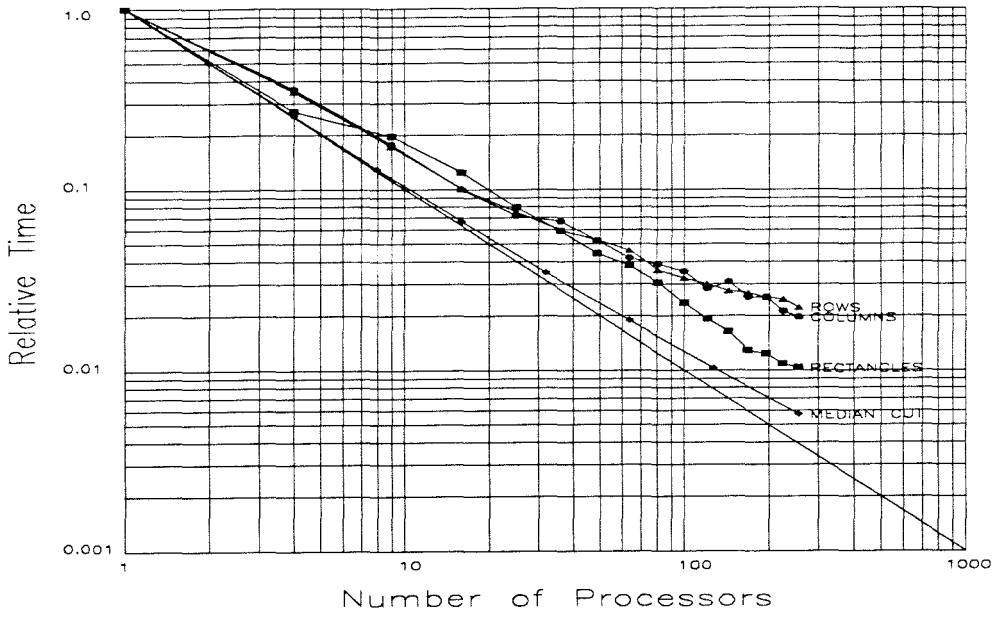
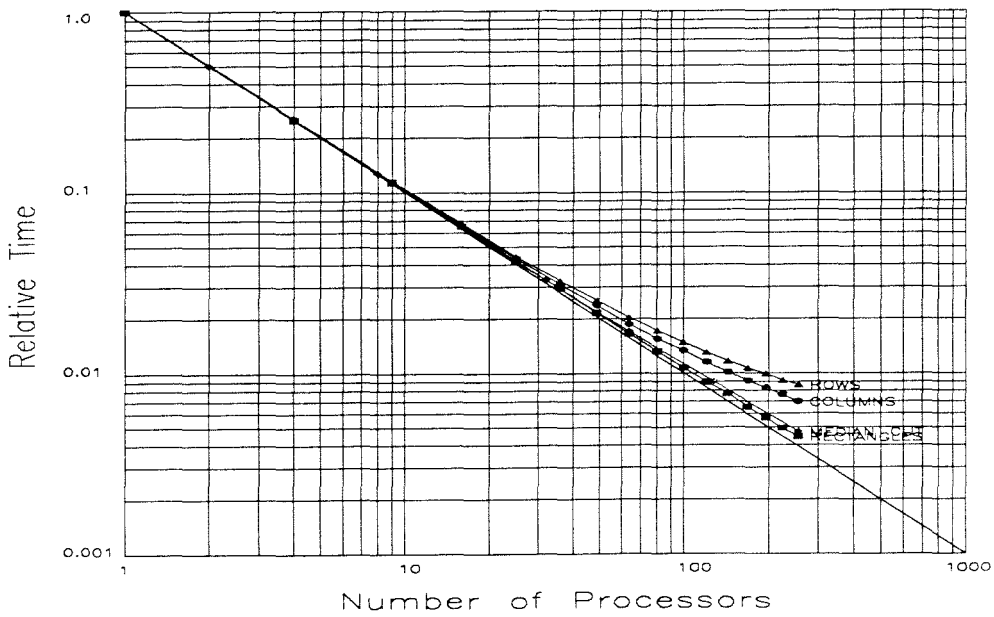


Figure A.10: X-Wing Fighter (xwf9)



File: xwf9.sta, 96065 Polygons
Time = Max(n)



File: xwf9.sta, 96065 Polygons
Time = Avg(n)

Figure A.10: X-Wing Fighter (xwf9) *(cont.)*

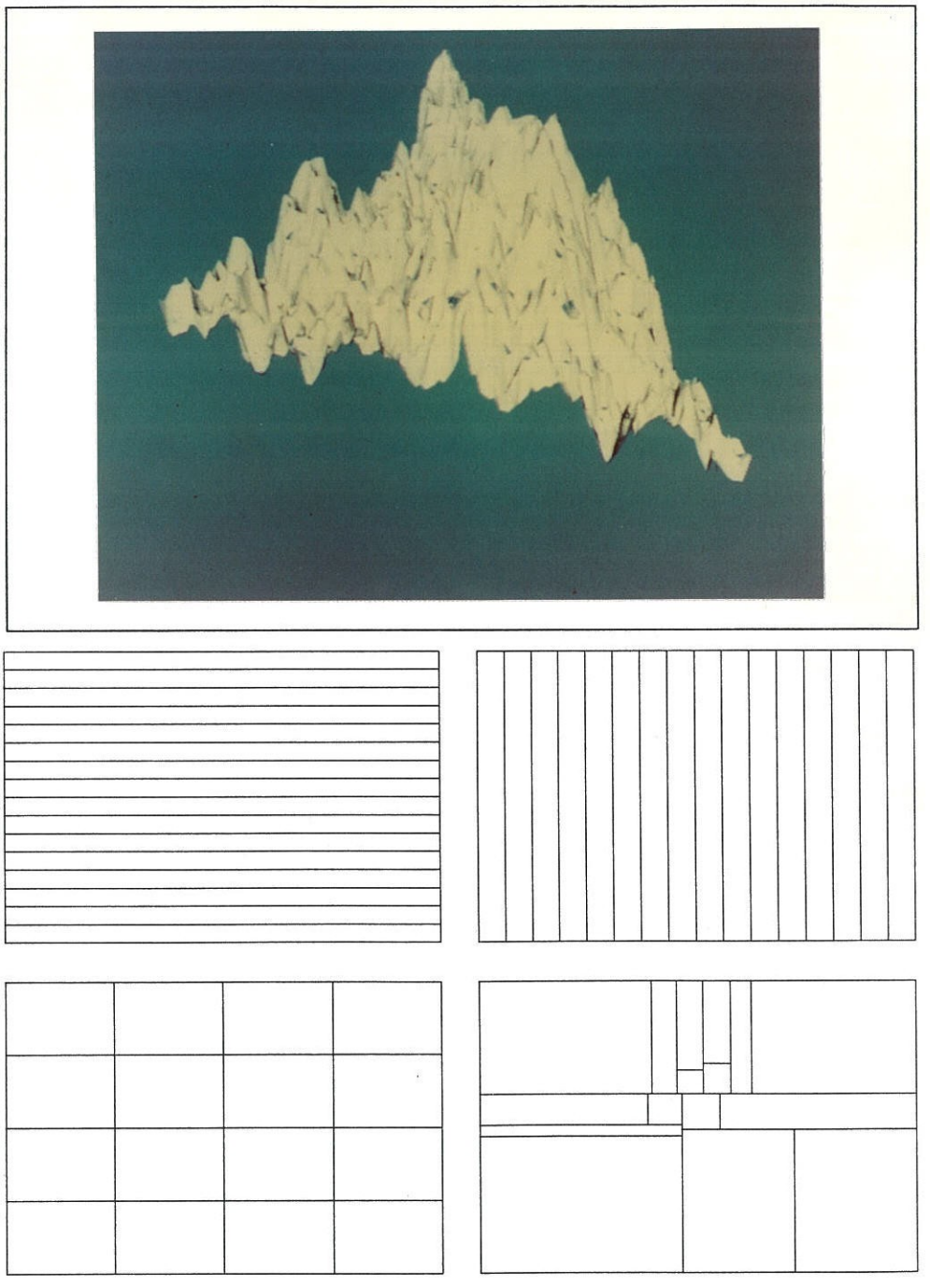
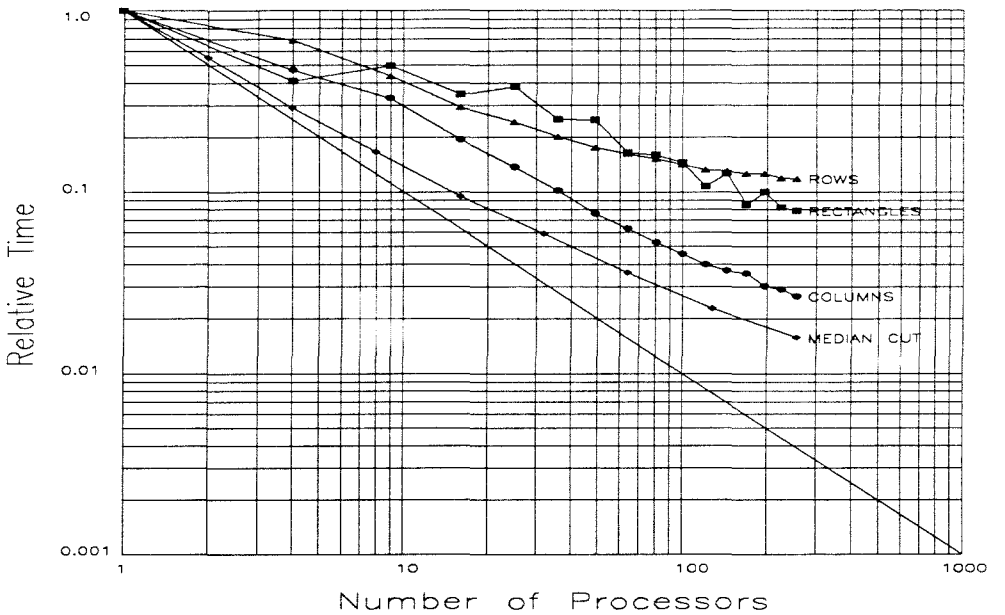


Figure A.11: Fractal Landscape (frac8)



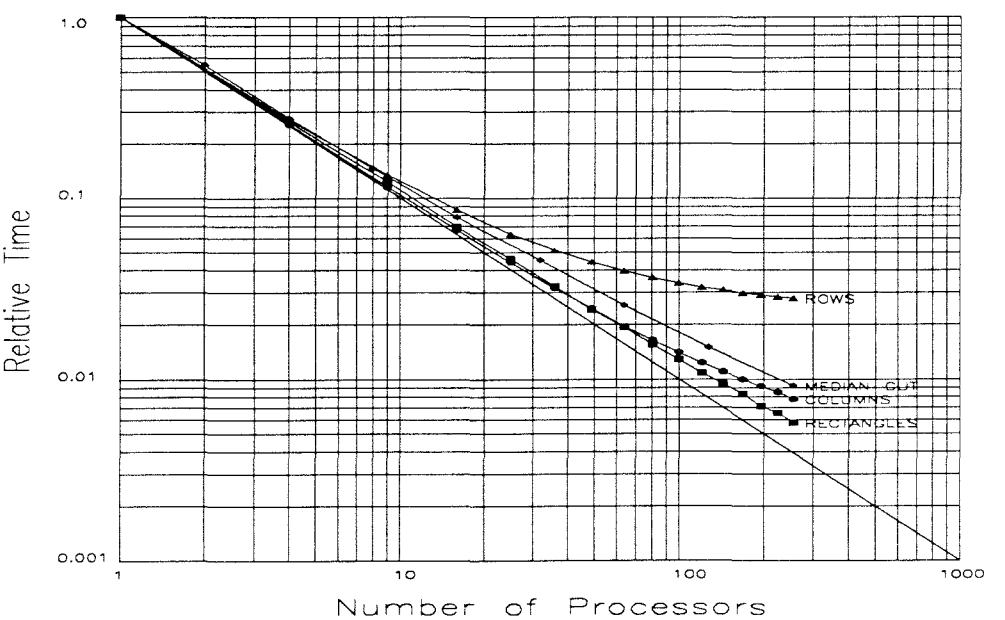


Figure A.11: Fractal Landscape (frac8) *(cont.)*

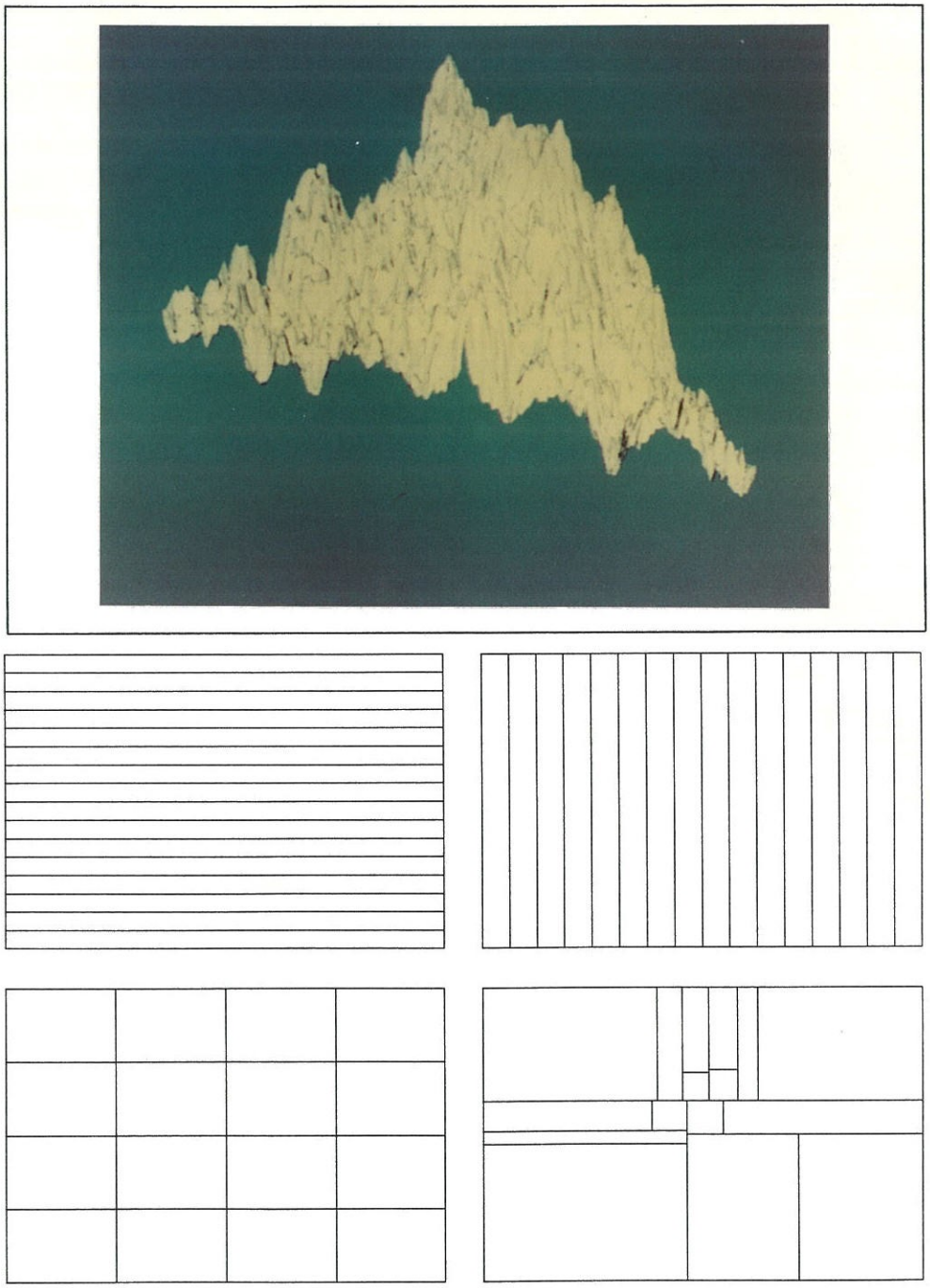
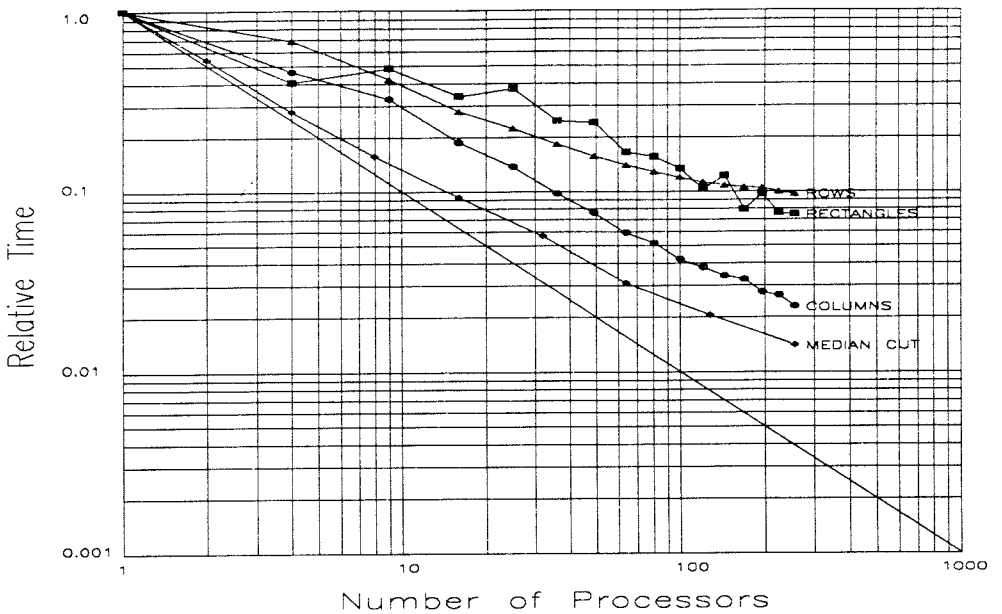


Figure A.12: Fractal Landscape (frac16)



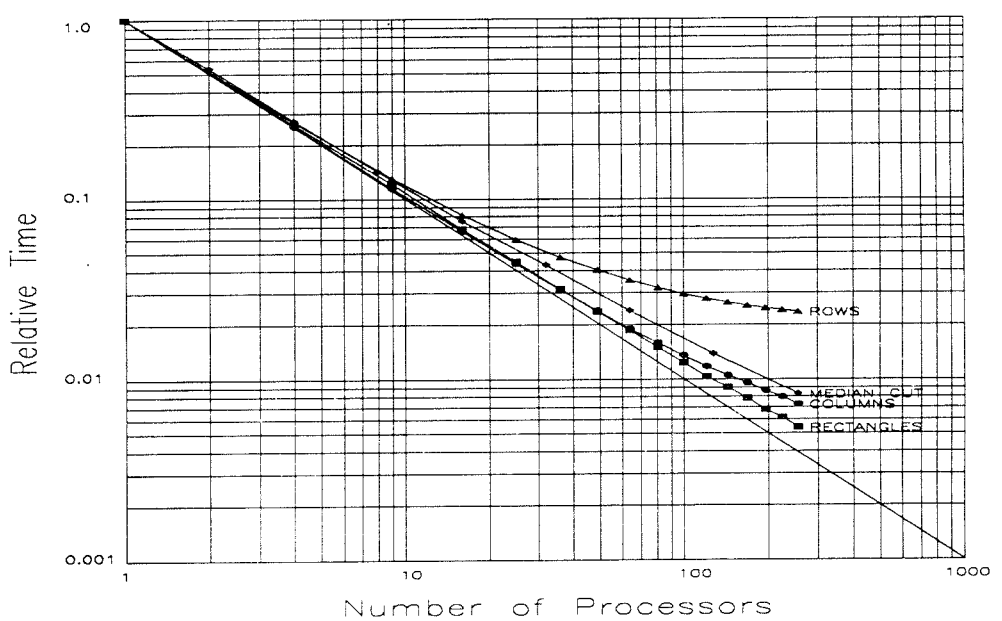


Figure A.12: Fractal Landscape (frac16) *(cont.)*

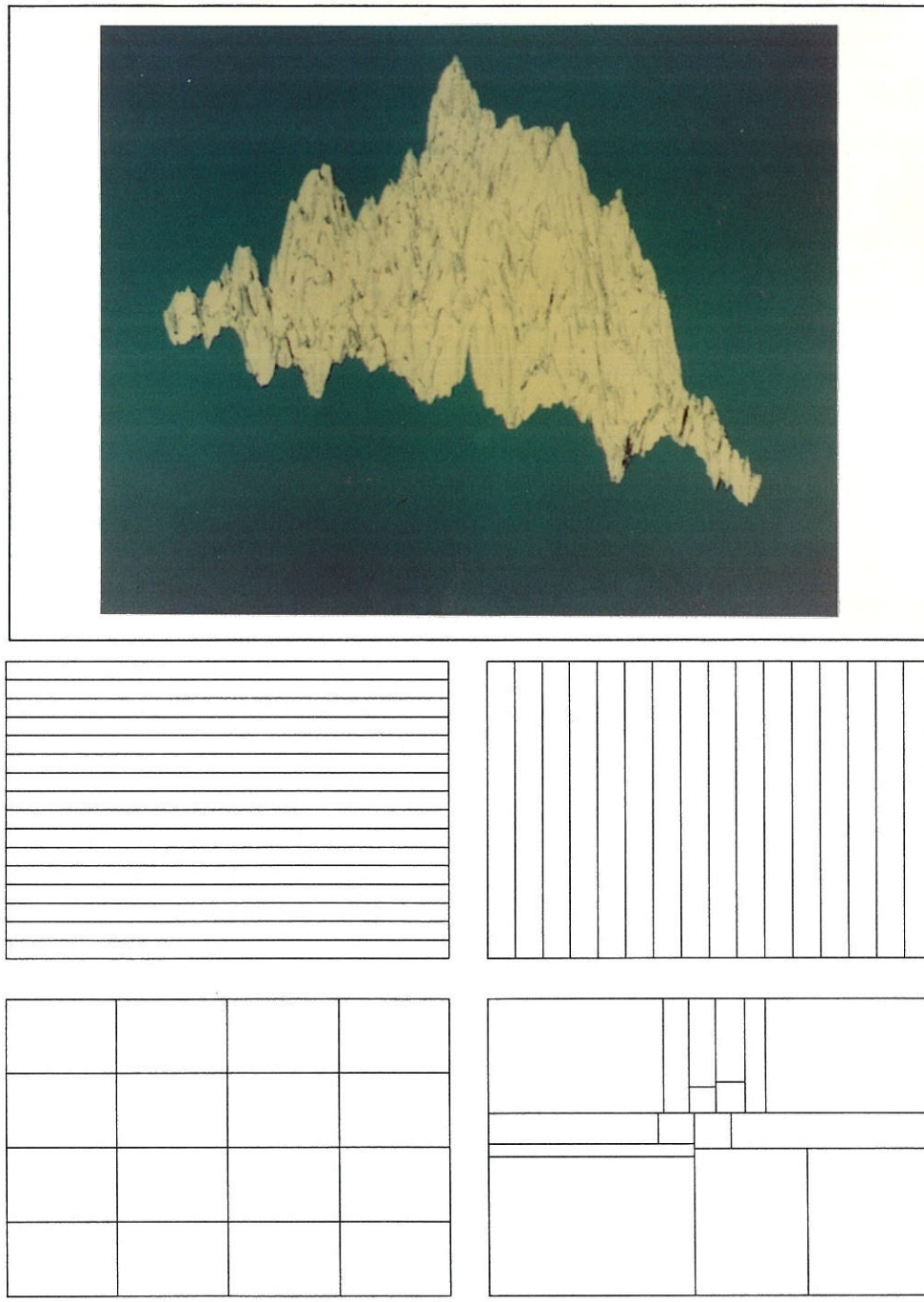
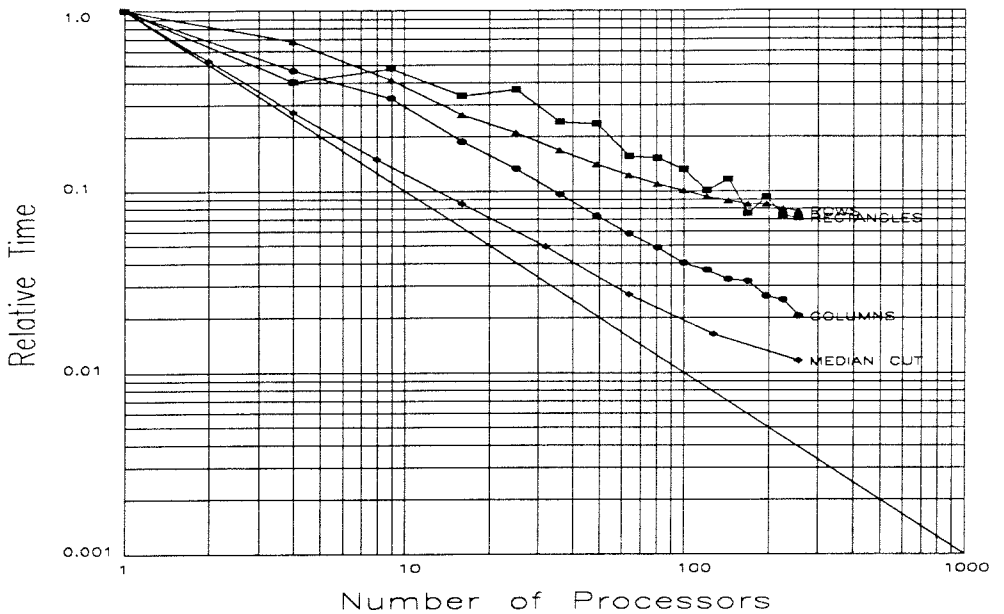
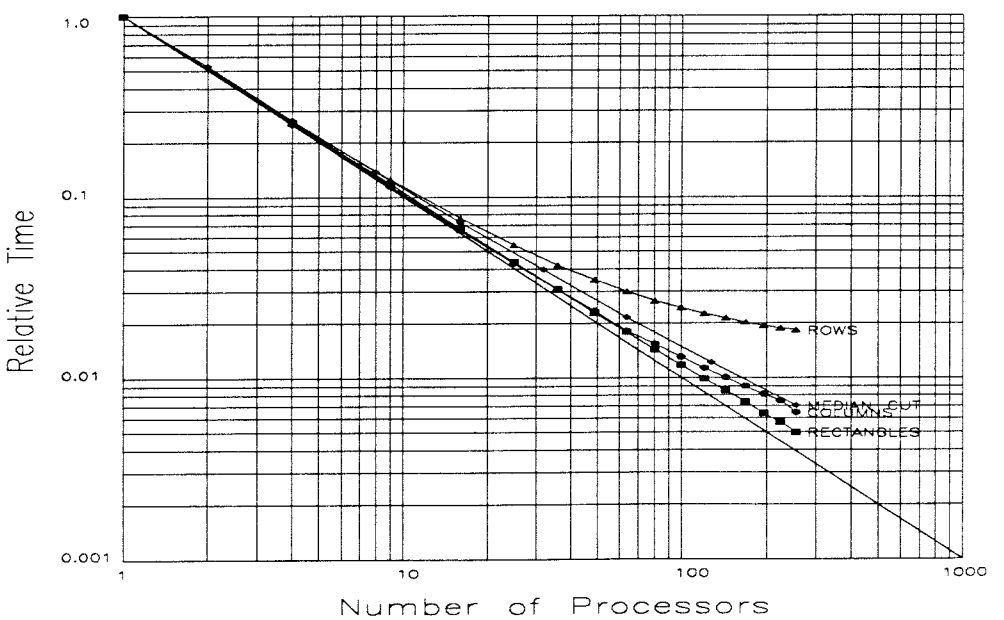


Figure A.13: Fractal Landscape (frac32)



File: frac32.sta, 32570 Polygons
Time = Max(n)



File: frac32.sta, 32570 Polygons
Time = Avg(n)

Figure A.13: Fractal Landscape (frac32) *(cont.)*

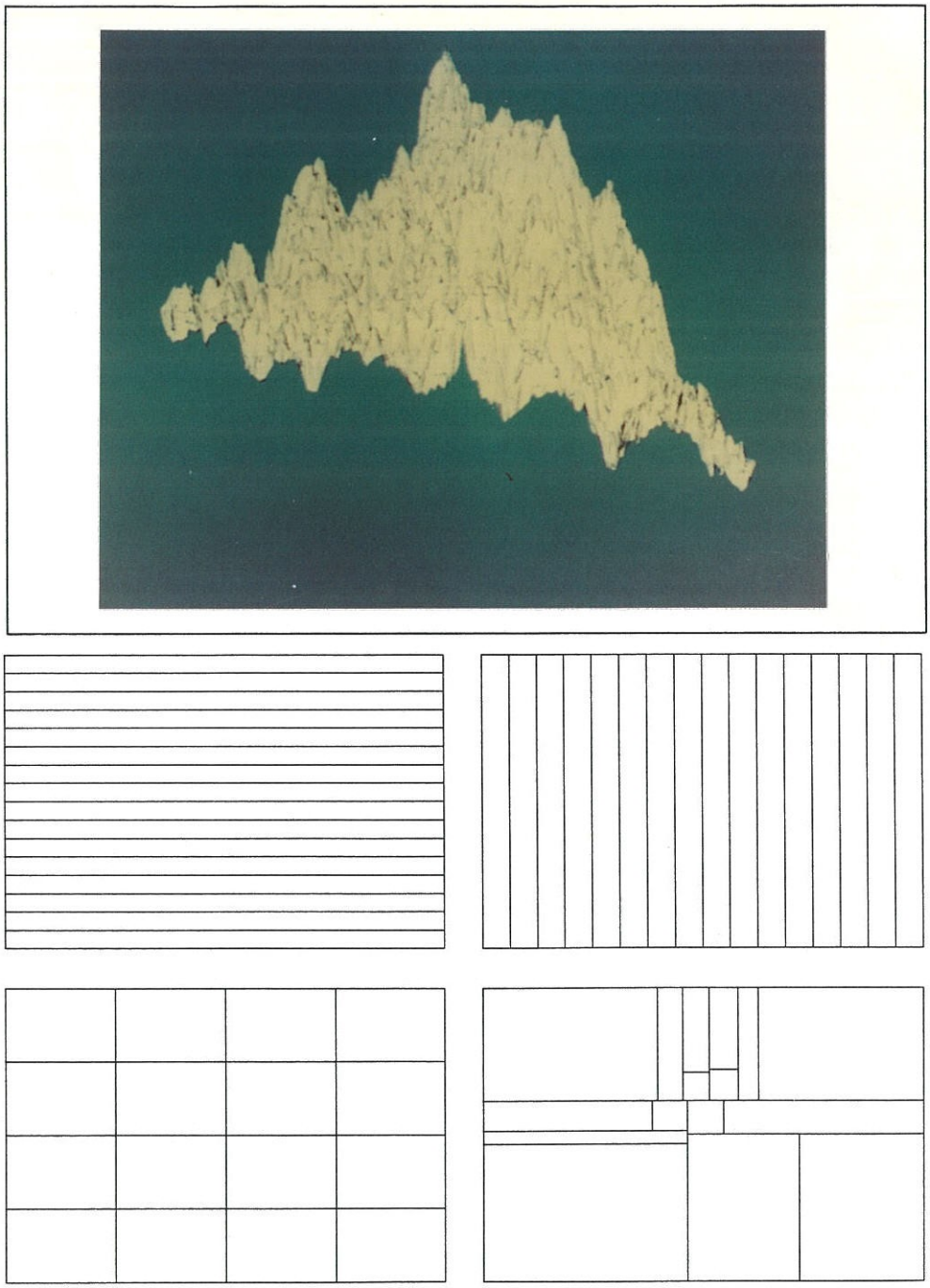
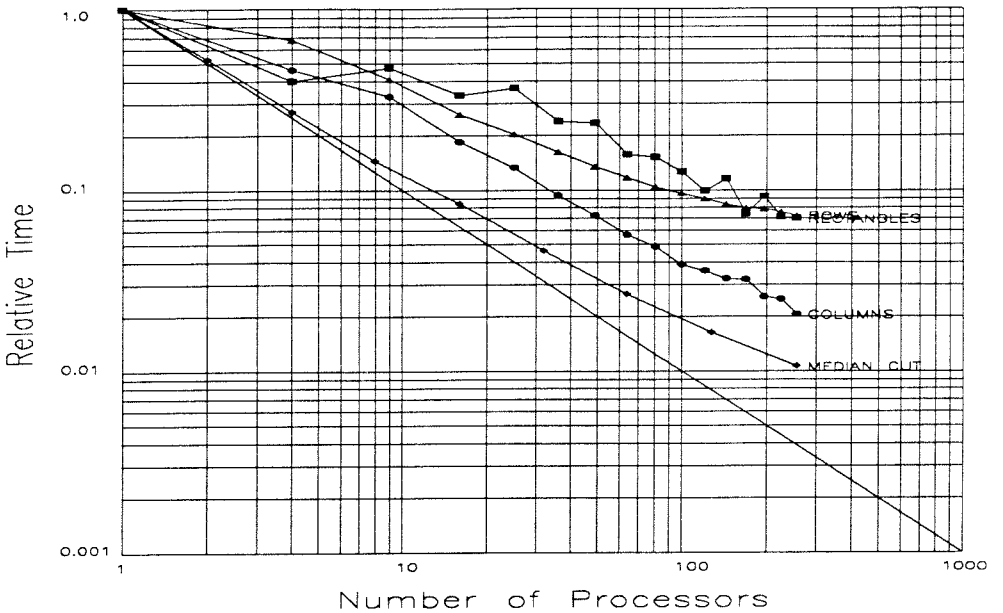


Figure A.14: Fractal Landscape (frac40)



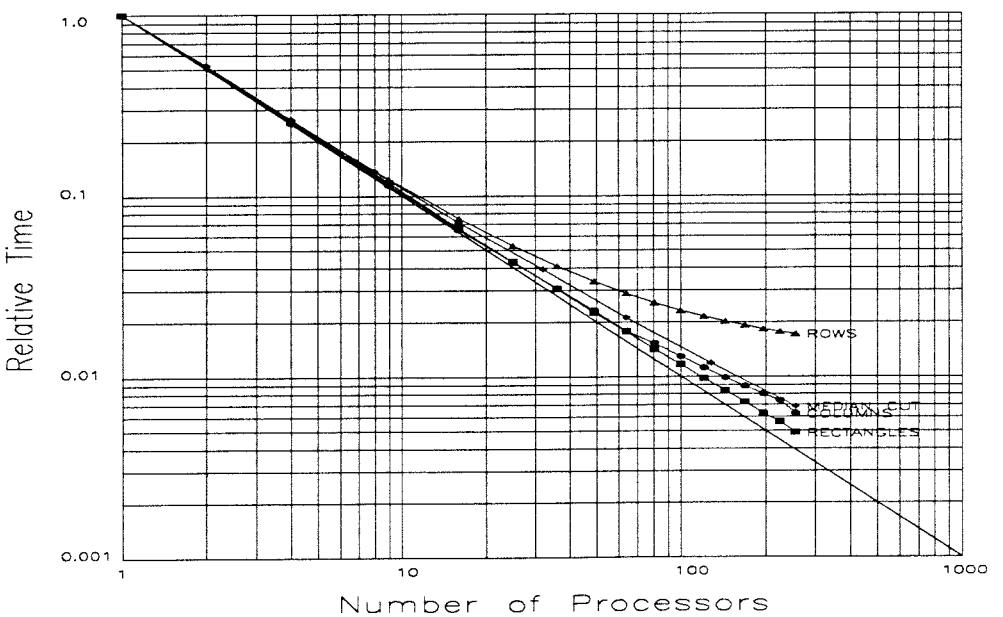


Figure A.14: Fractal Landscape (frac40) *(cont.)*

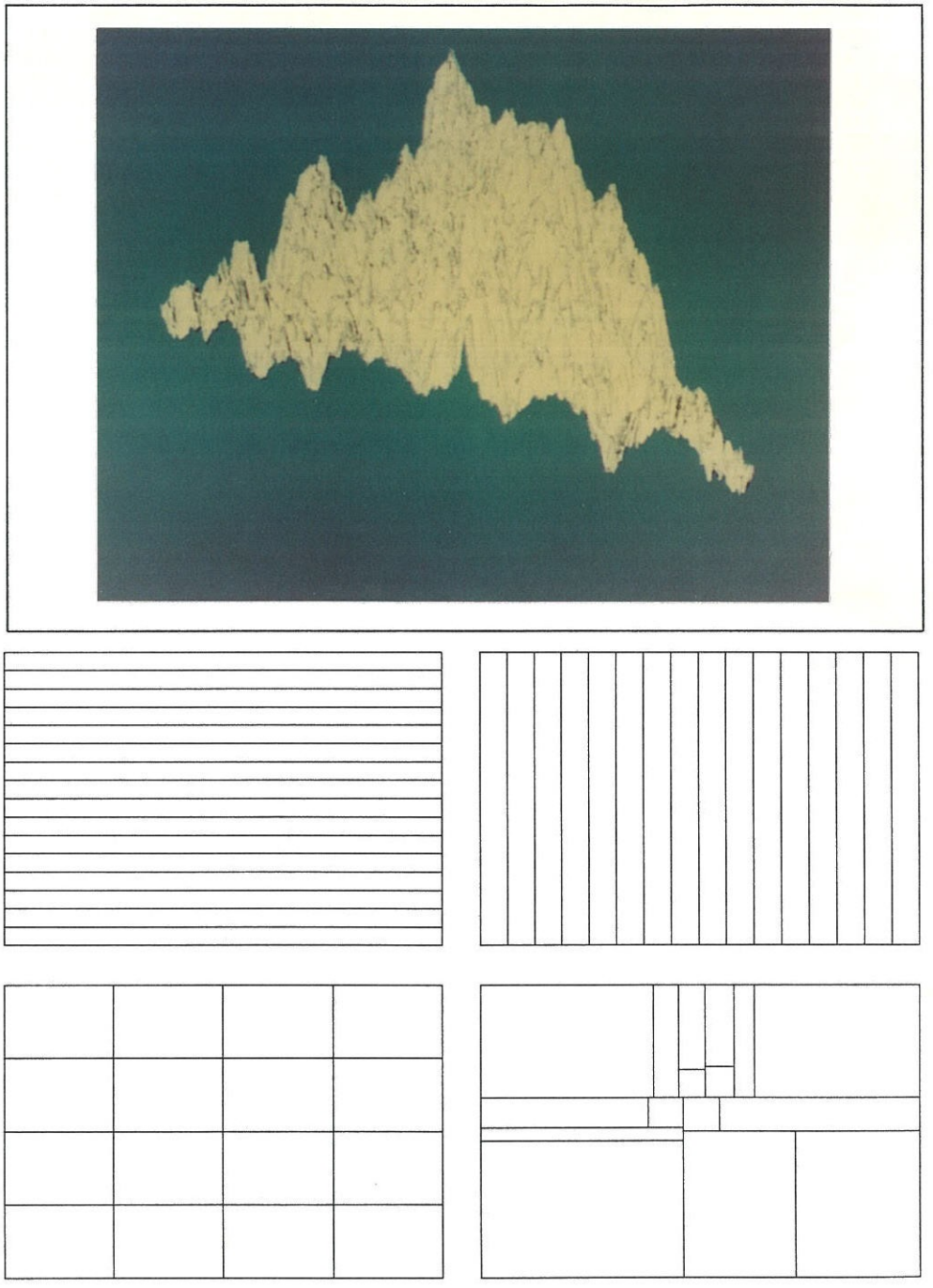
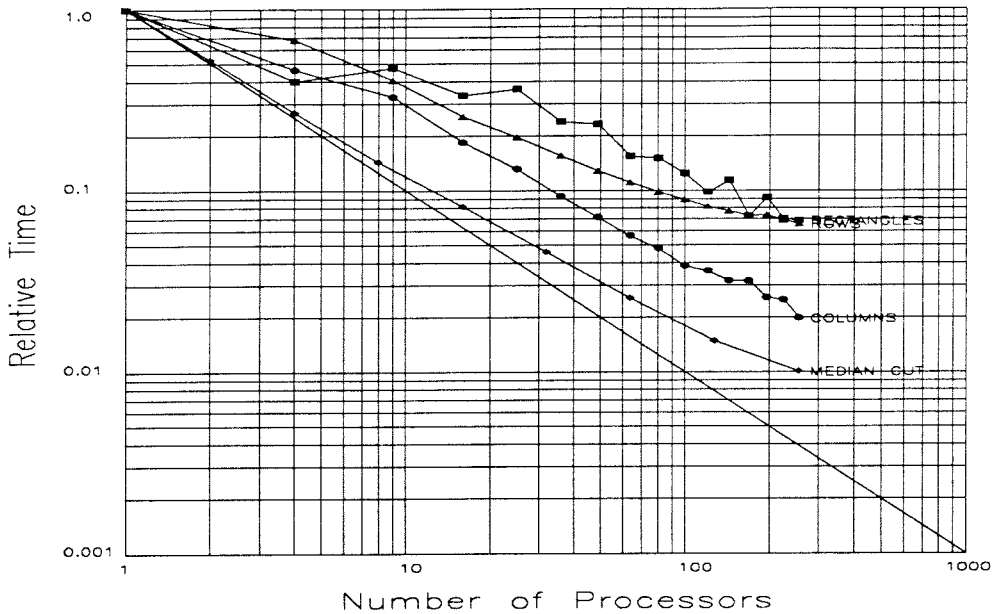


Figure A.15: Fractal Landscape (frac64)



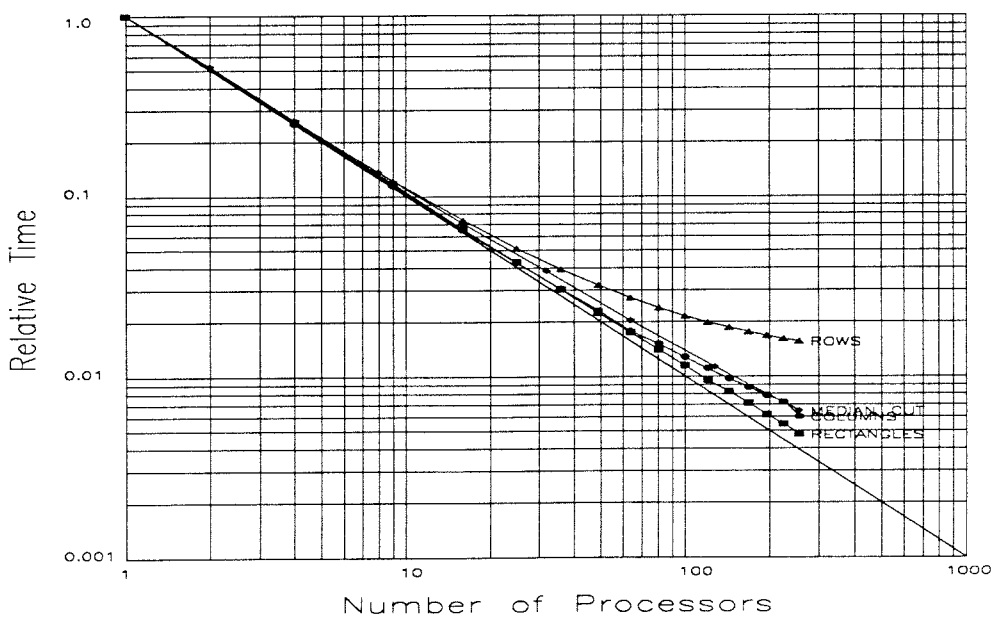


Figure A.15: Fractal Landscape (frac64) *(cont.)*

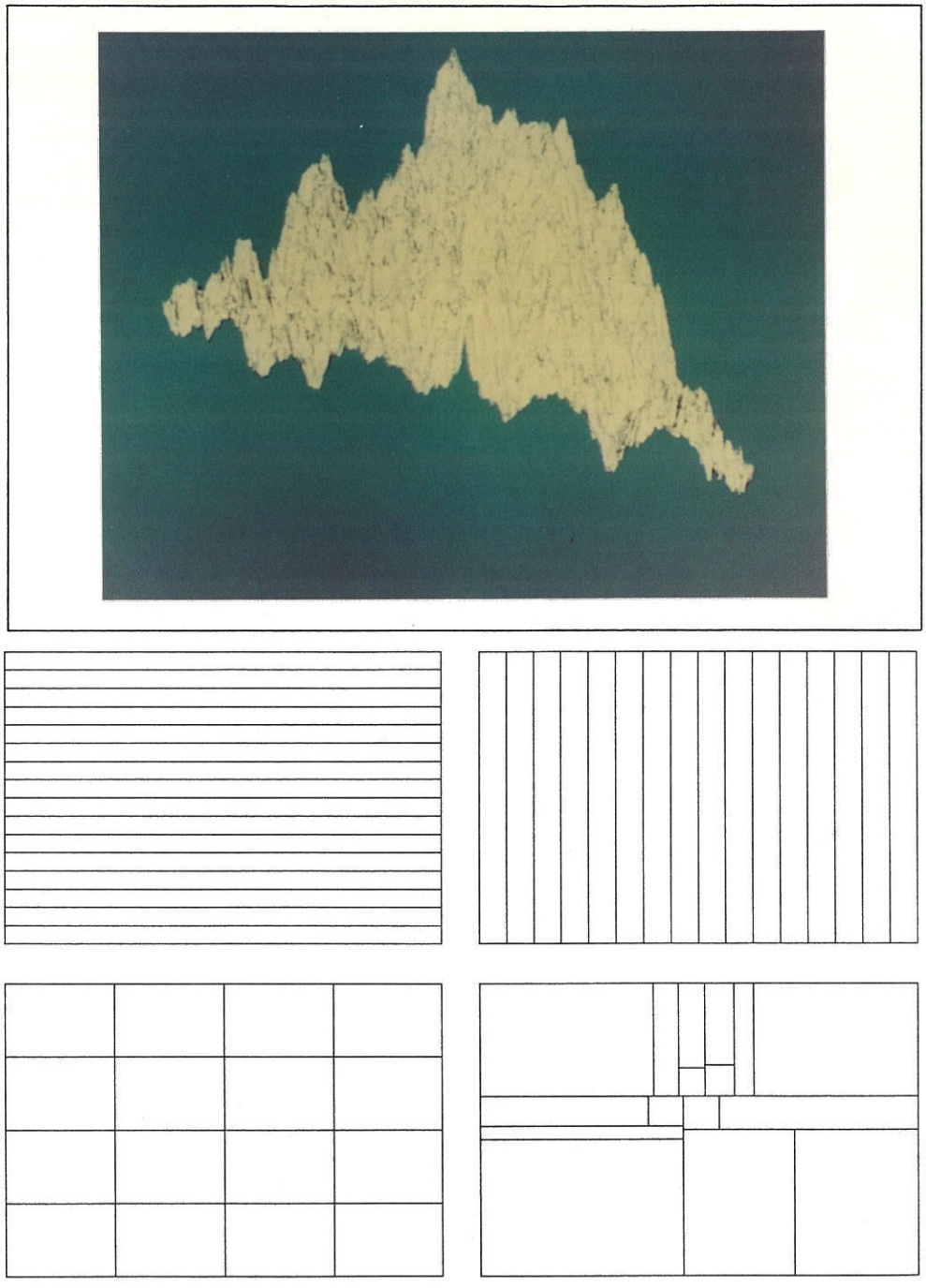
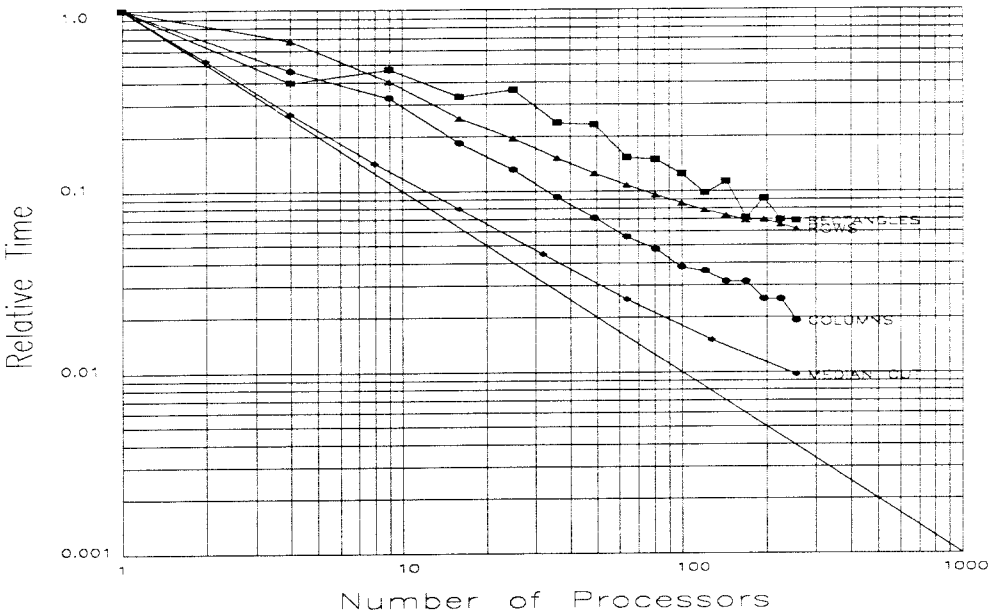


Figure A.16: Fractal Landscape (frac85)



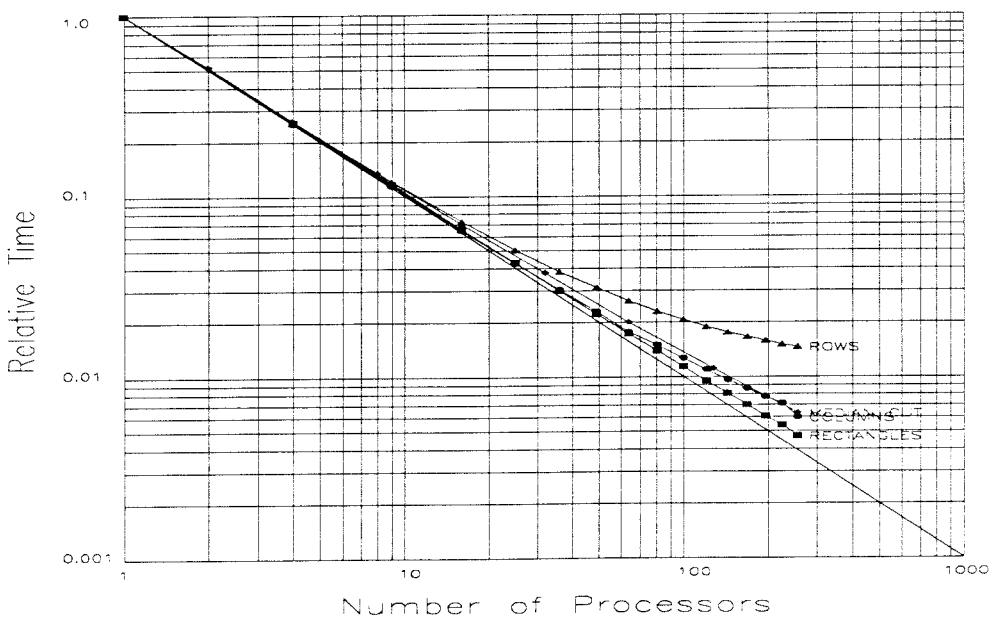


Figure A.16: Fractal Landscape (frac85) *(cont.)*

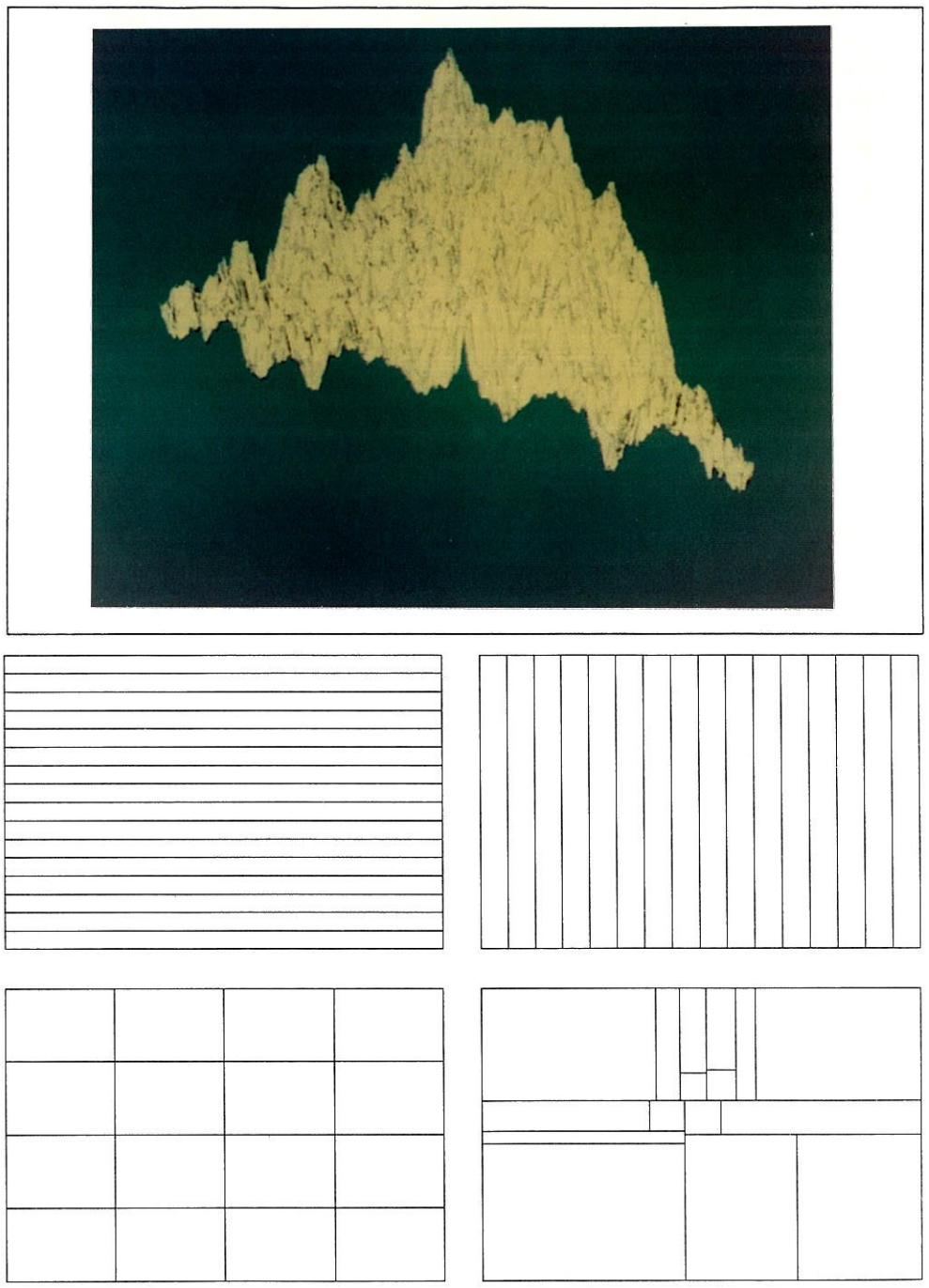
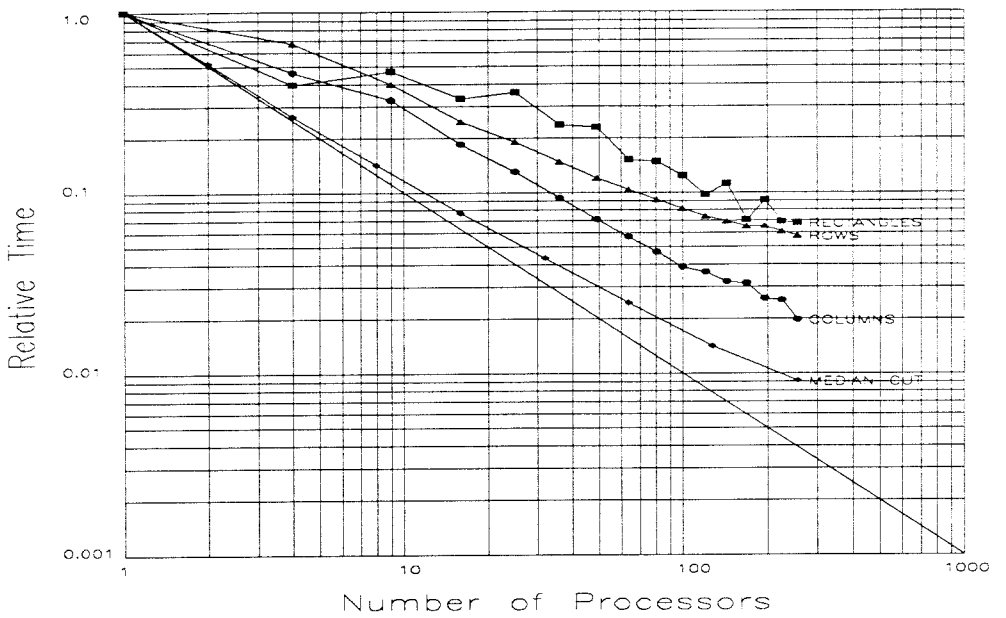


Figure A.17: Fractal Landscape (frac96)



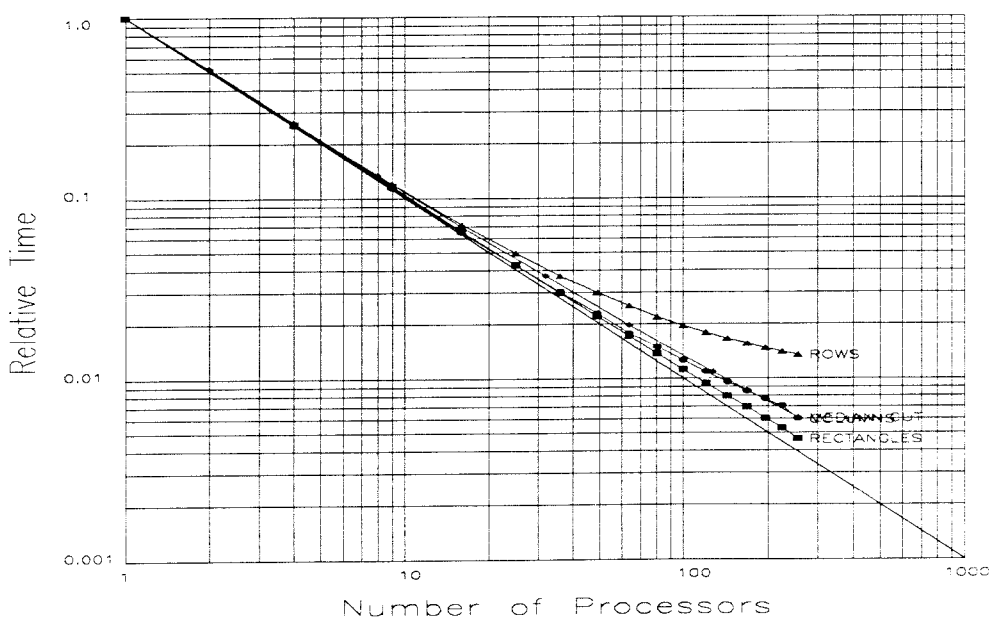


Figure A.17: Fractal Landscape (frac96) *(cont.)*

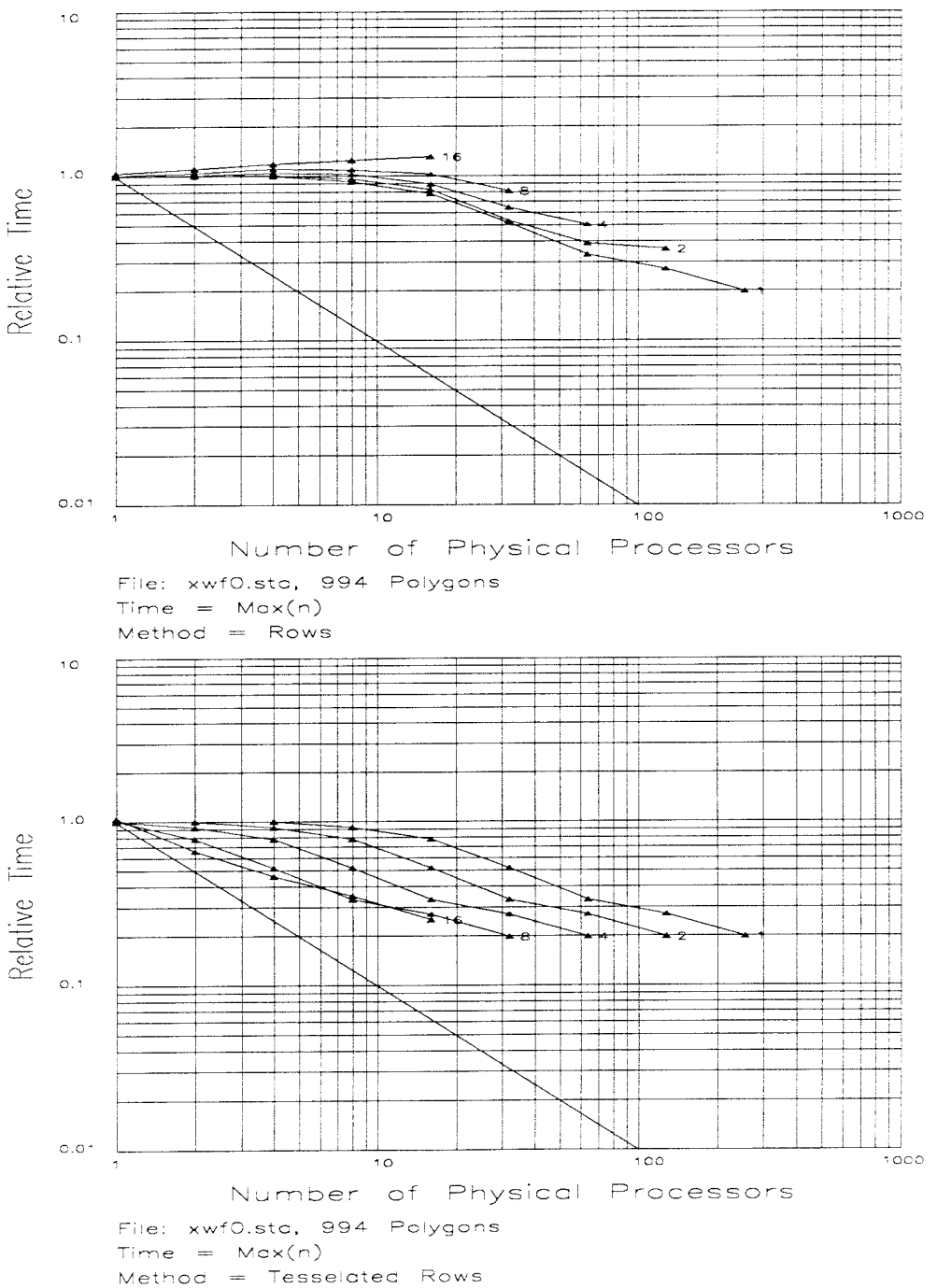
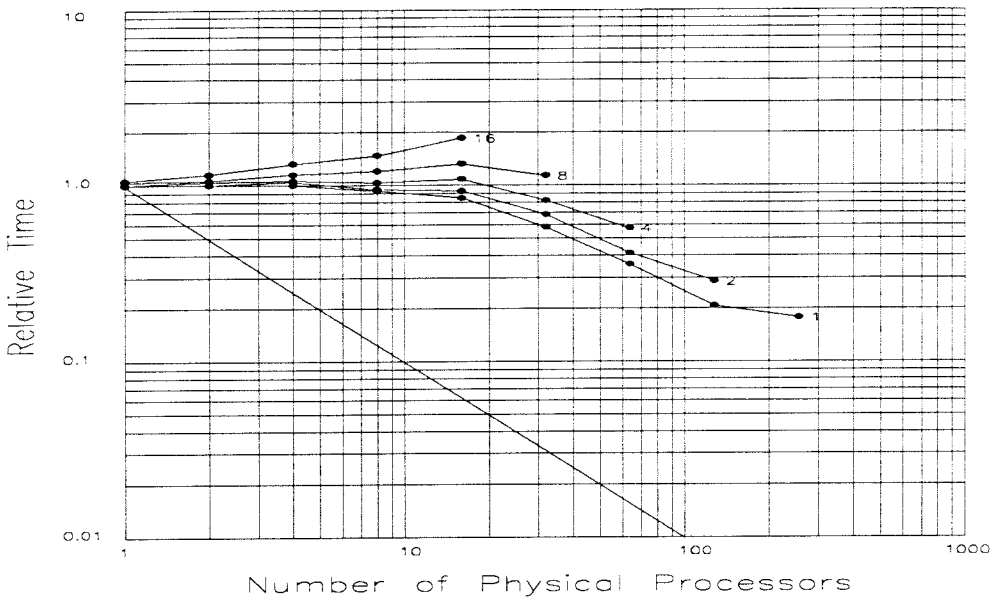


Figure A.18: X-Wing Virtual Simulation Results (xwf0)



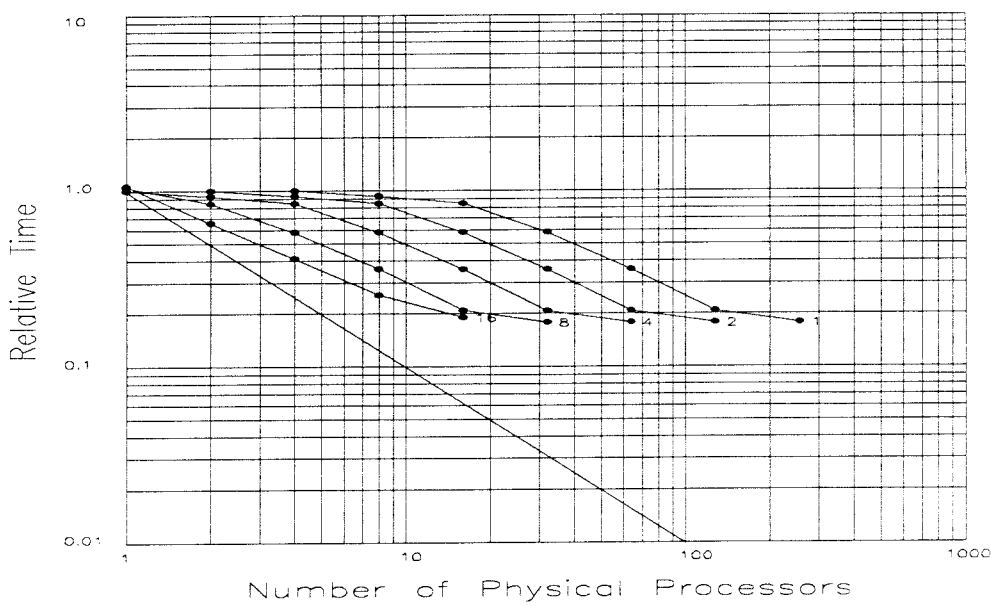
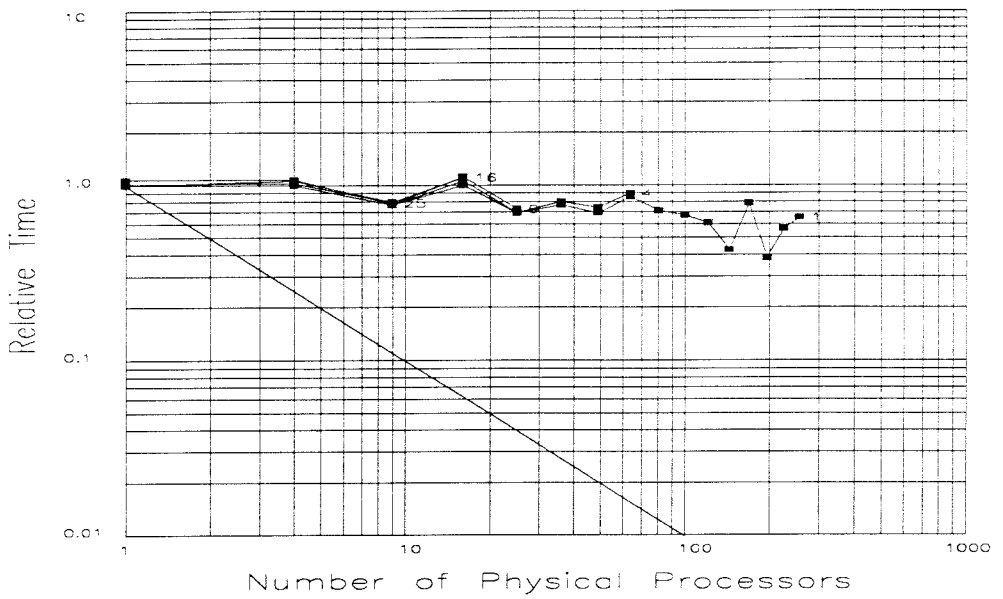


Figure A.18: X-Wing Virtual Simulation Results (xwf0) *(cont.)*



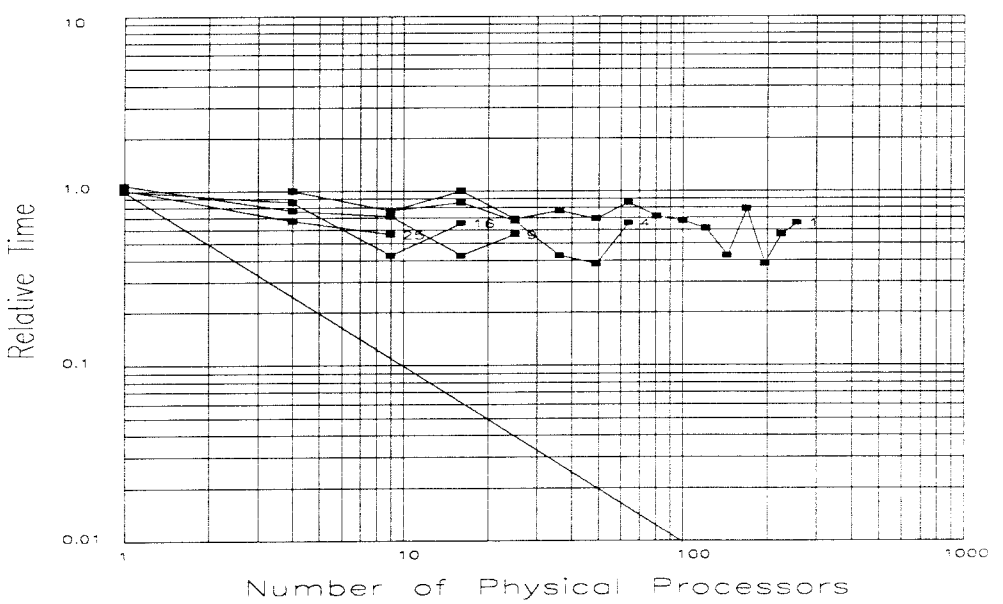
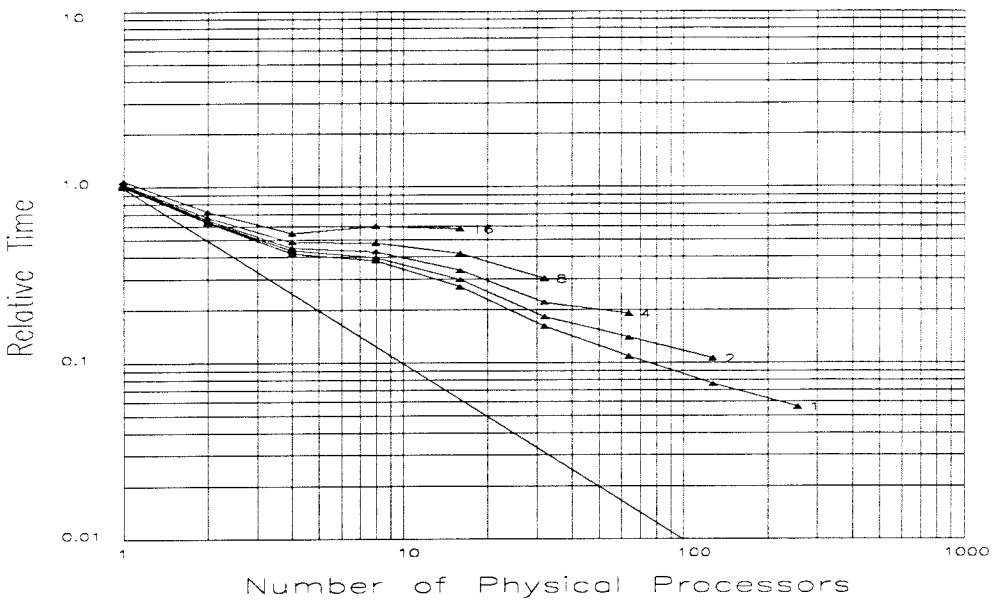


Figure A.18: X-Wing Virtual Simulation Results (xwf0) *(cont.)*



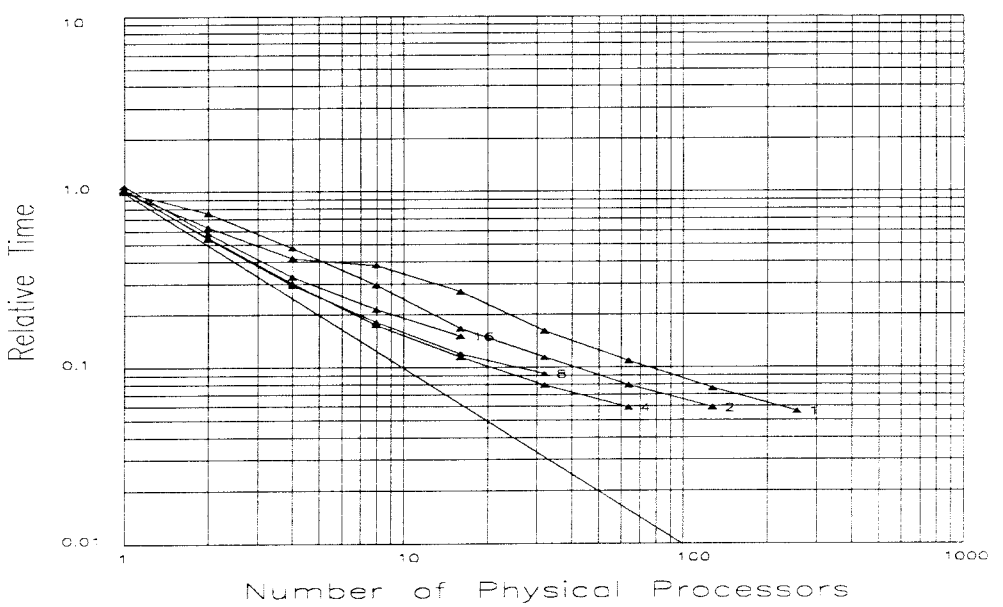
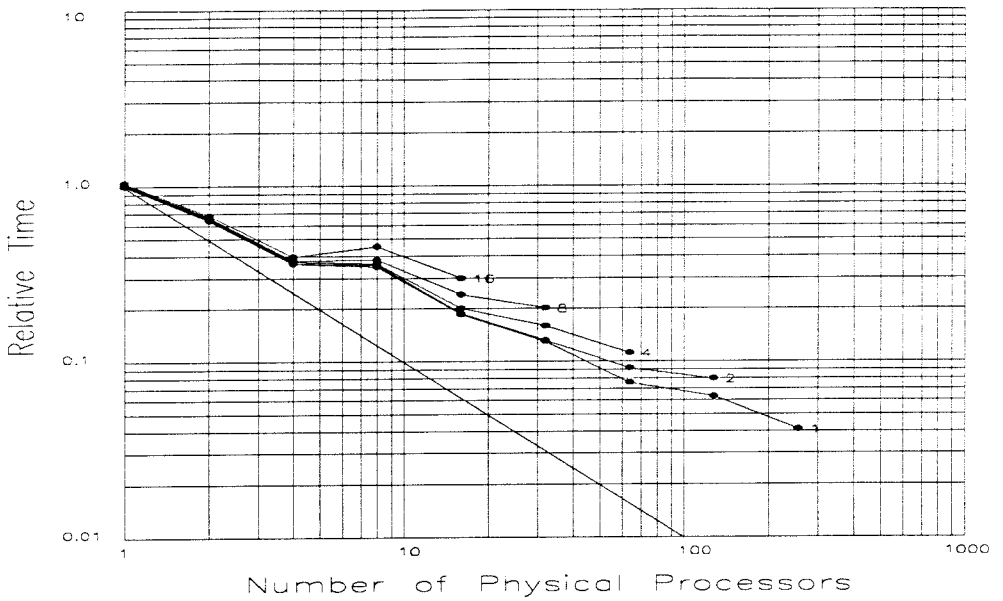


Figure A.19: X-Wing Virtual Simulation Results (xwf1)



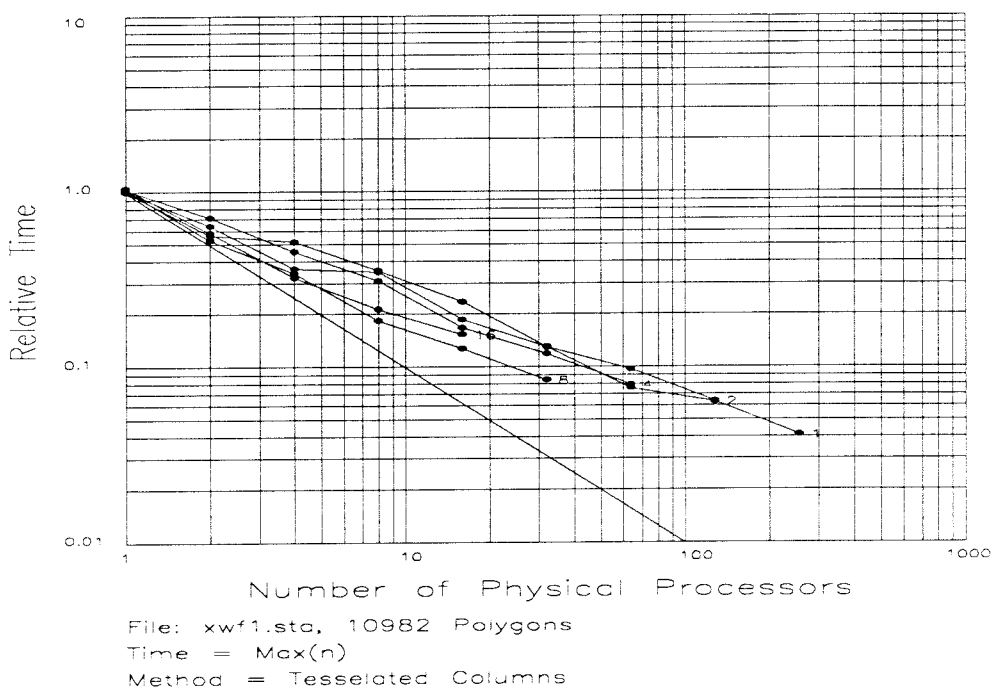
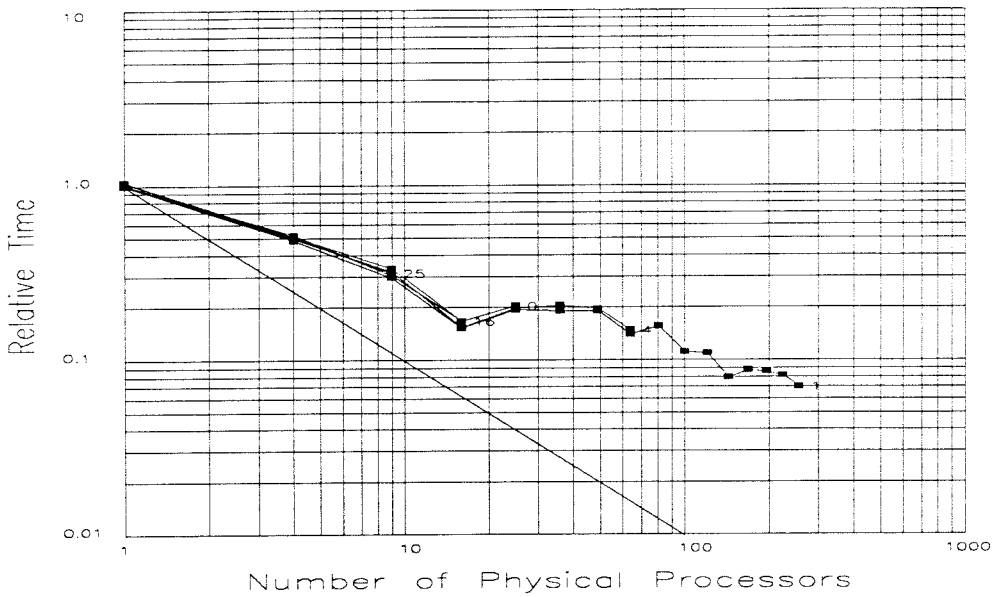


Figure A.19: X-Wing Virtual Simulation Results (xwf1) *(cont.)*



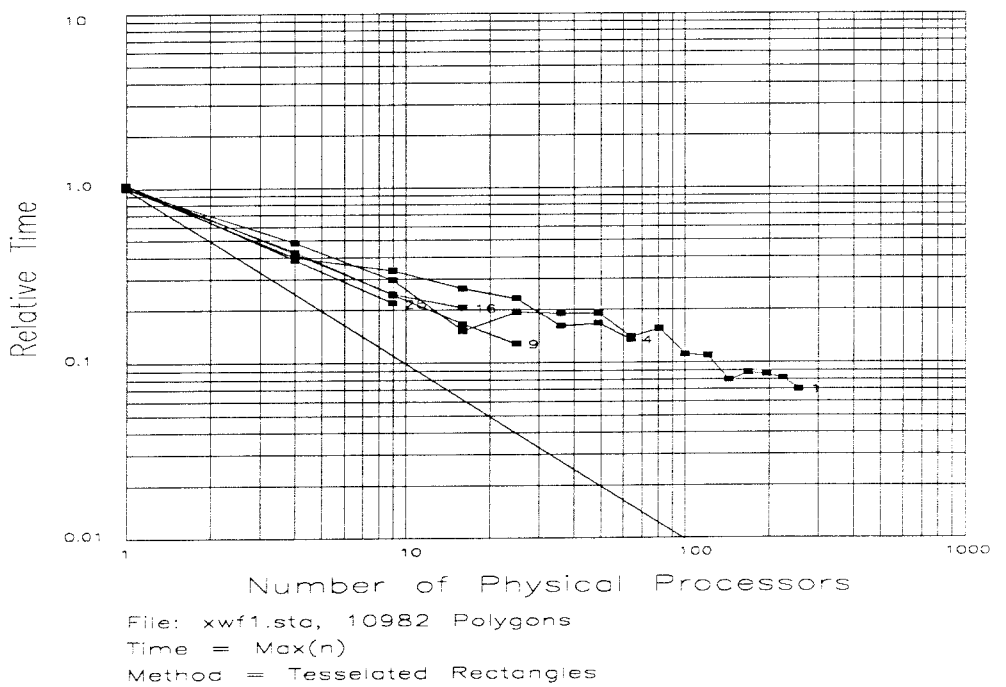
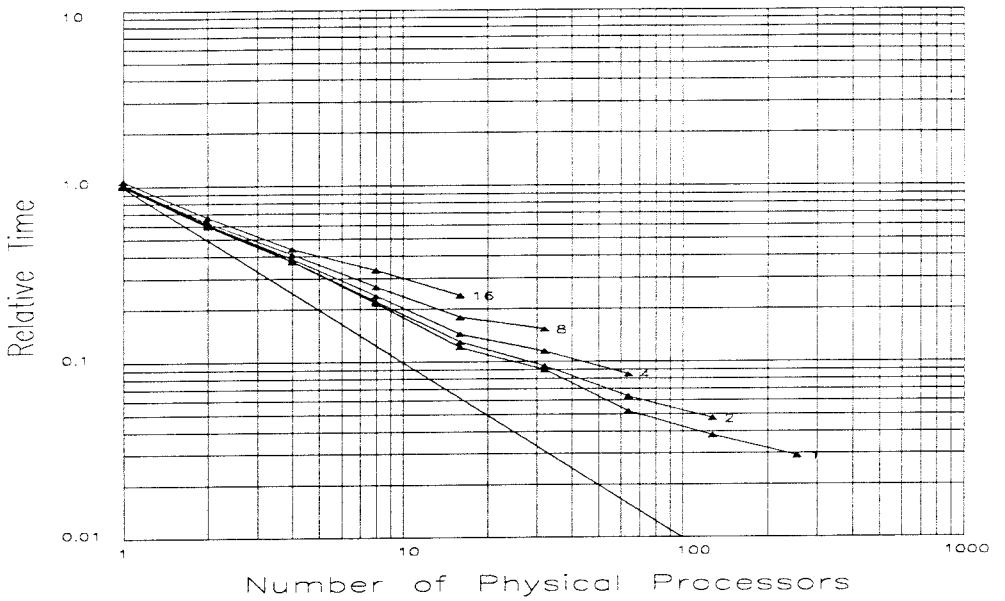


Figure A.19: X-Wing Virtual Simulation Results (xwf1) *(cont.)*



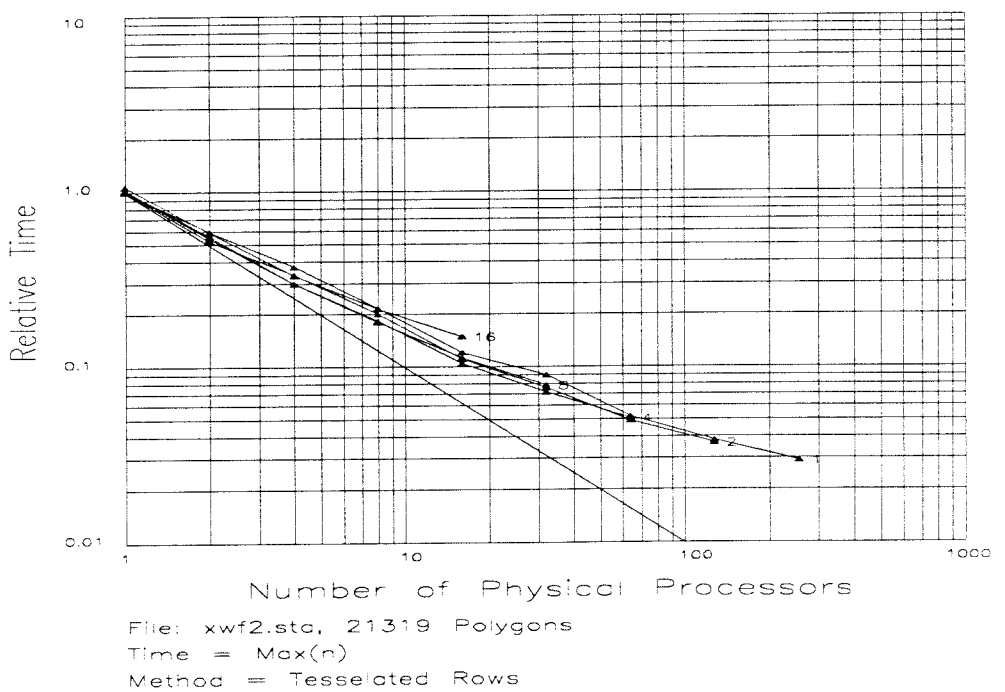
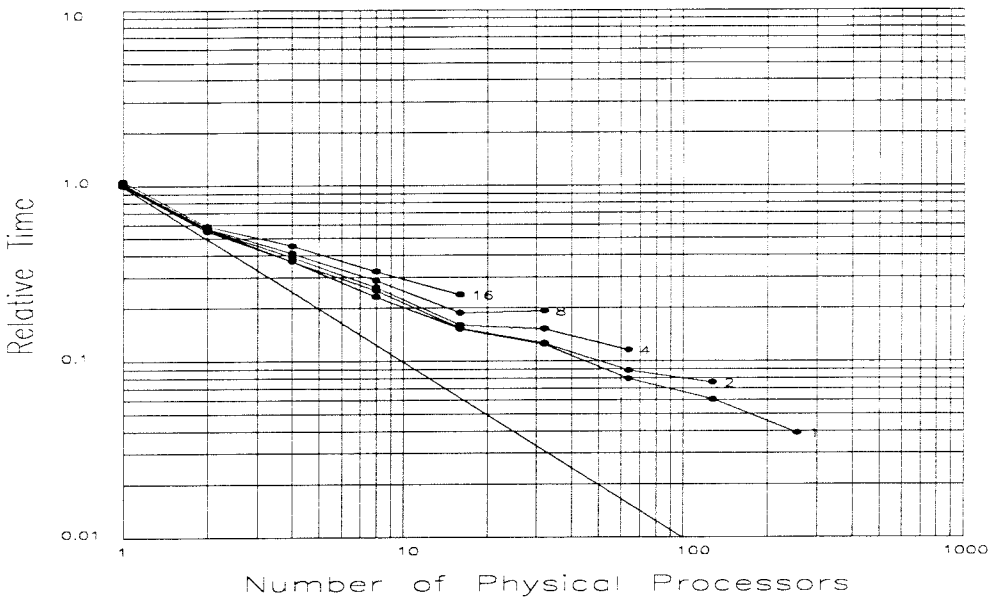


Figure A.20: X-Wing Virtual Simulation Results (xwf2)



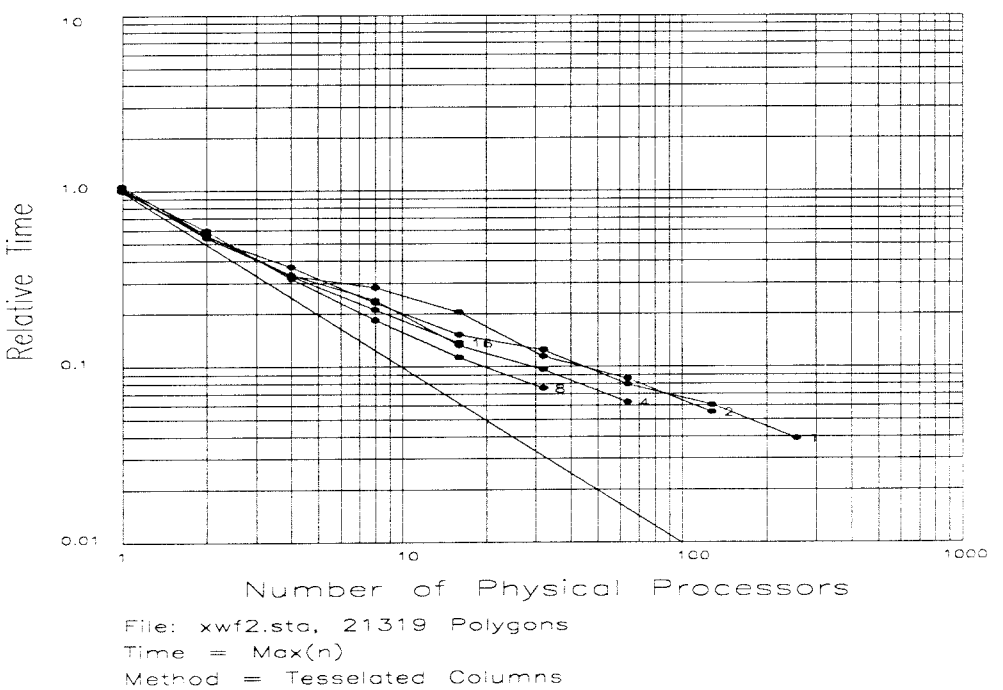
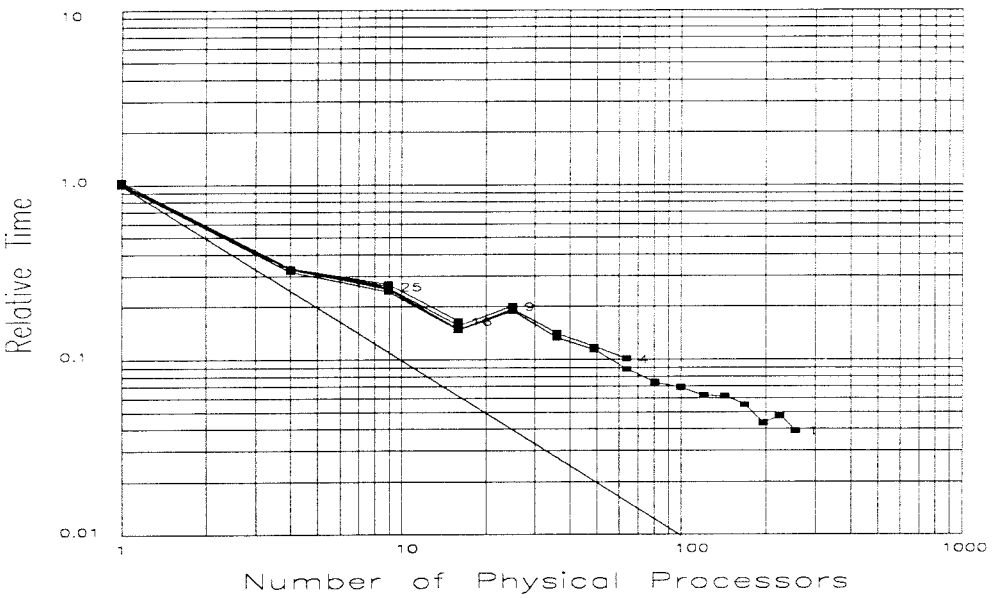


Figure A.20: X-Wing Virtual Simulation Results (xwf2) *(cont.)*



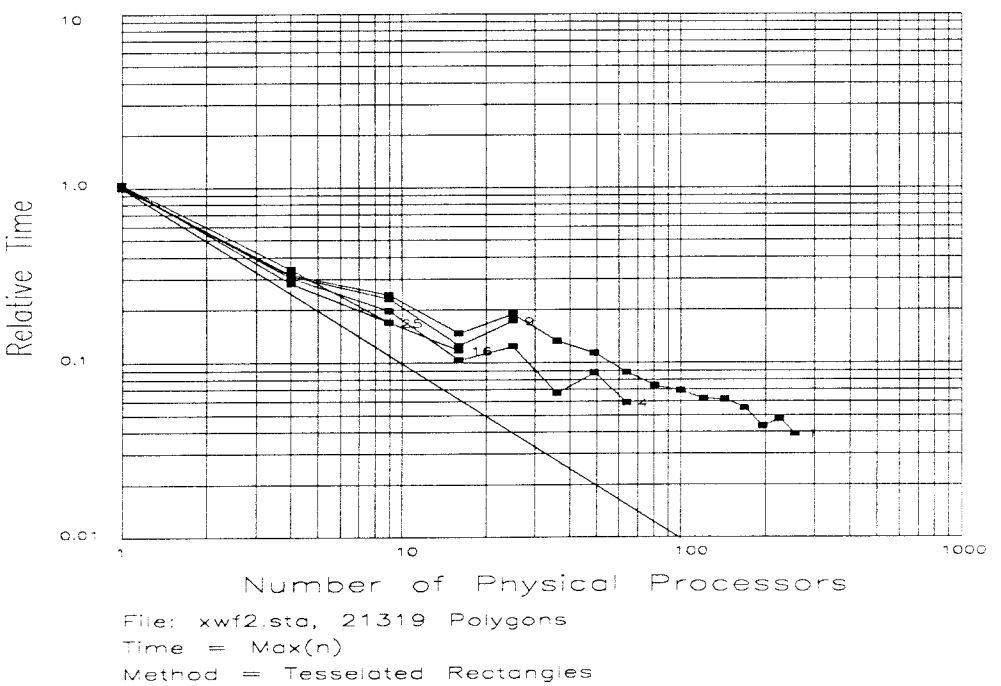
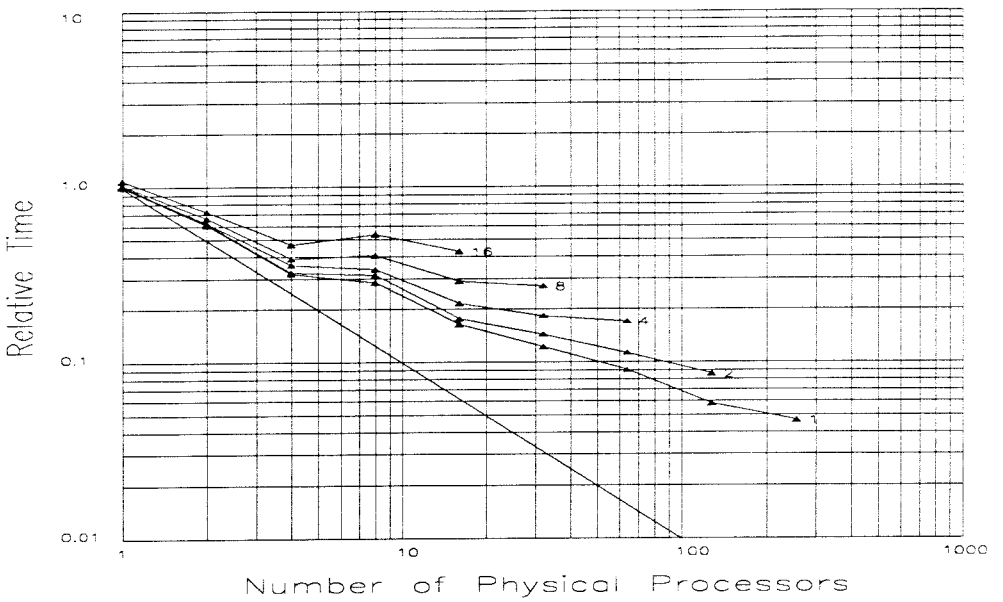


Figure A.20: X-Wing Virtual Simulation Results (xwf2) *(cont.)*



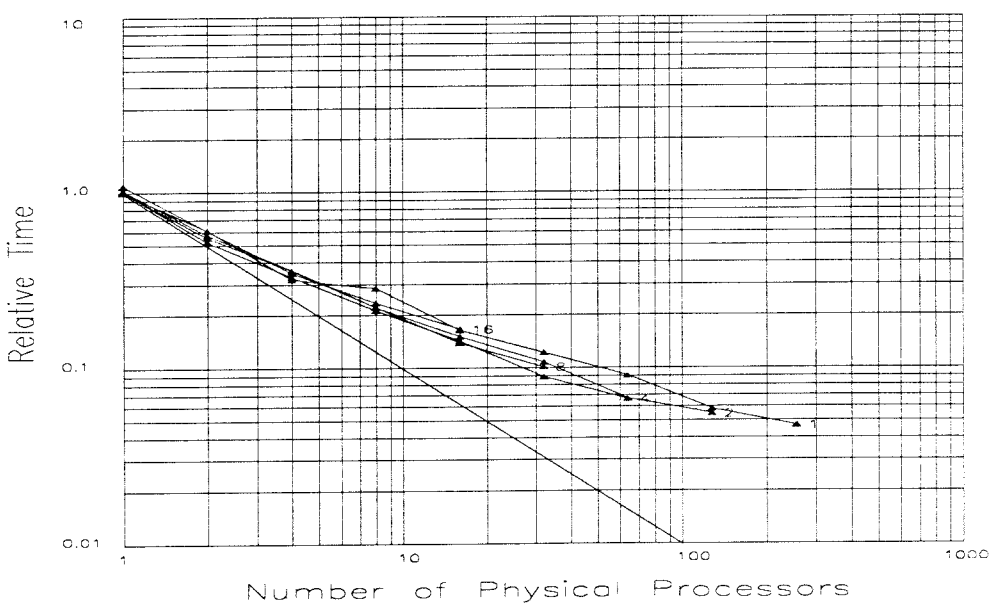
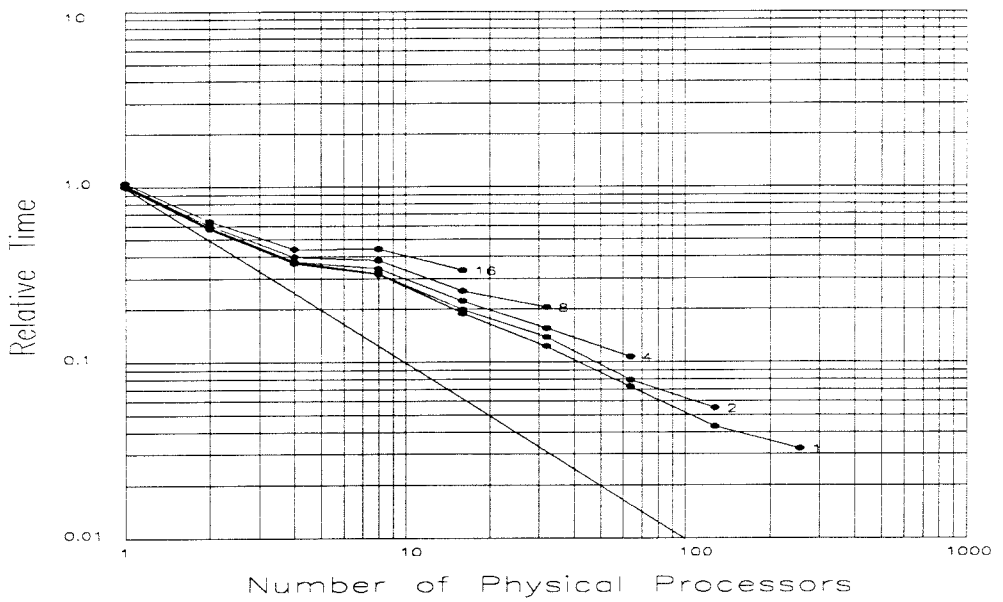


Figure A.21: X-Wing Virtual Simulation Results (xwf3)



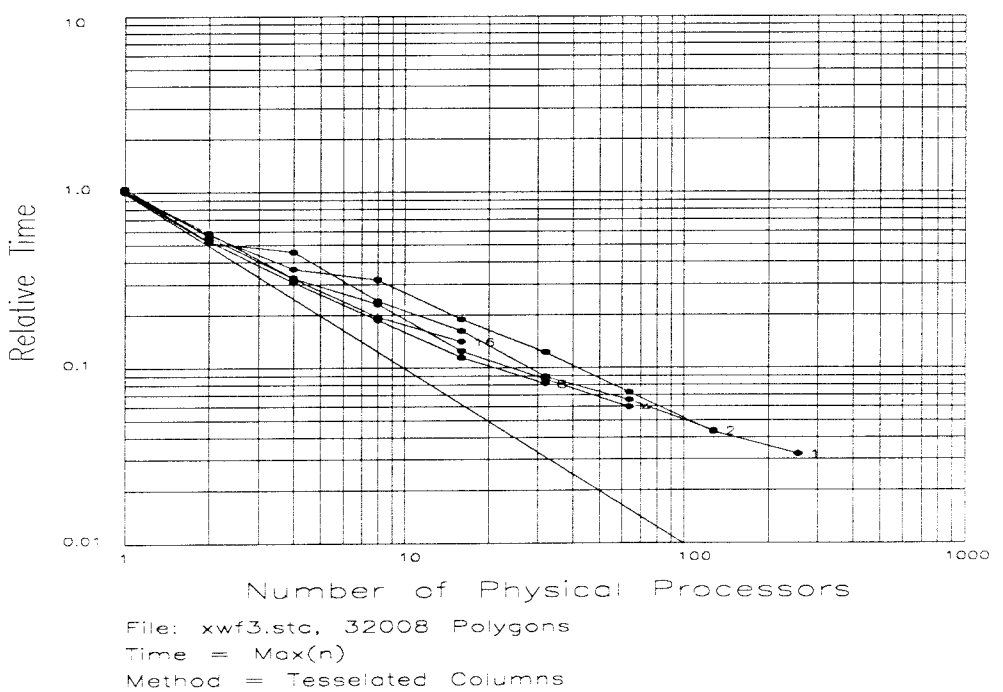
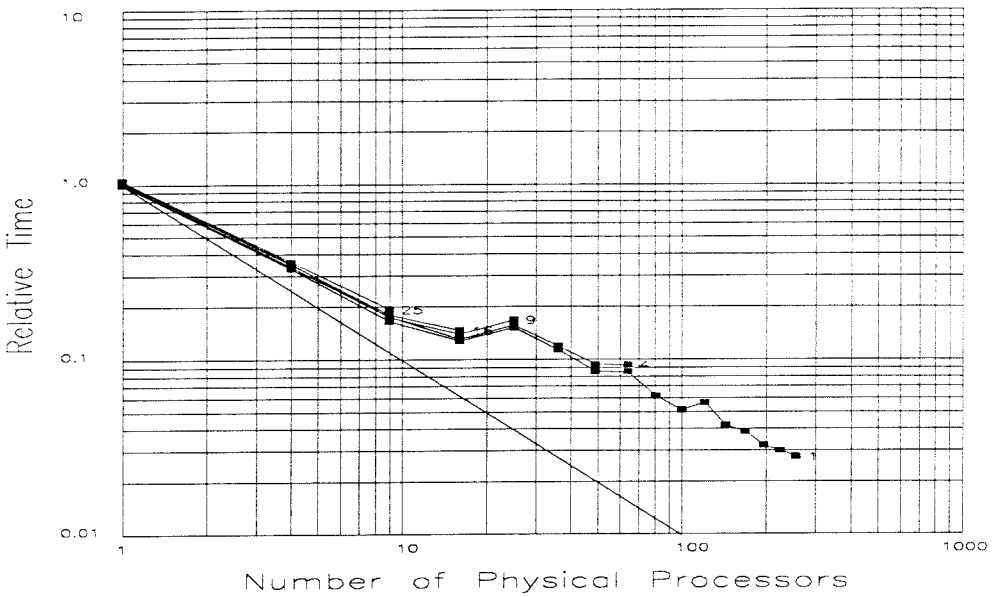


Figure A.21: X-Wing Virtual Simulation Results (xwf3) *(cont.)*



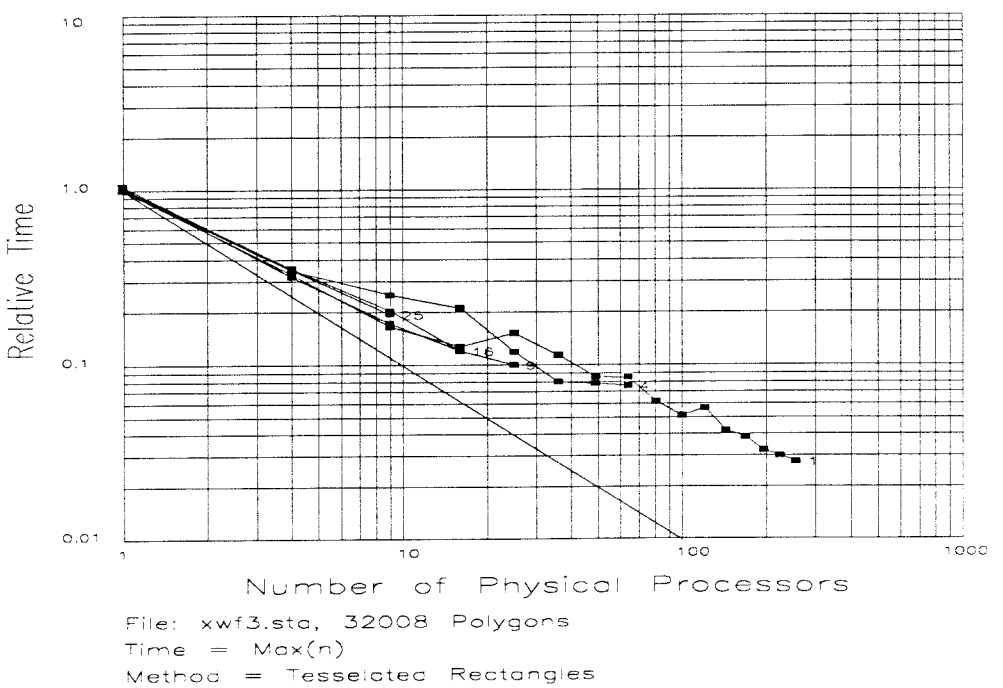
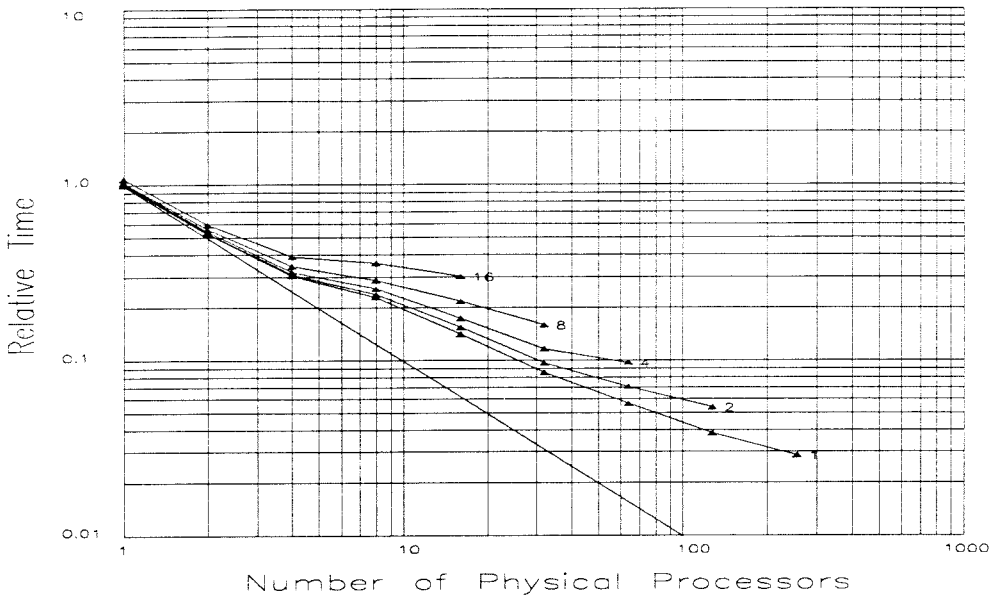


Figure A.21: X-Wing Virtual Simulation Results (xwf3) *(cont.)*



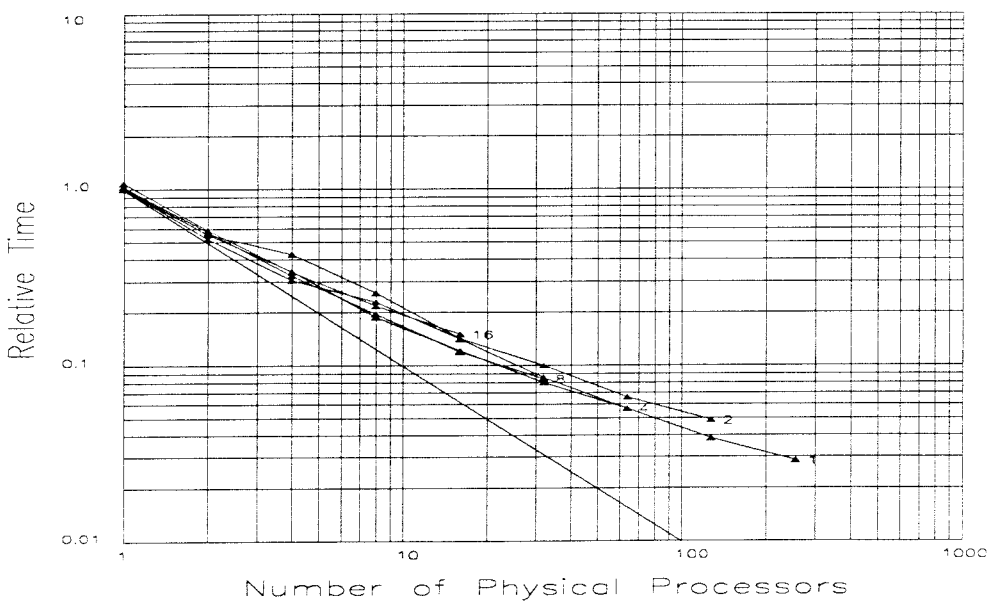
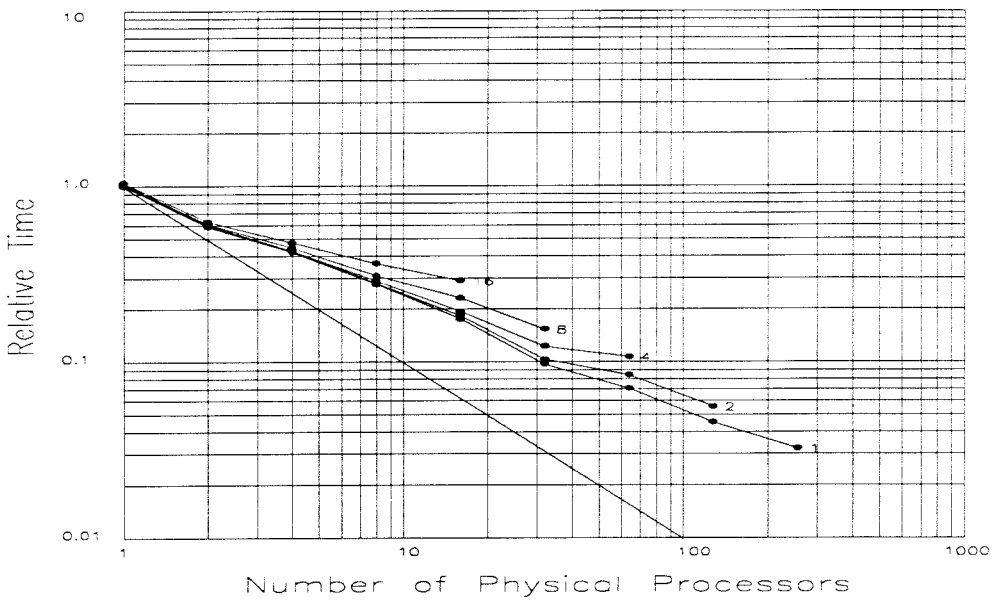
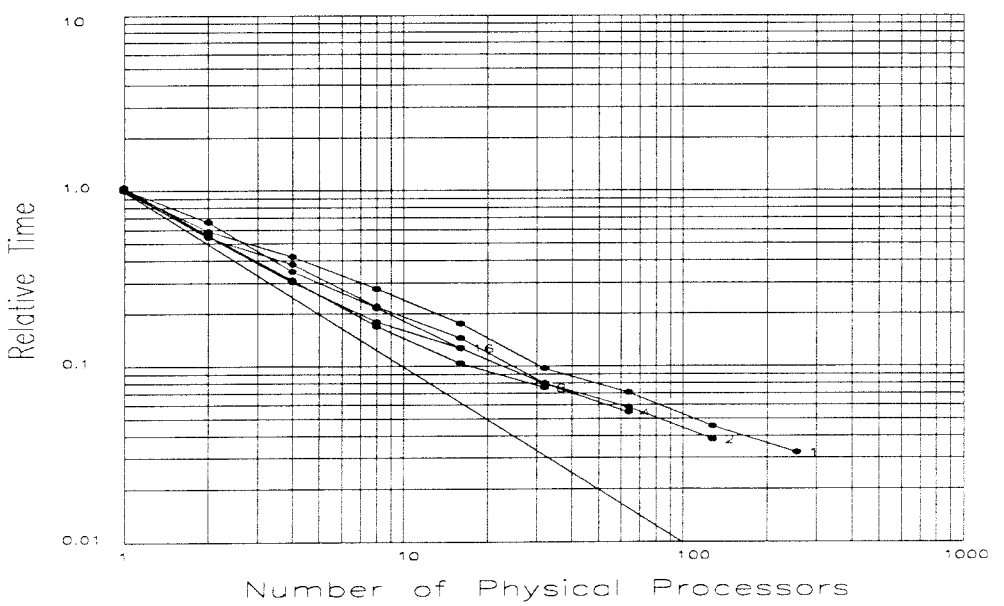


Figure A.22: X-Wing Virtual Simulation Results (xwf4)

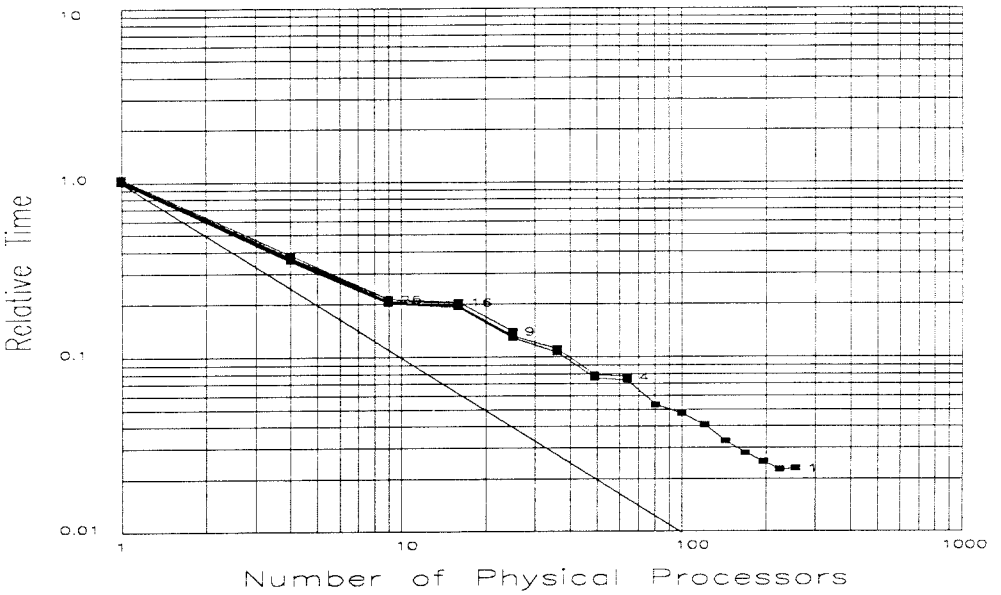


File: xwf4.sta, 43146 Polygons
Time = Max(n)
Method = Columns



File: xwf4.sta, 43146 Polygons
Time = Max(n)
Method = Tesselated Columns

Figure A.22: X-Wing Virtual Simulation Results (xwf4) *(cont.)*



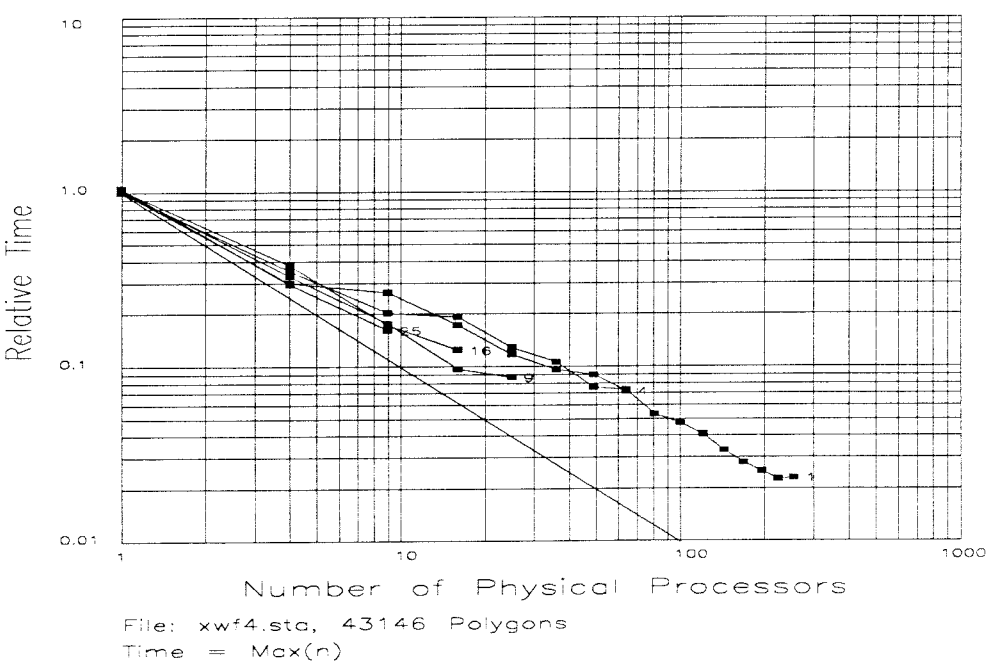
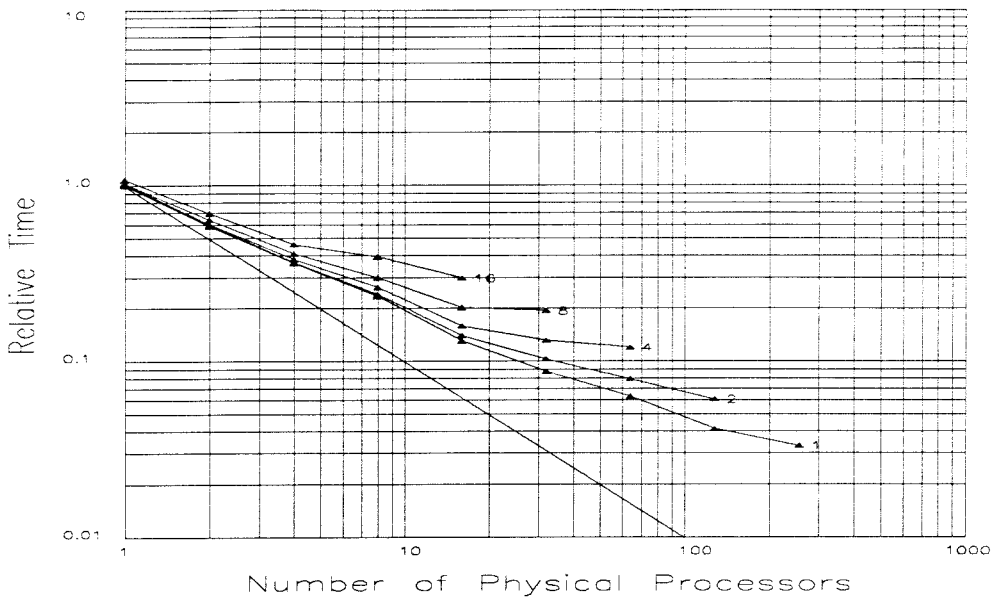


Figure A.22: X-Wing Virtual Simulation Results (xwf4) *(cont.)*



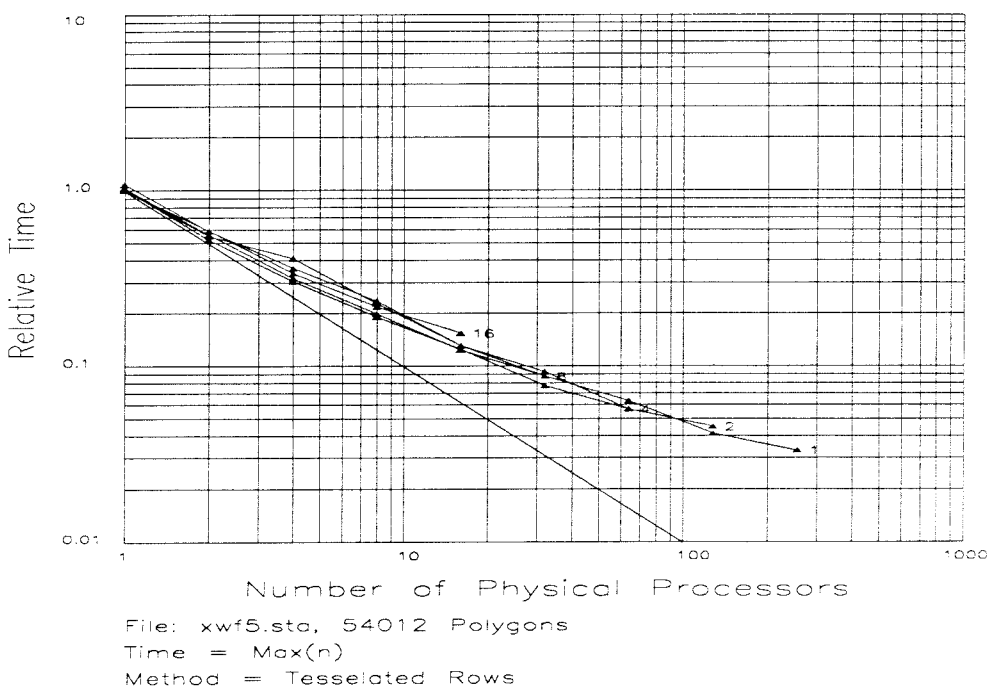
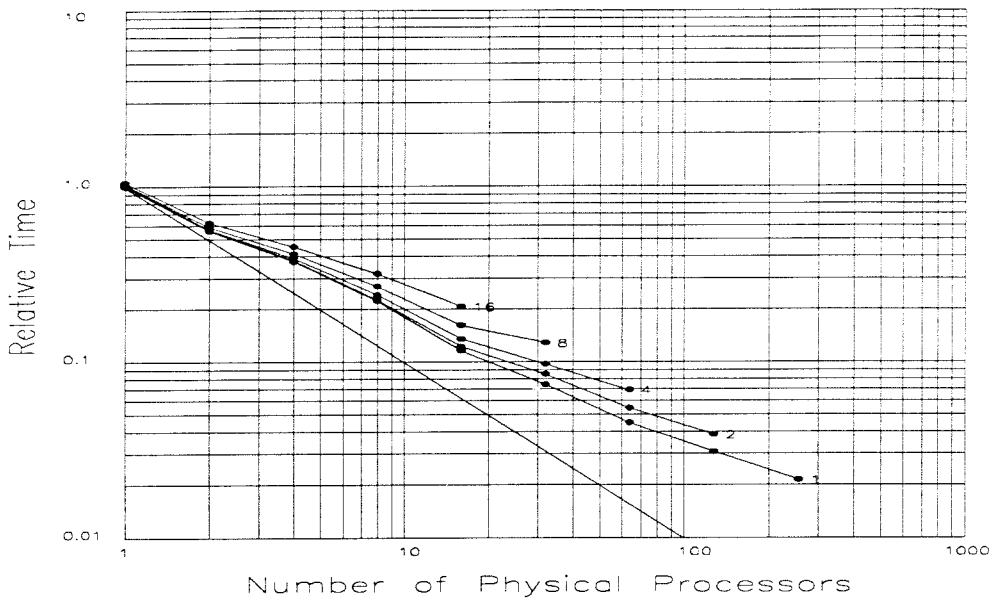


Figure A.23: X-Wing Virtual Simulation Results (xwf5)



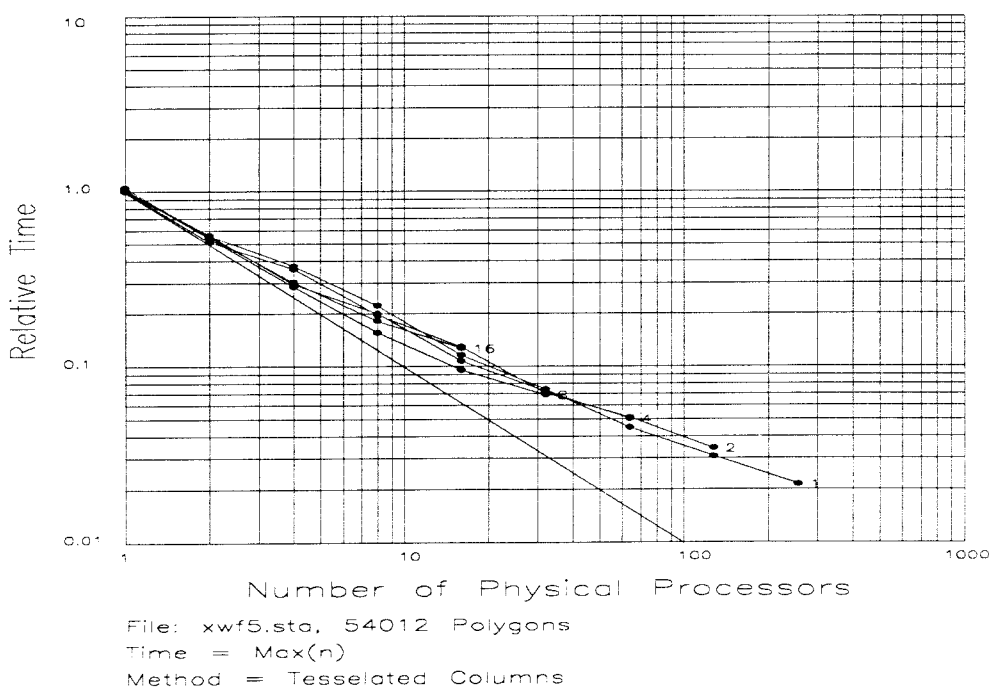
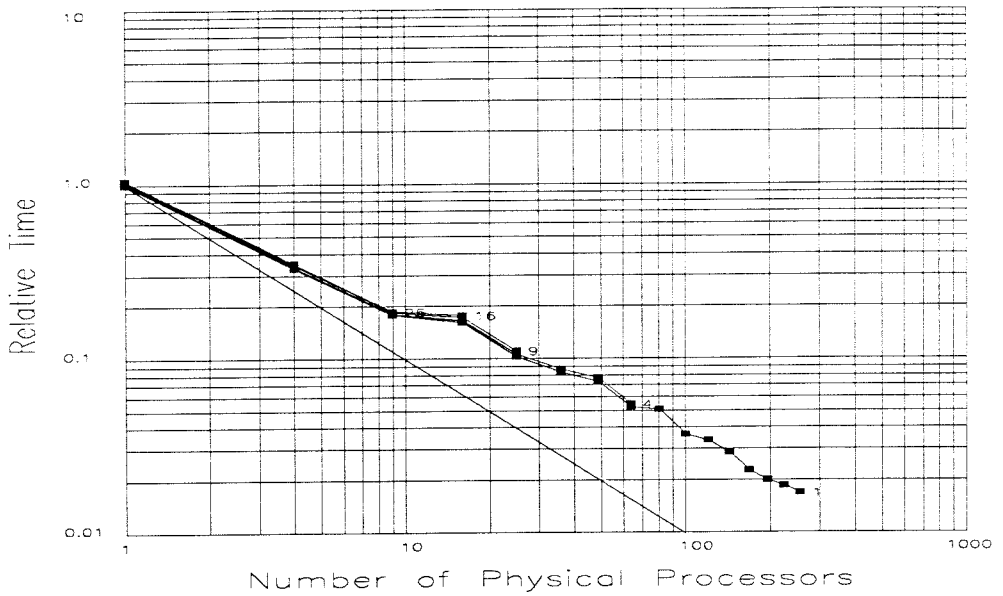


Figure A.23: X-Wing Virtual Simulation Results (xwf5) *(cont.)*



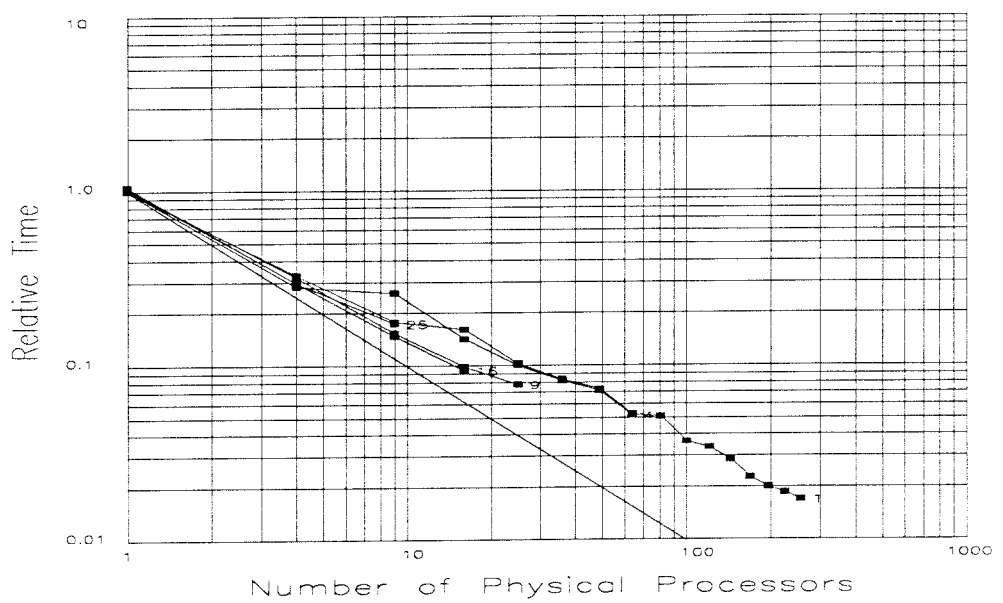
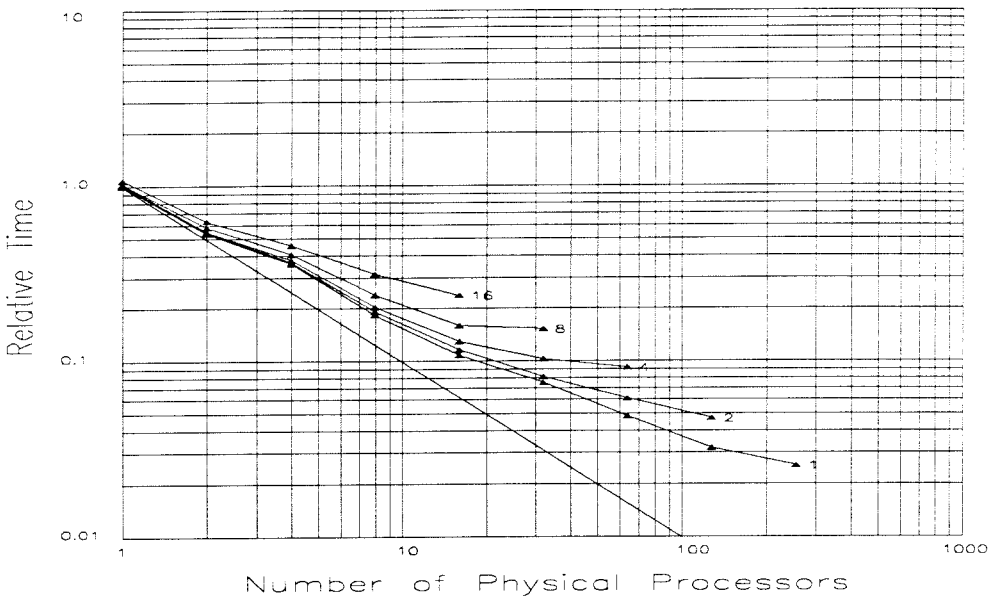


Figure A.23: X-Wing Virtual Simulation Results (xwf5) *(cont.)*



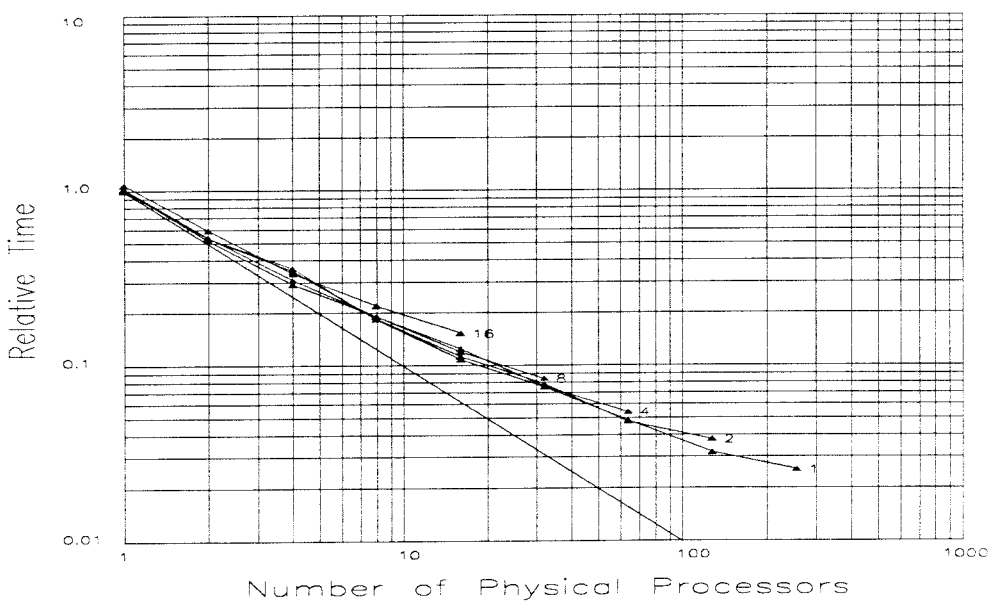
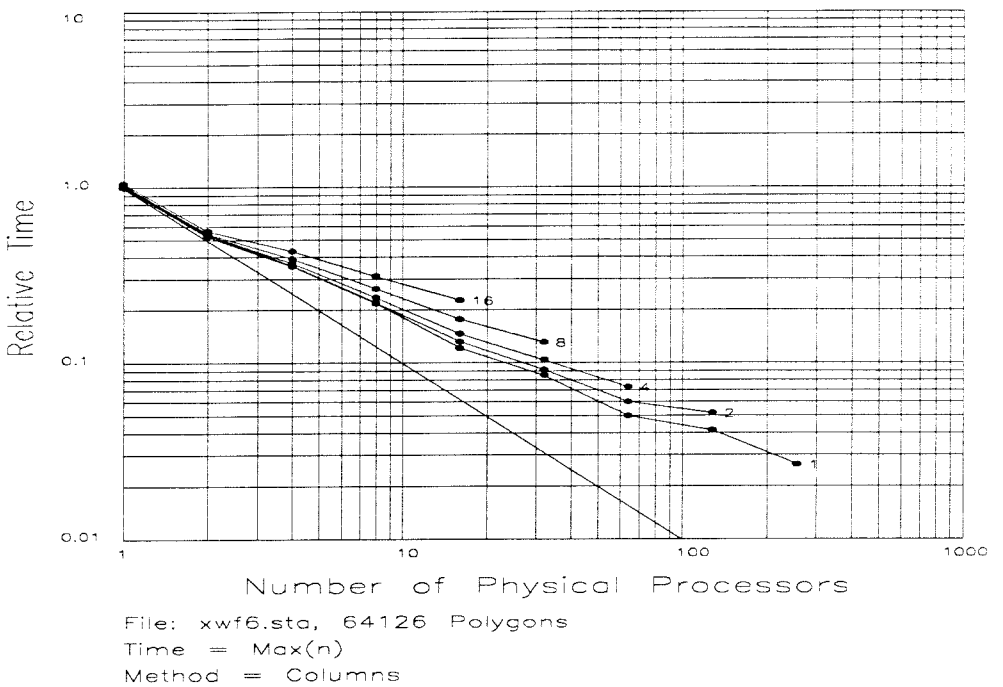


Figure A.24: X-Wing Virtual Simulation Results (xwf6)



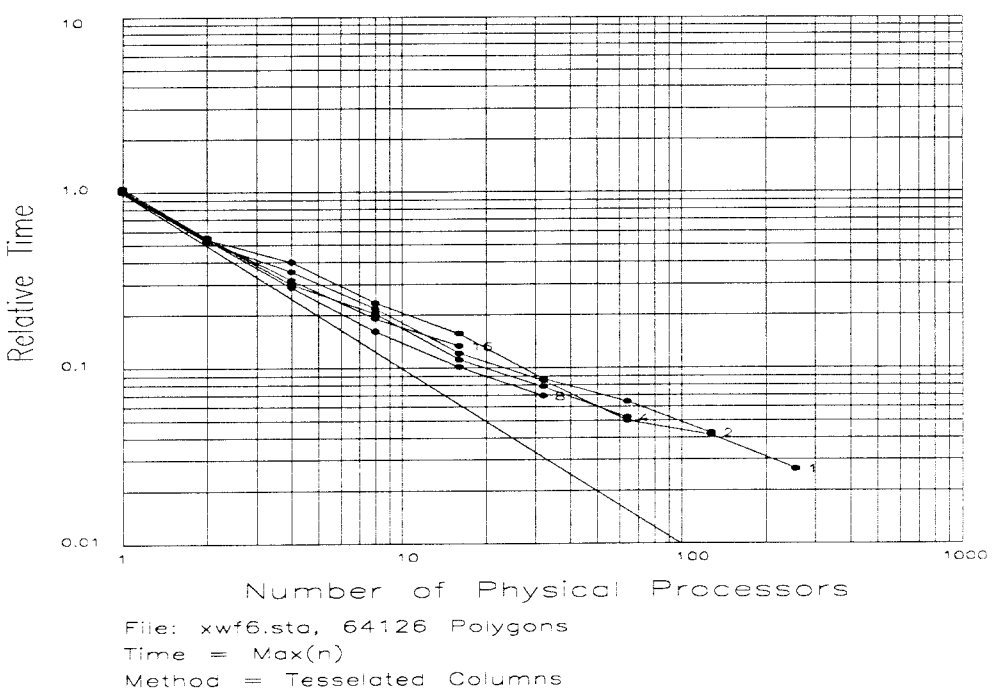
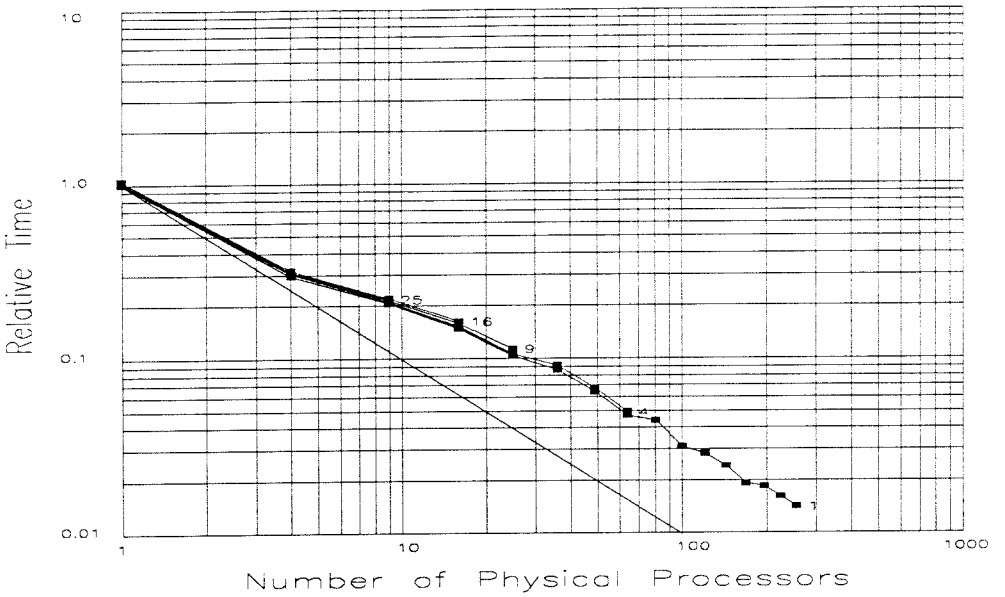


Figure A.24: X-Wing Virtual Simulation Results (xwf6) *(cont.)*



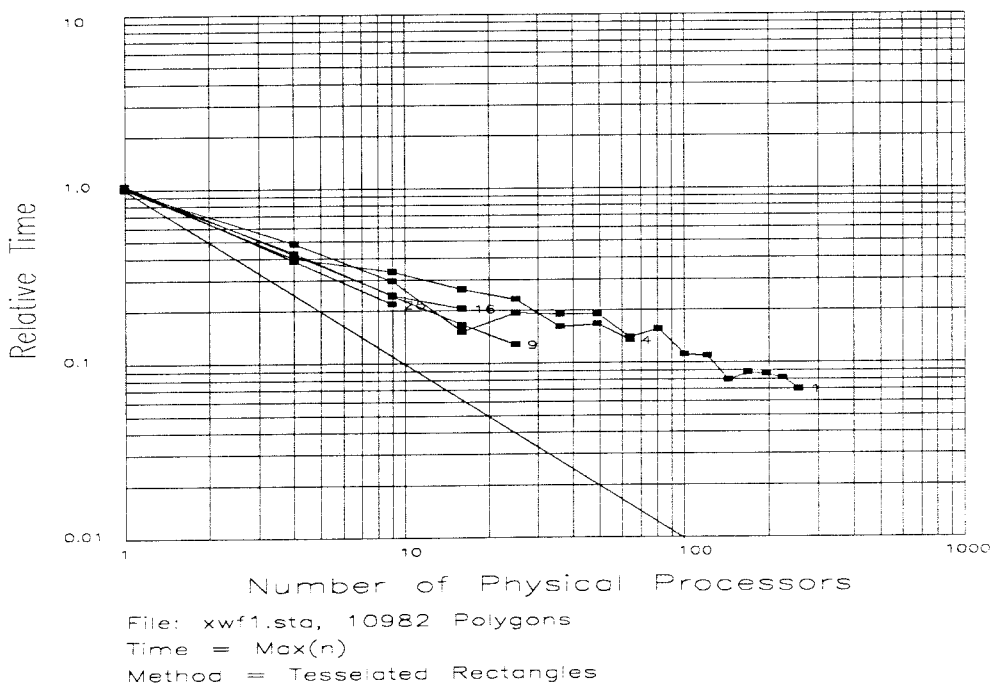
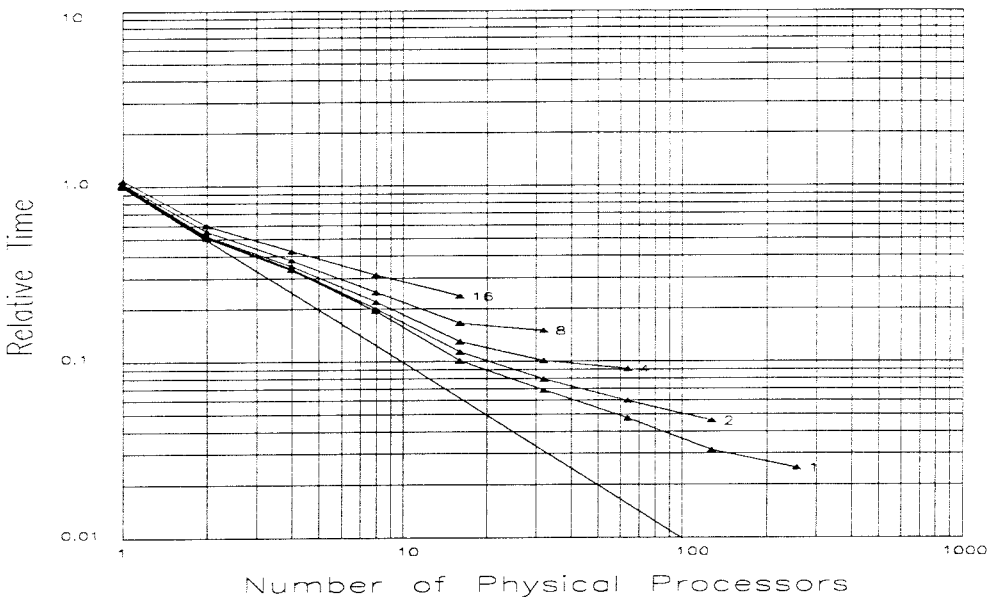


Figure A.24: X-Wing Virtual Simulation Results (xwf6) *(cont.)*



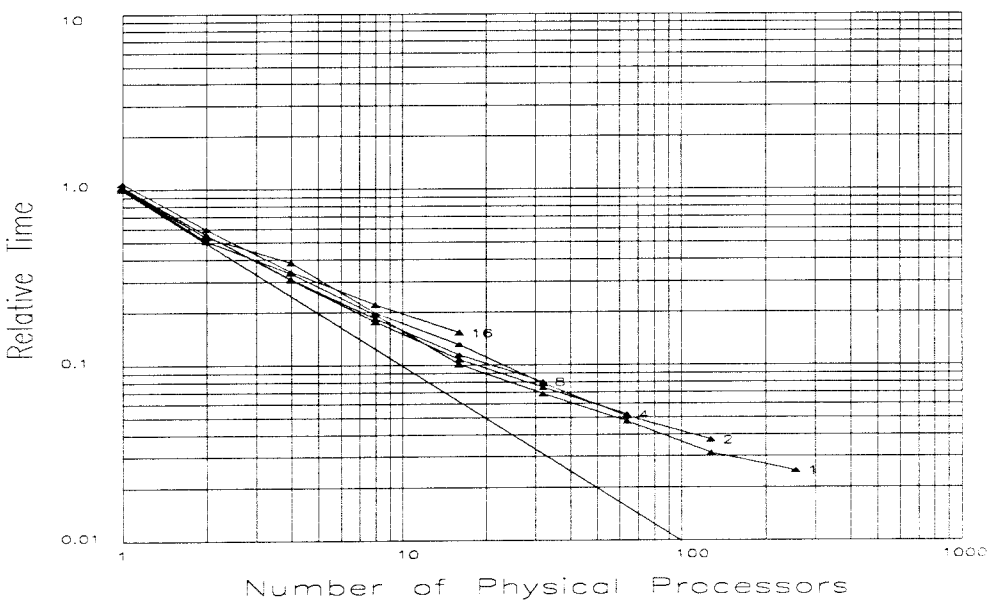
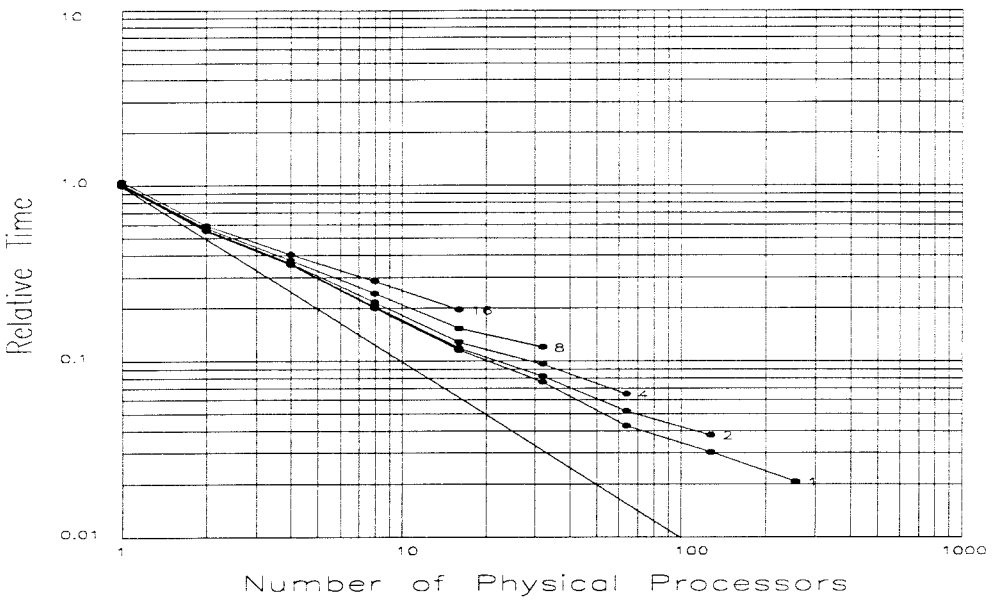


Figure A.25: X-Wing Virtual Simulation Results (xwf7)



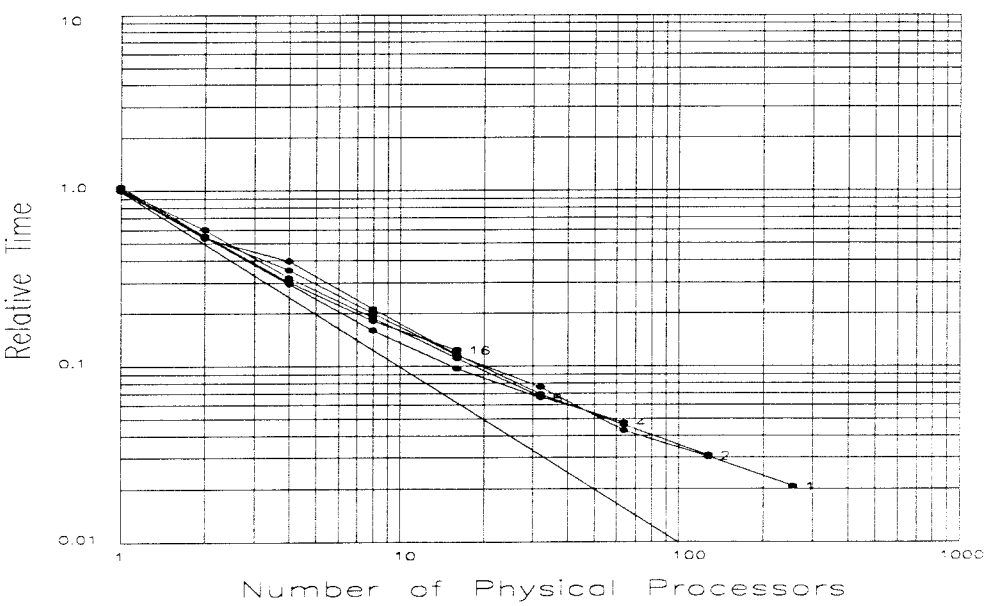
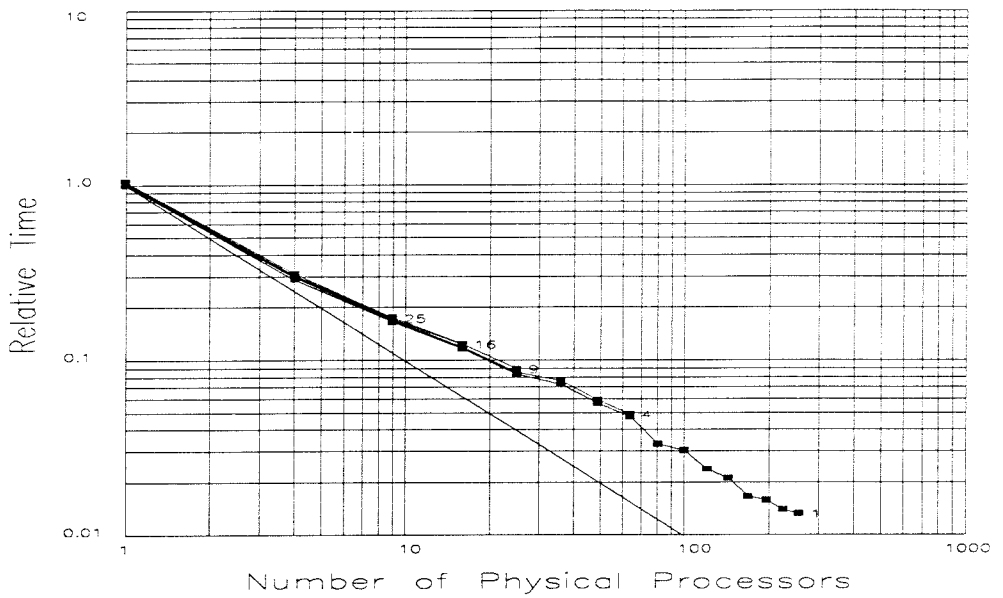


Figure A.25: X-Wing Virtual Simulation Results (xwf7) *(cont.)*



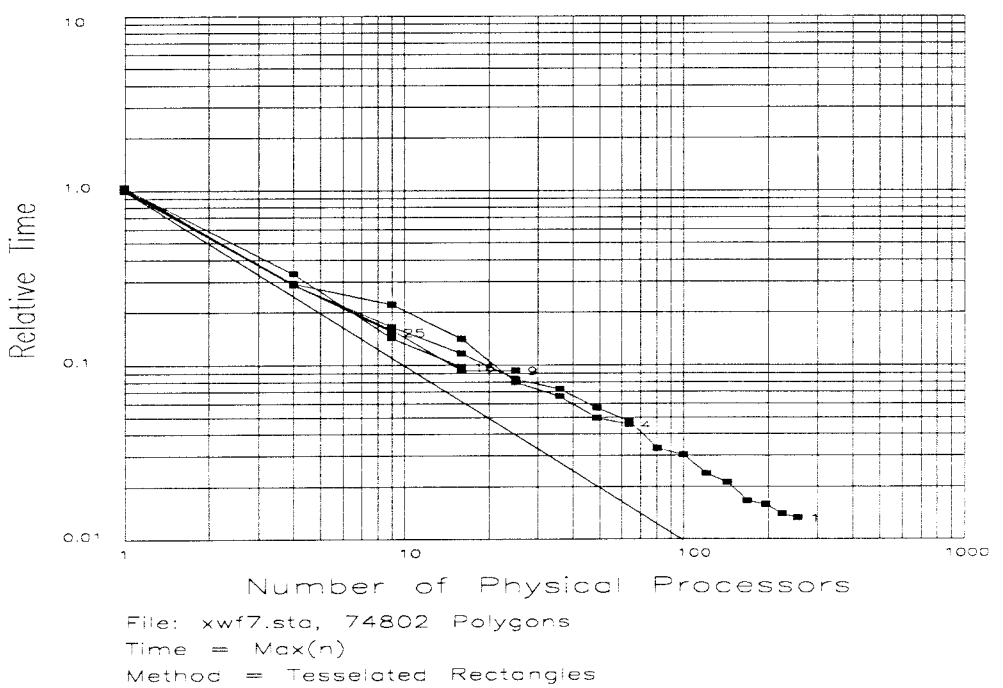
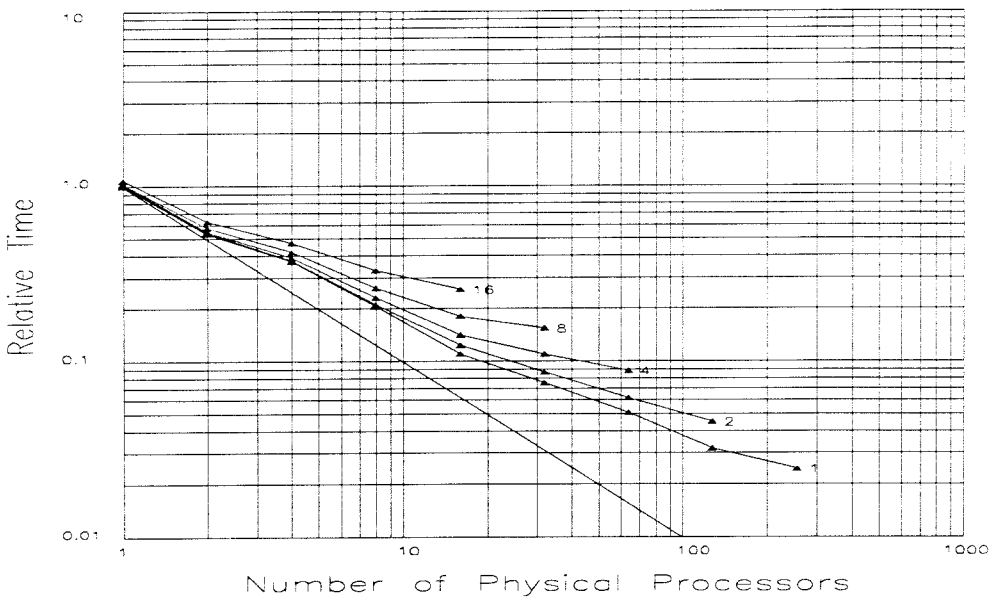


Figure A.25: X-Wing Virtual Simulation Results (xwf7) *(cont.)*



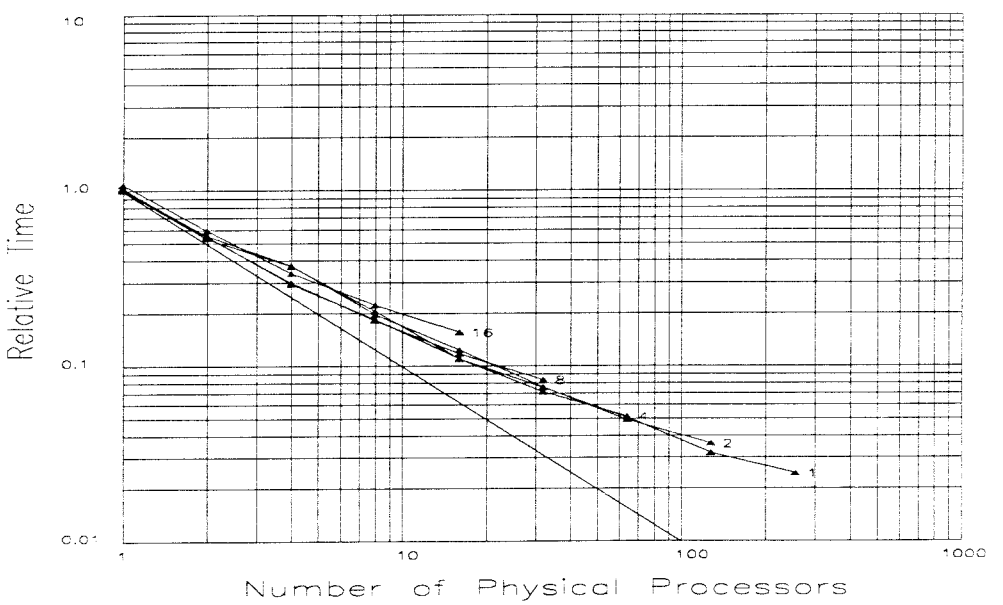
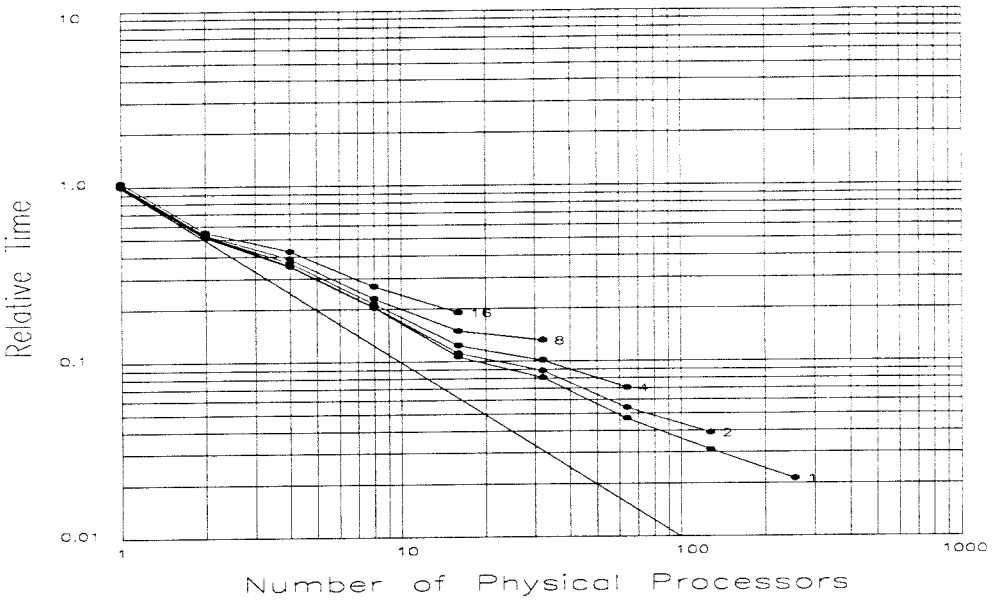


Figure A.26: X-Wing Virtual Simulation Results (xwf8)



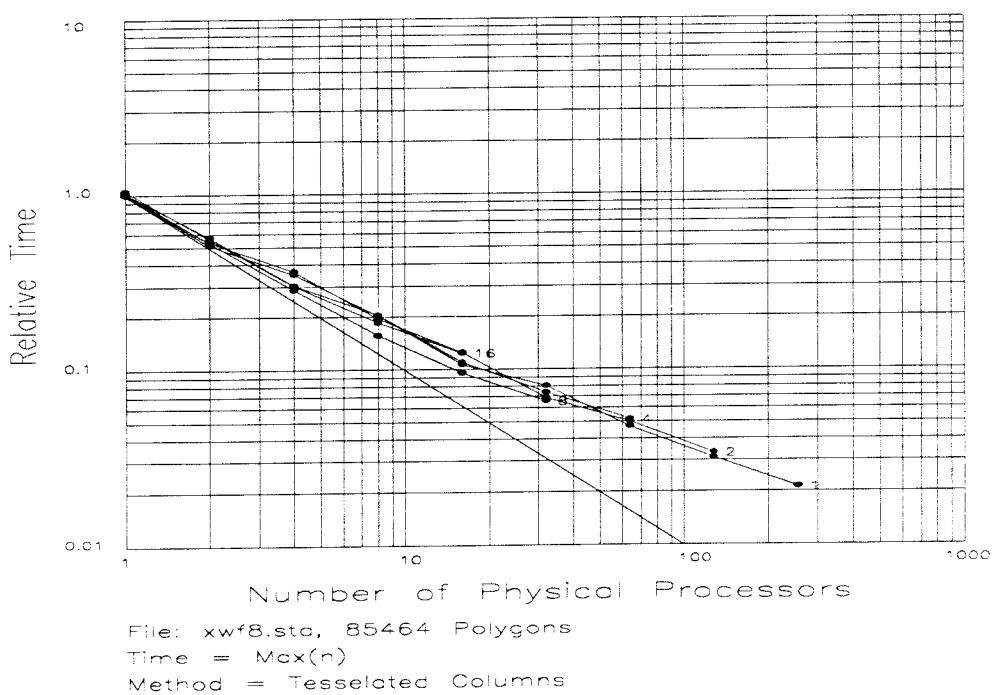
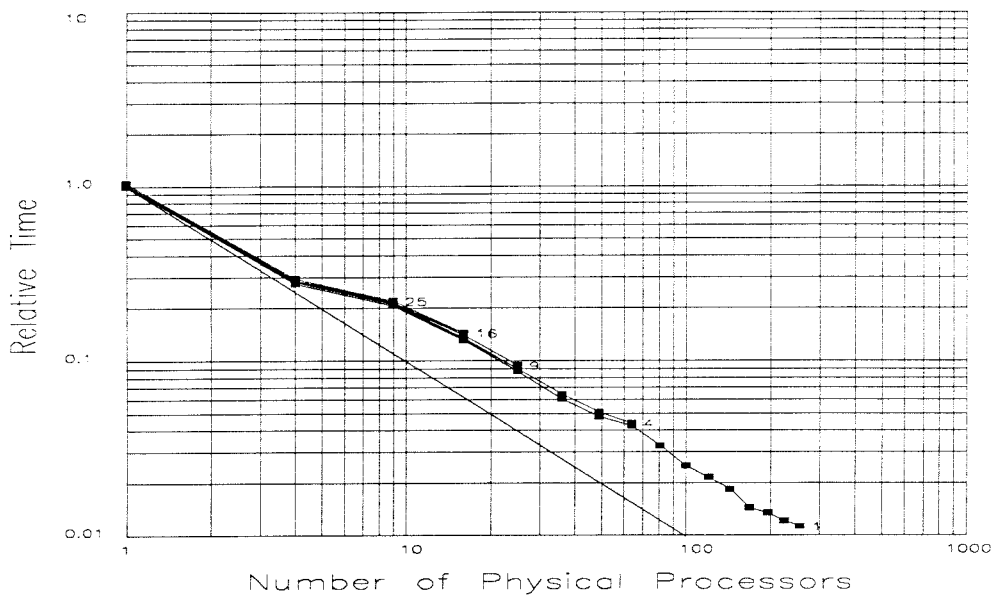


Figure A.26: X-Wing Virtual Simulation Results (xwf8) *(cont.)*



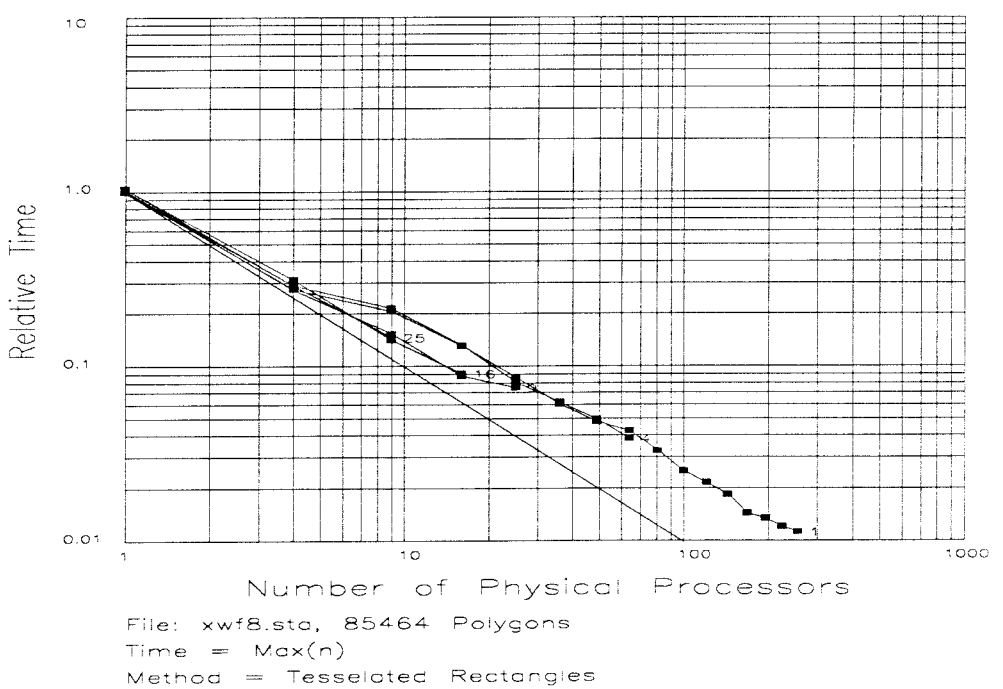
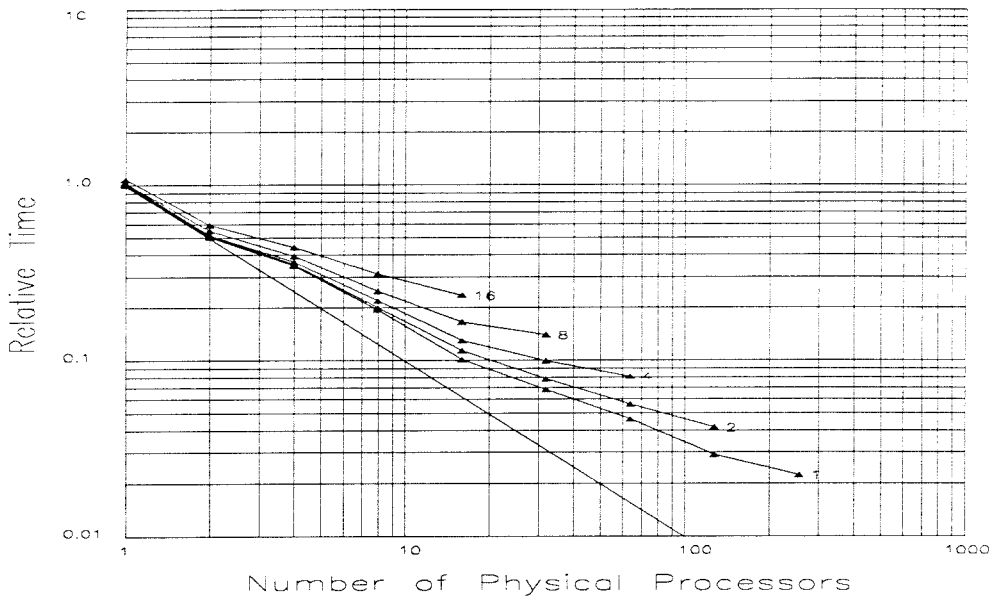


Figure A.26: X-Wing Virtual Simulation Results (xwf8) *(cont.)*



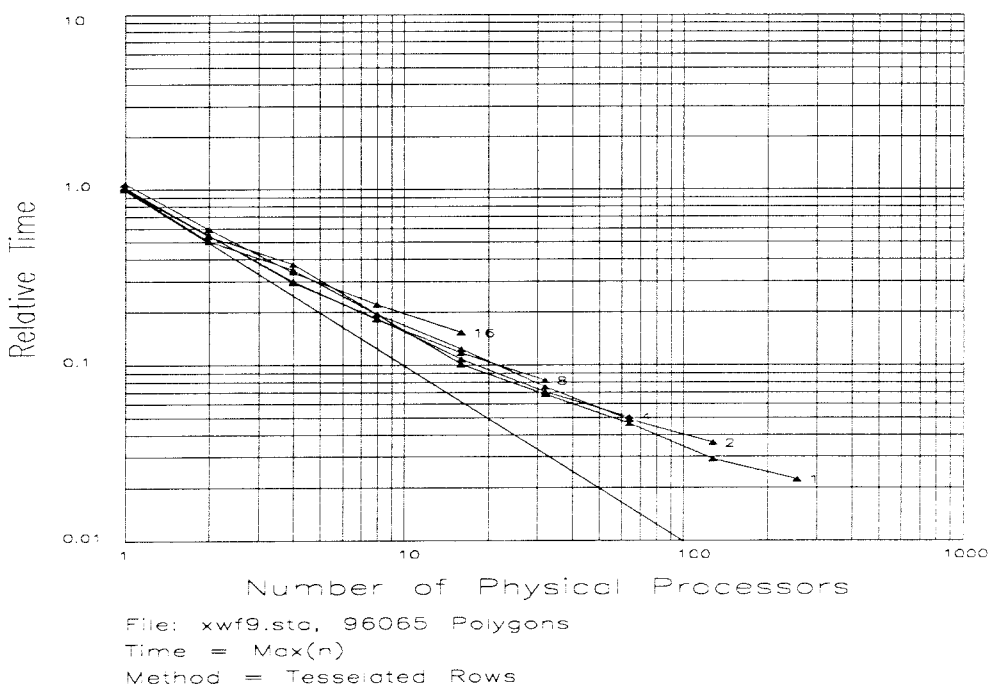
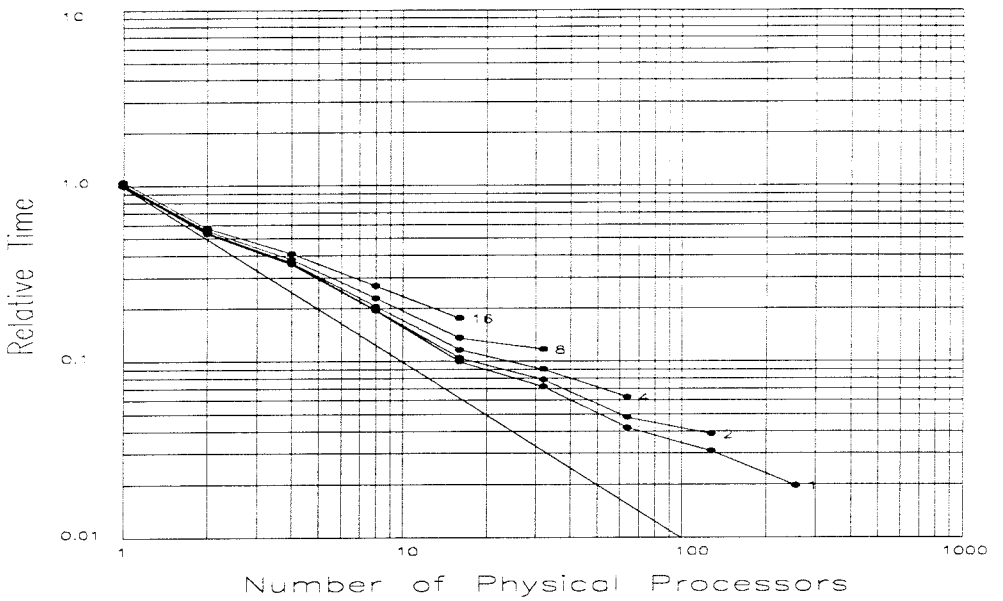


Figure A.27: X-Wing Virtual Simulation Results (xwf9)



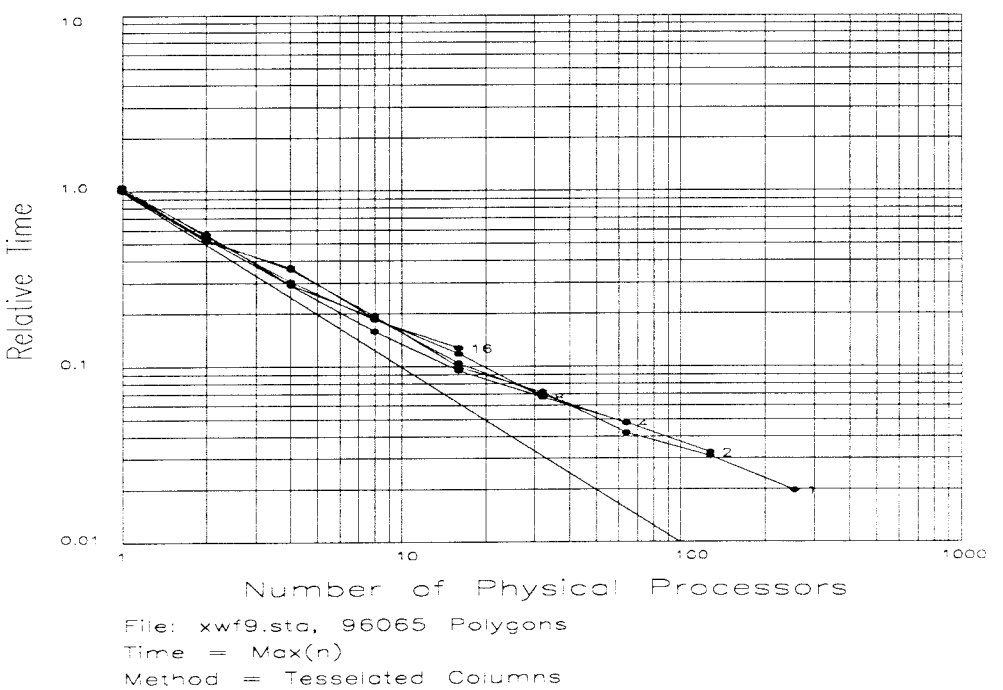
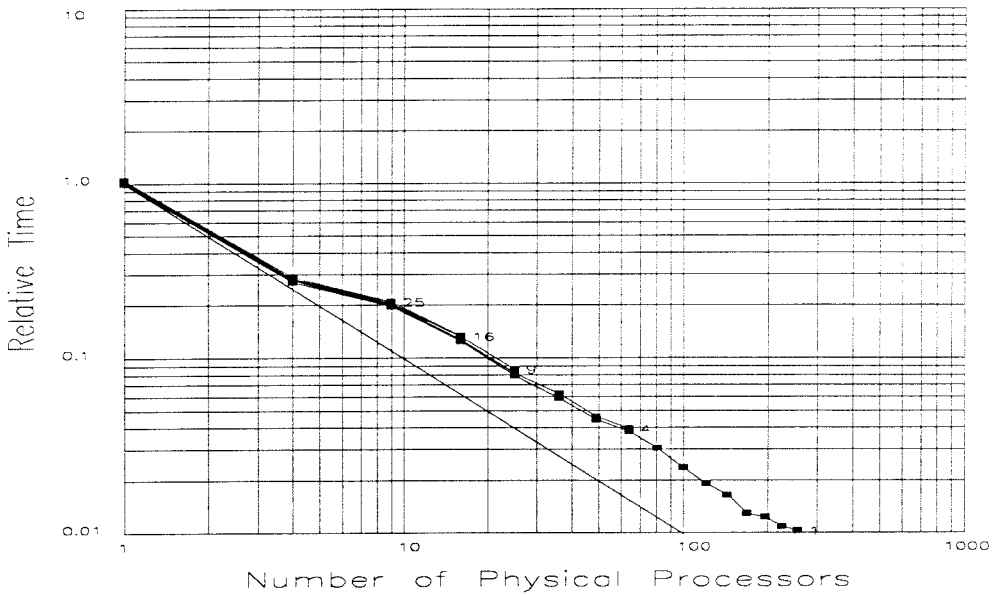


Figure A.27: X-Wing Virtual Simulation Results (xwf9) *(cont.)*



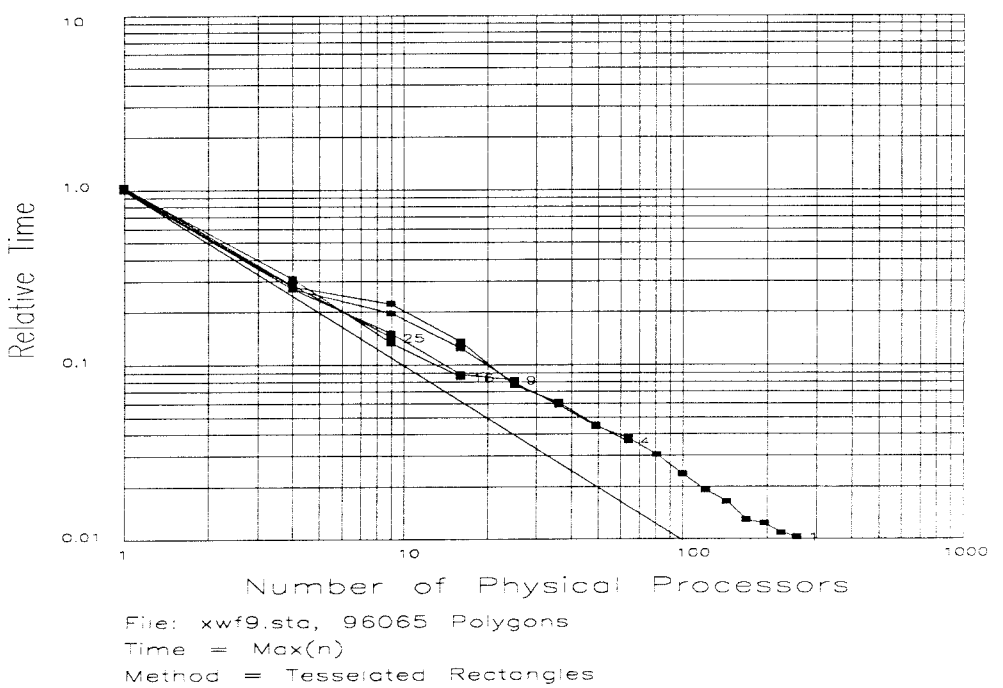


Figure A.27: X-Wing Virtual Simulation Results (xwf9) *(cont.)*

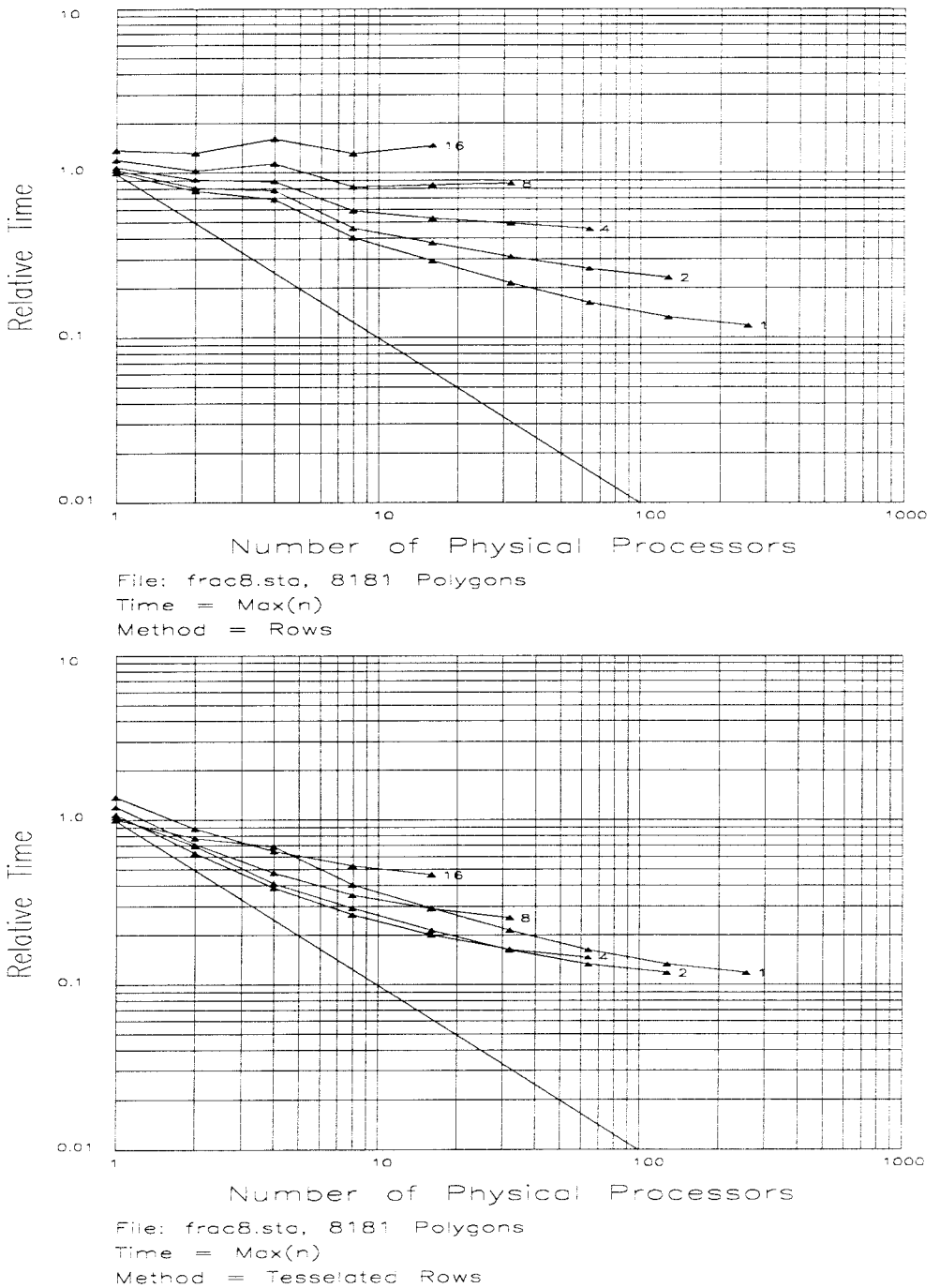
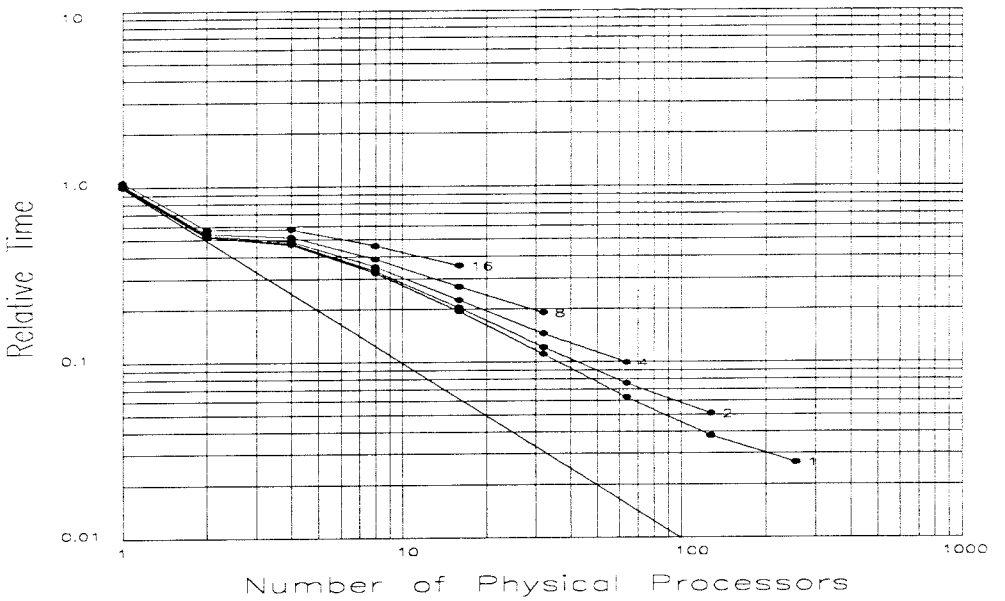


Figure A.28: Fractal Landscape Virtual Simulation Results (frac8)



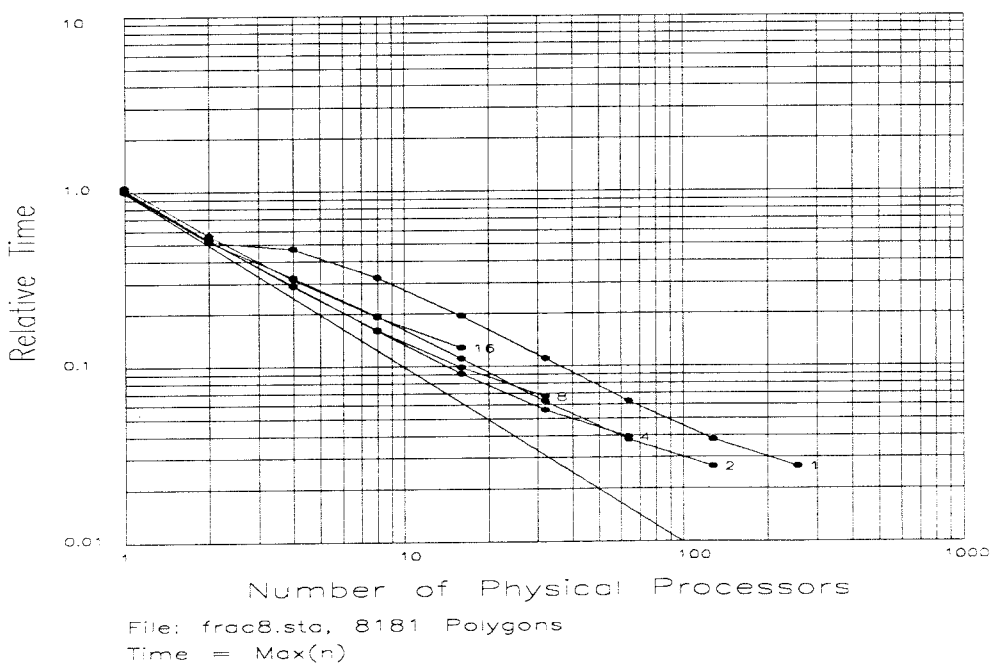
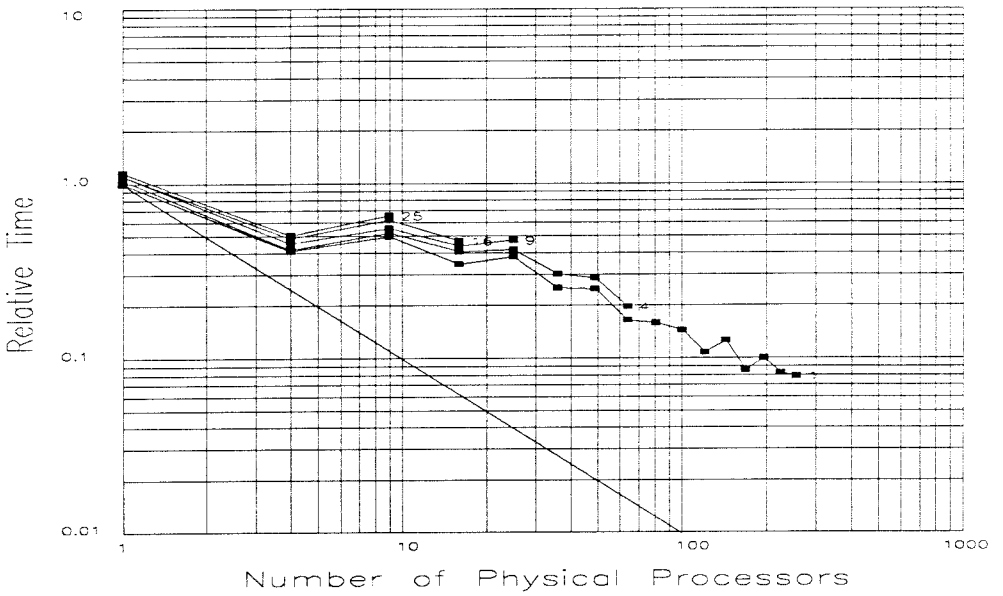


Figure A.28: Fractal Landscape Virtual Simulation Results (frac8) *(cont.)*



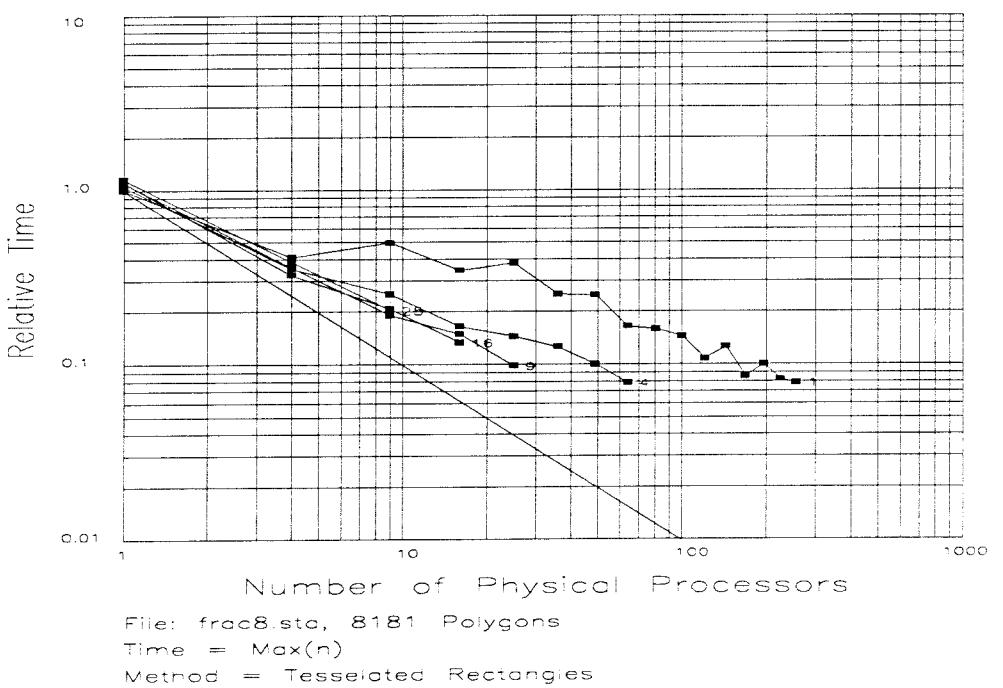
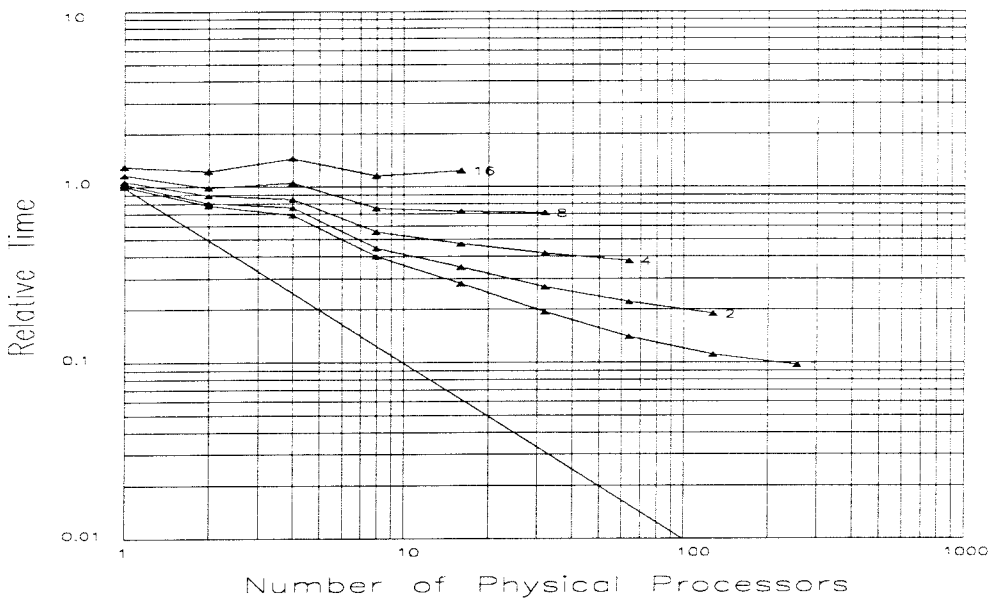


Figure A.28: Fractal Landscape Virtual Simulation Results (frac8) *(cont.)*



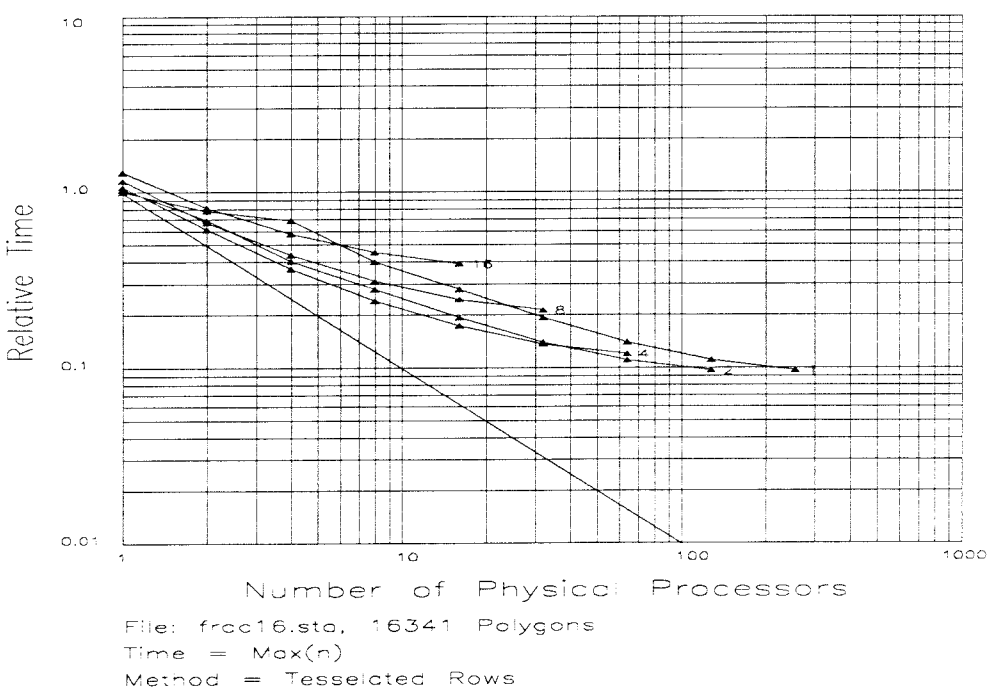
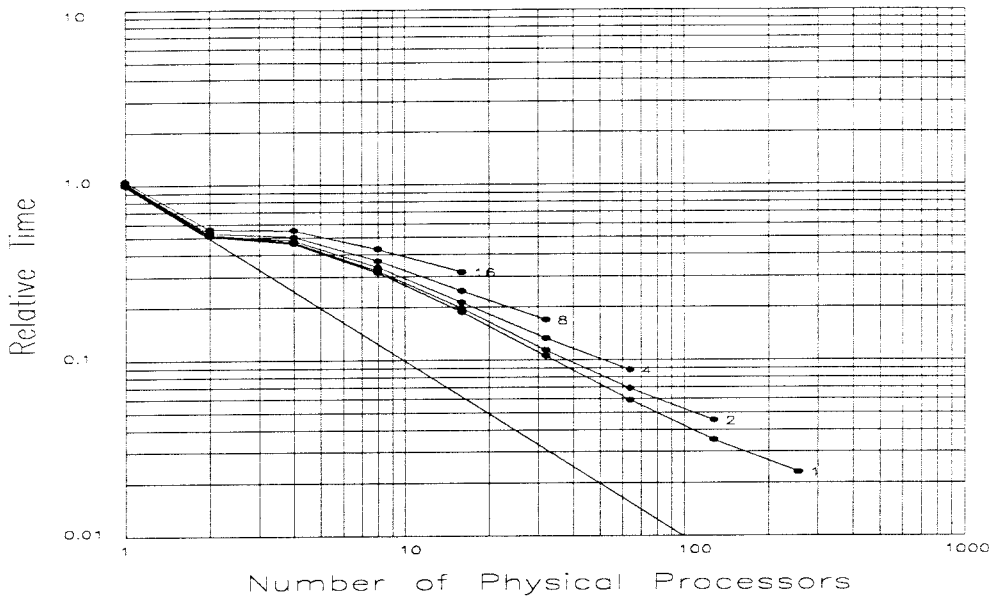


Figure A.29: Fractal Landscape Virtual Simulation Results (frac16)



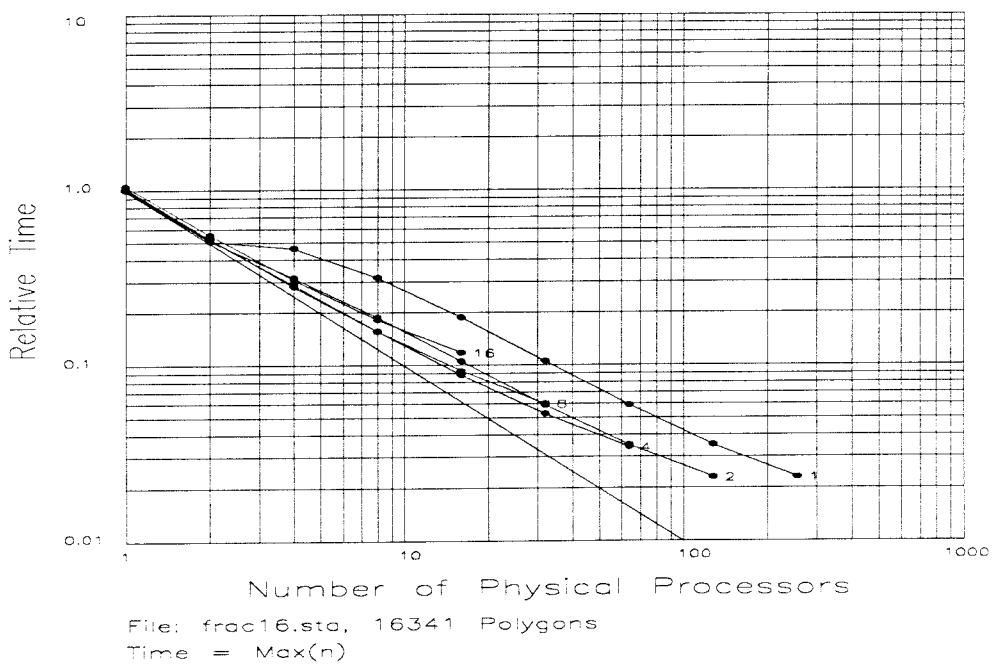


Figure A.29: Fractal Landscape Virtual Simulation Results (frac16) *(cont.)*

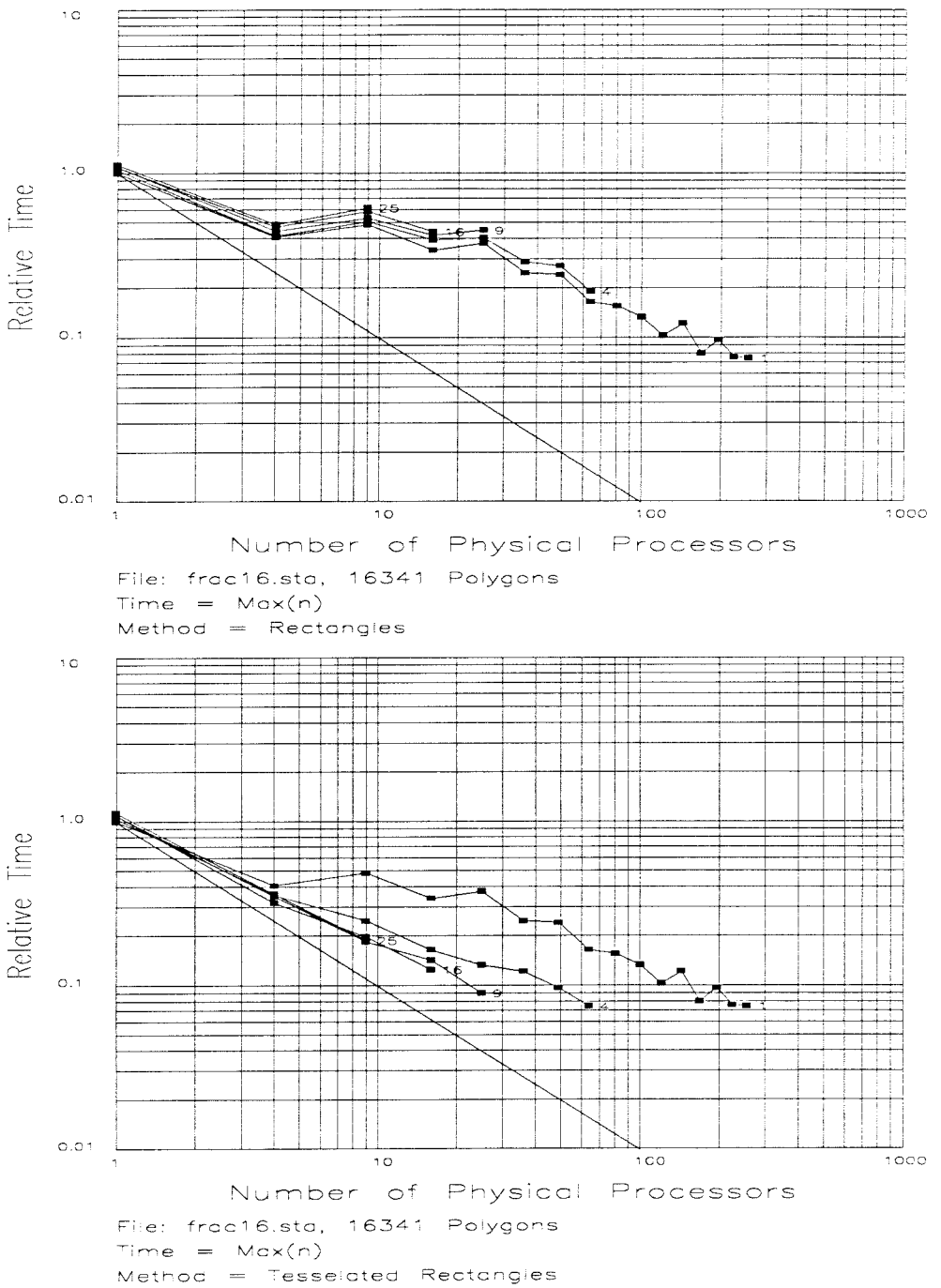


Figure A.29: Fractal Landscape Virtual Simulation Results (frac16) *(cont.)*

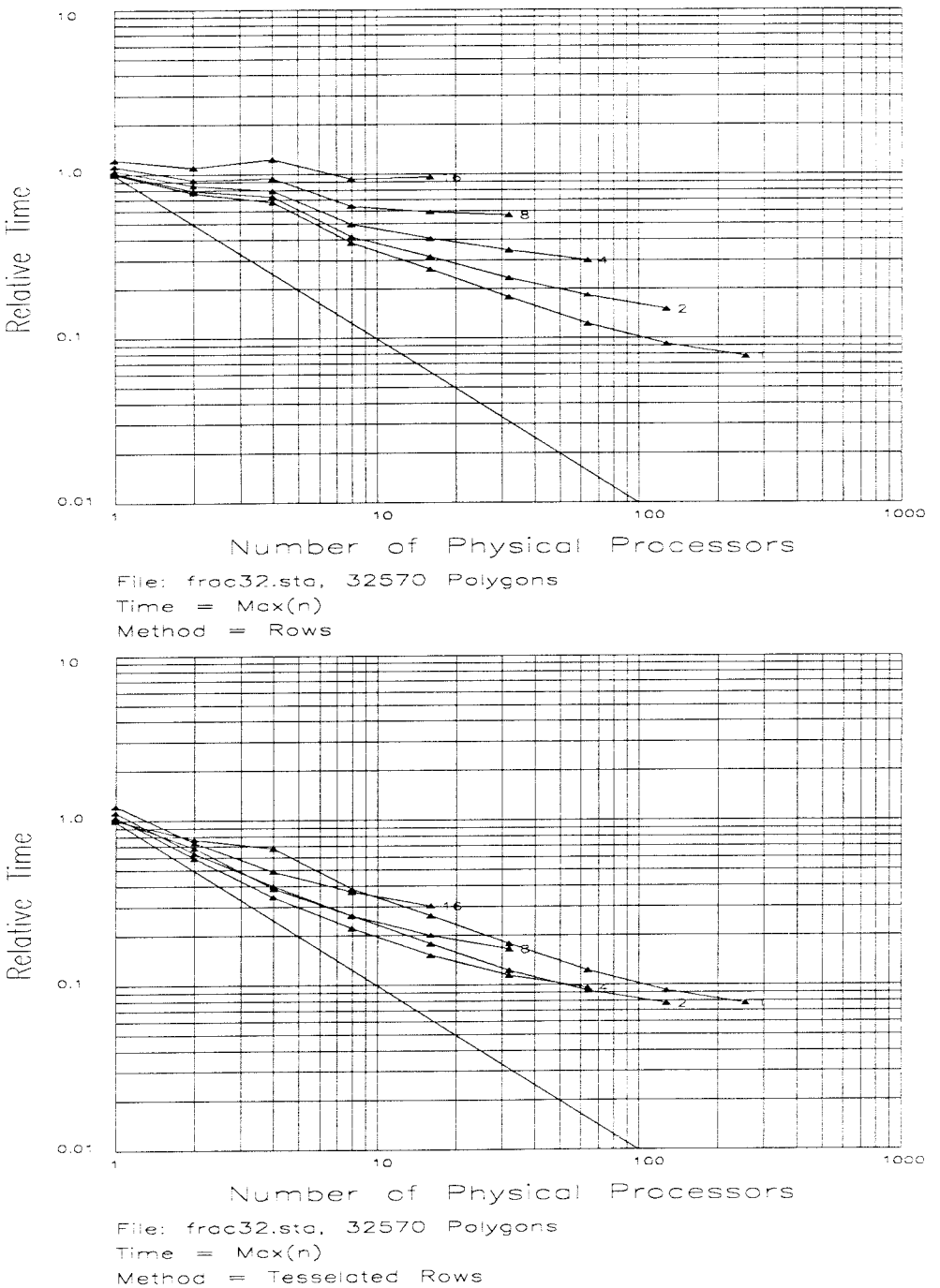


Figure A.30: Fractal Landscape Virtual Simulation Results (frac32)

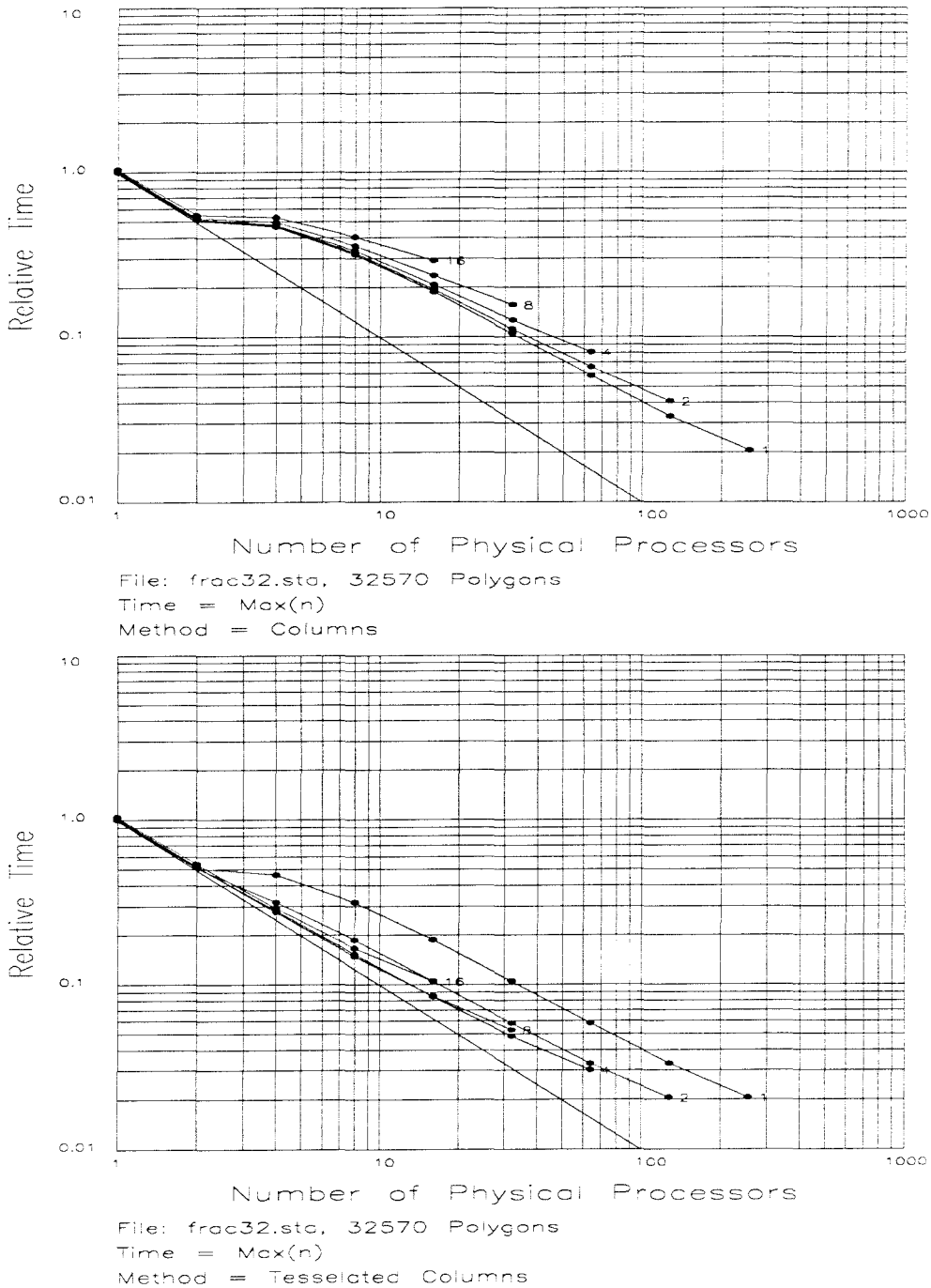


Figure A.30: Fractal Landscape Virtual Simulation Results (frac32) *(cont.)*

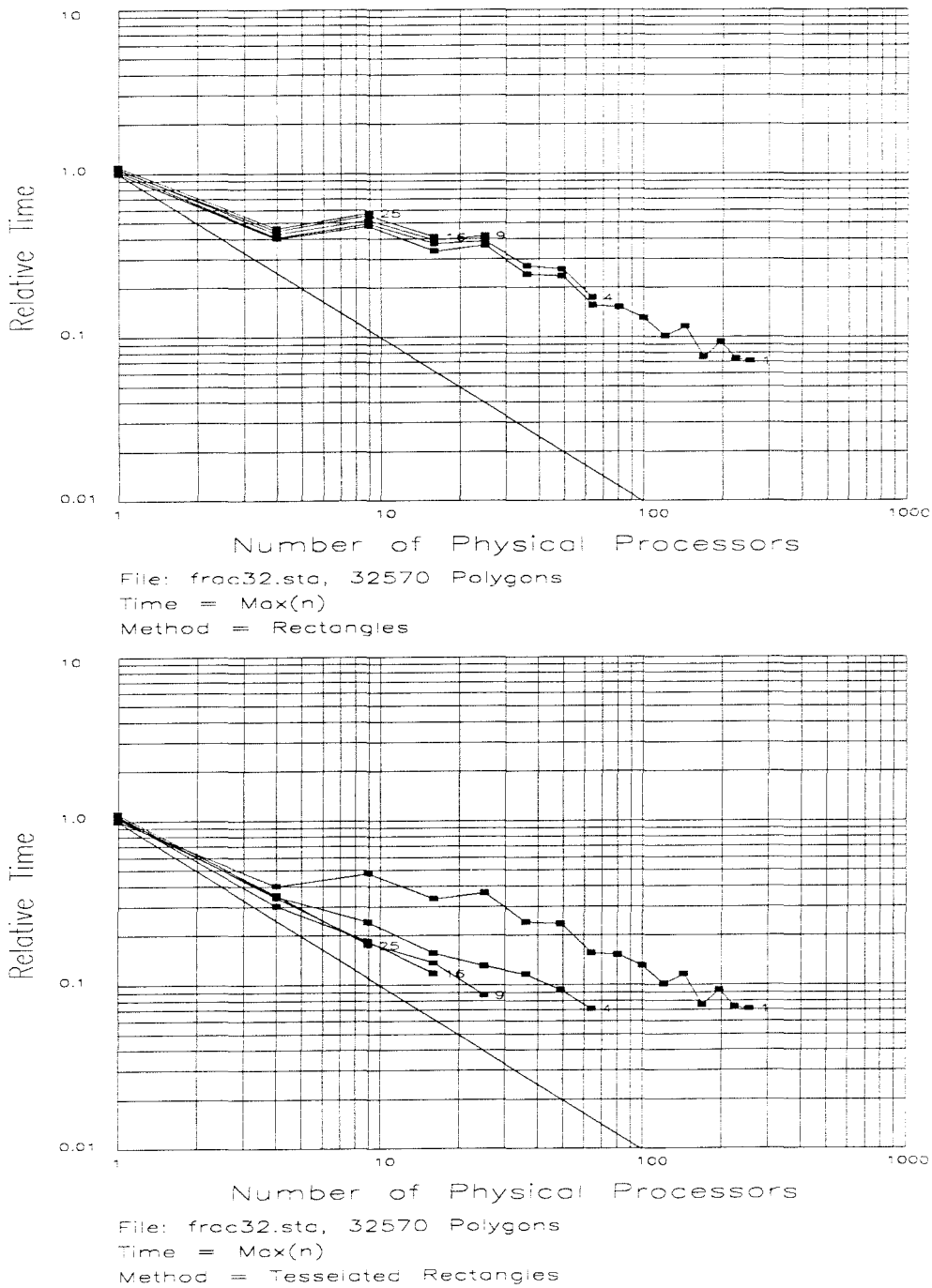


Figure A.30: Fractal Landscape Virtual Simulation Results (frac32) *(cont.)*

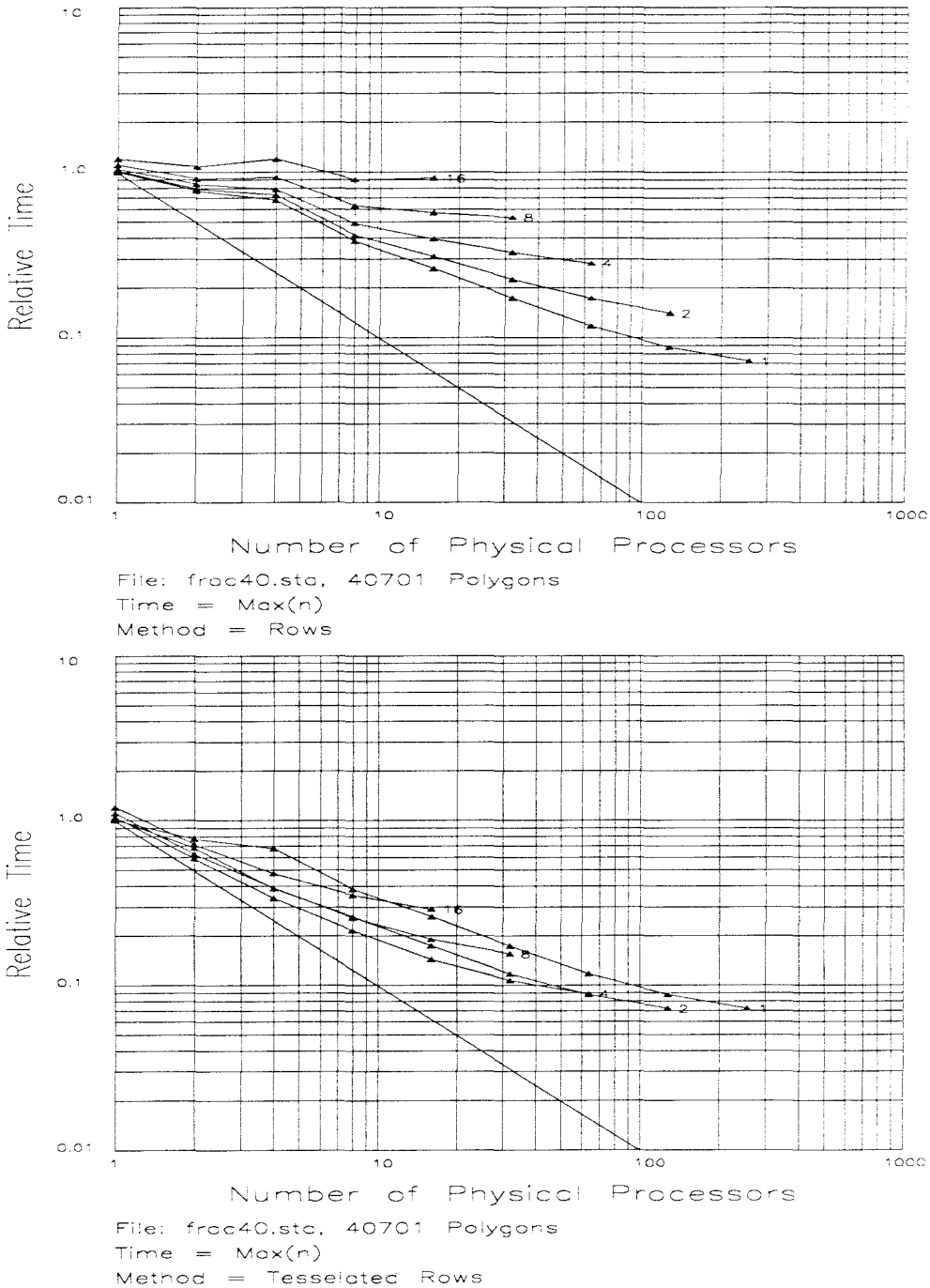


Figure A.31: Fractal Landscape Virtual Simulation Results (frac40)

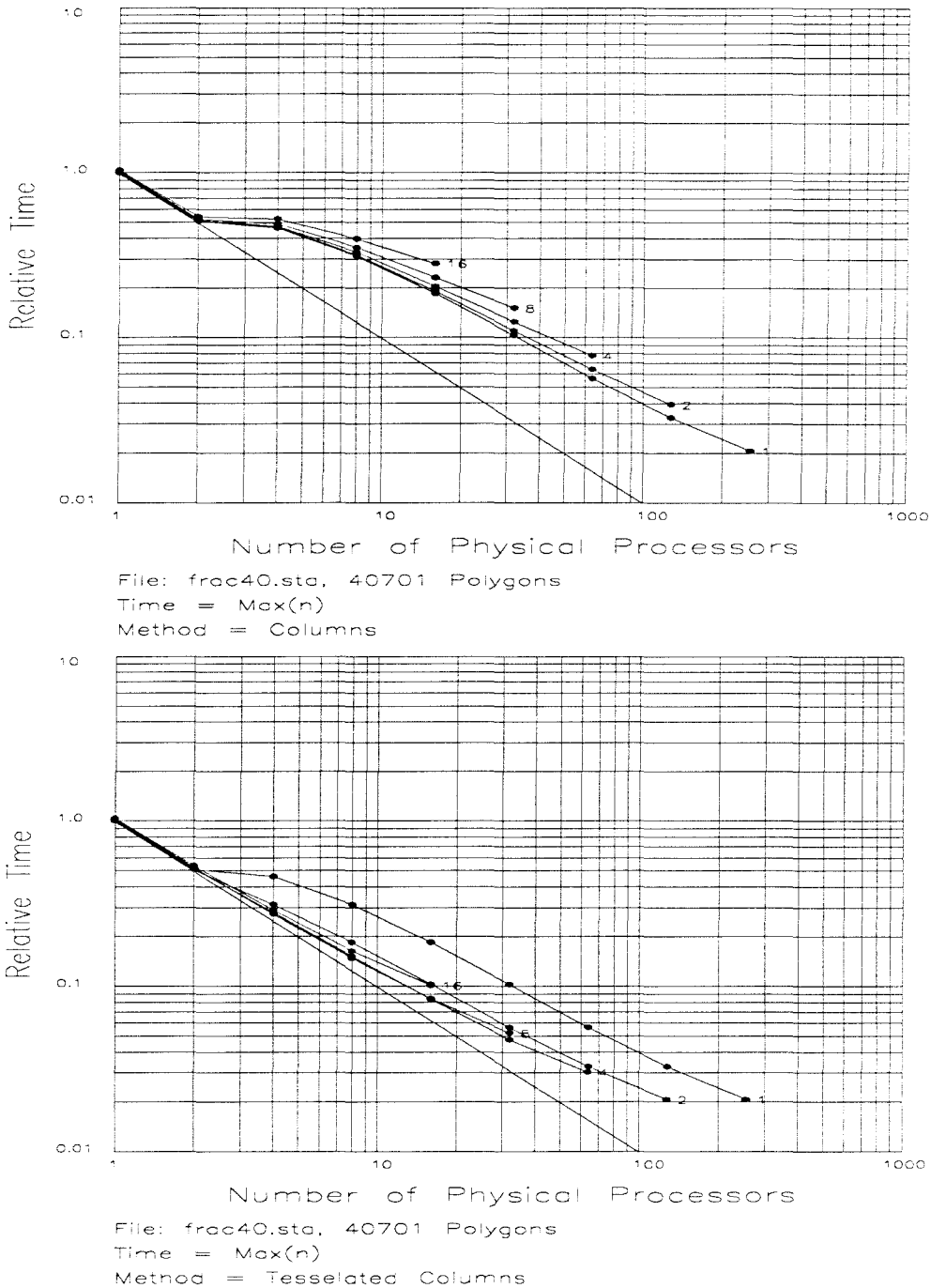


Figure A.31: Fractal Landscape Virtual Simulation Results (frac40) *(cont.)*

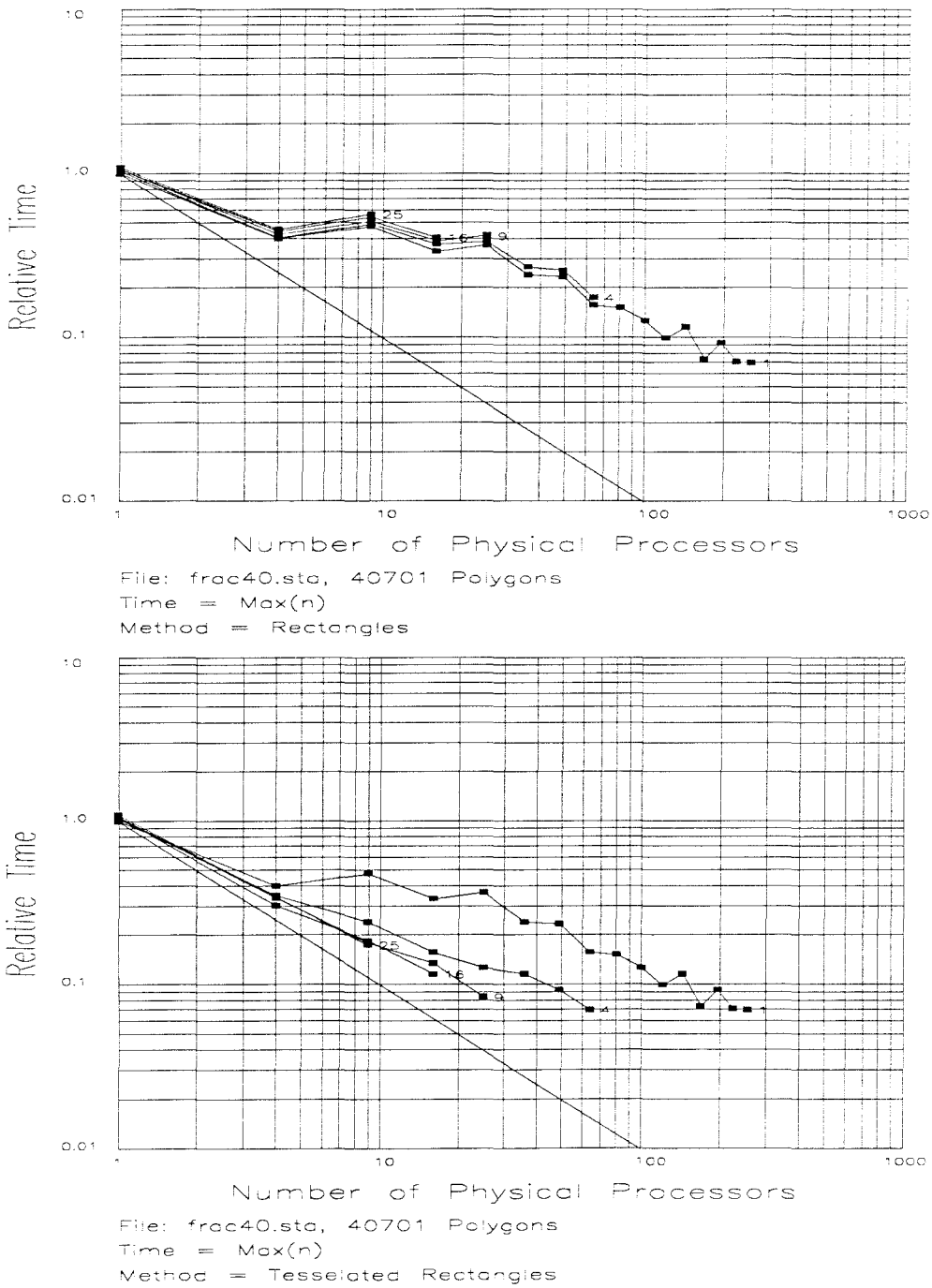


Figure A.31: Fractal Landscape Virtual Simulation Results (frac40) *(cont.)*

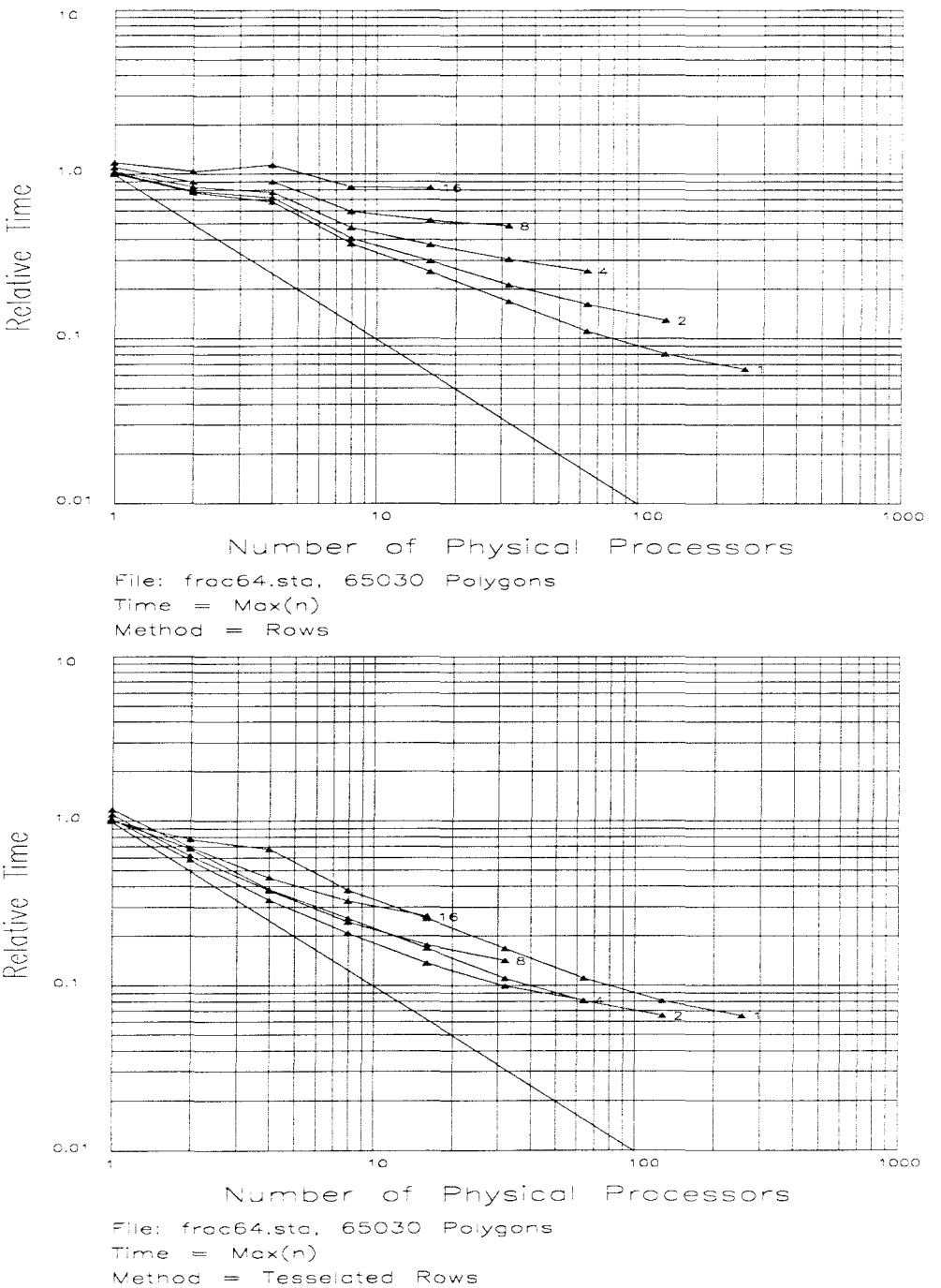


Figure A.32: Fractal Landscape Virtual Simulation Results (frac64)

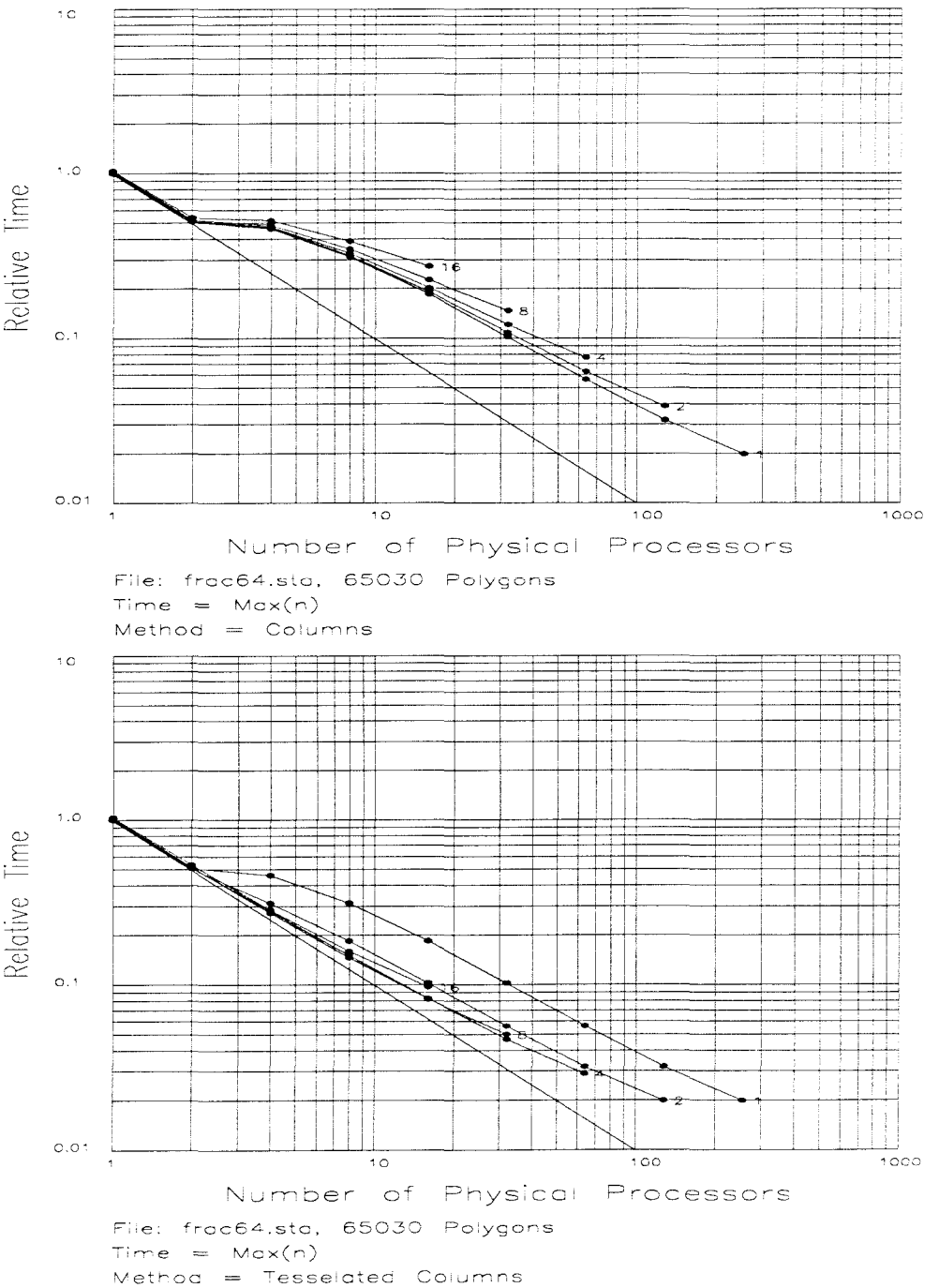


Figure A.32: Fractal Landscape Virtual Simulation Results (frac64) *(cont.)*

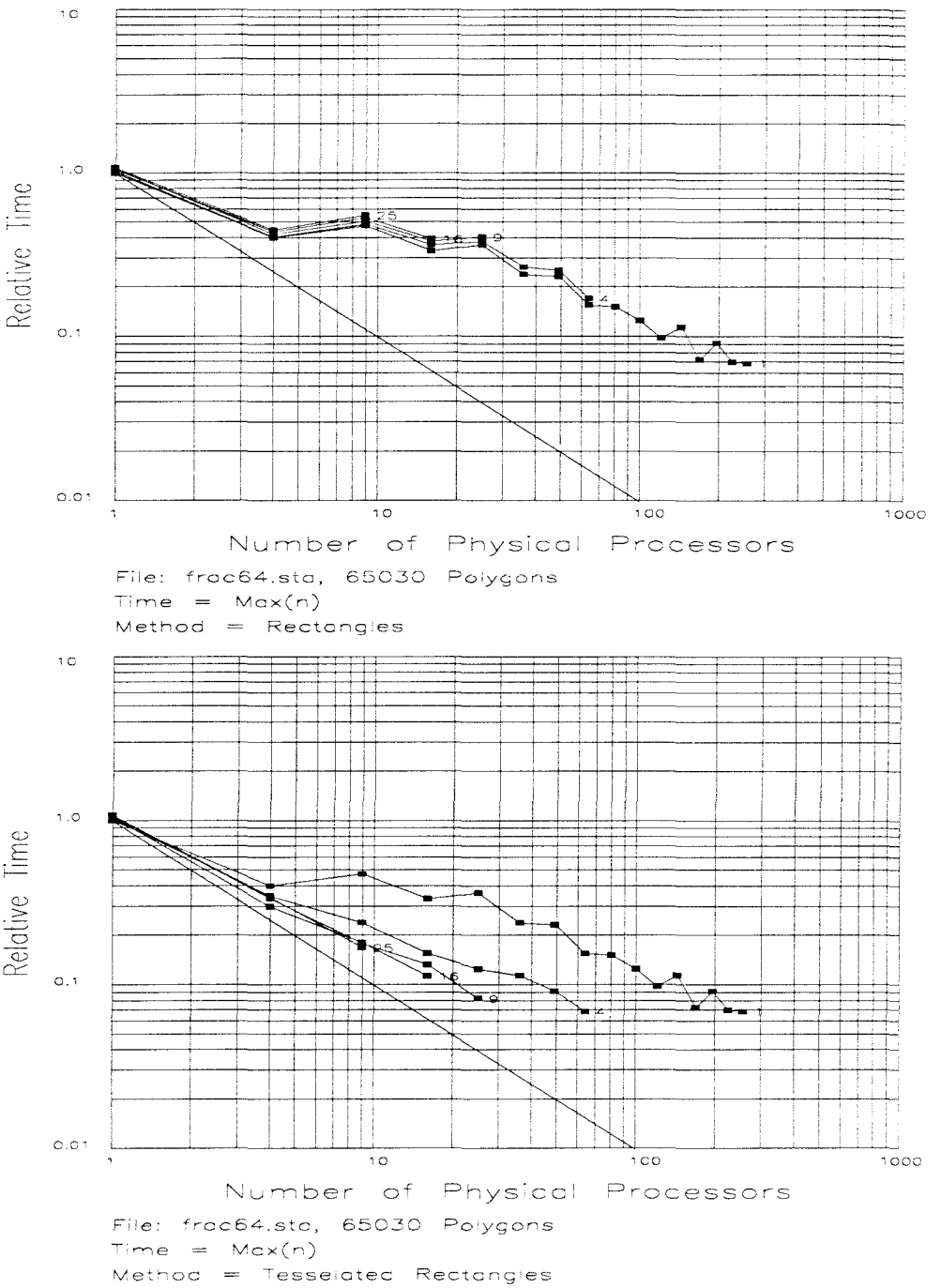


Figure A.32: Fractal Landscape Virtual Simulation Results (frac64) *(cont.)*

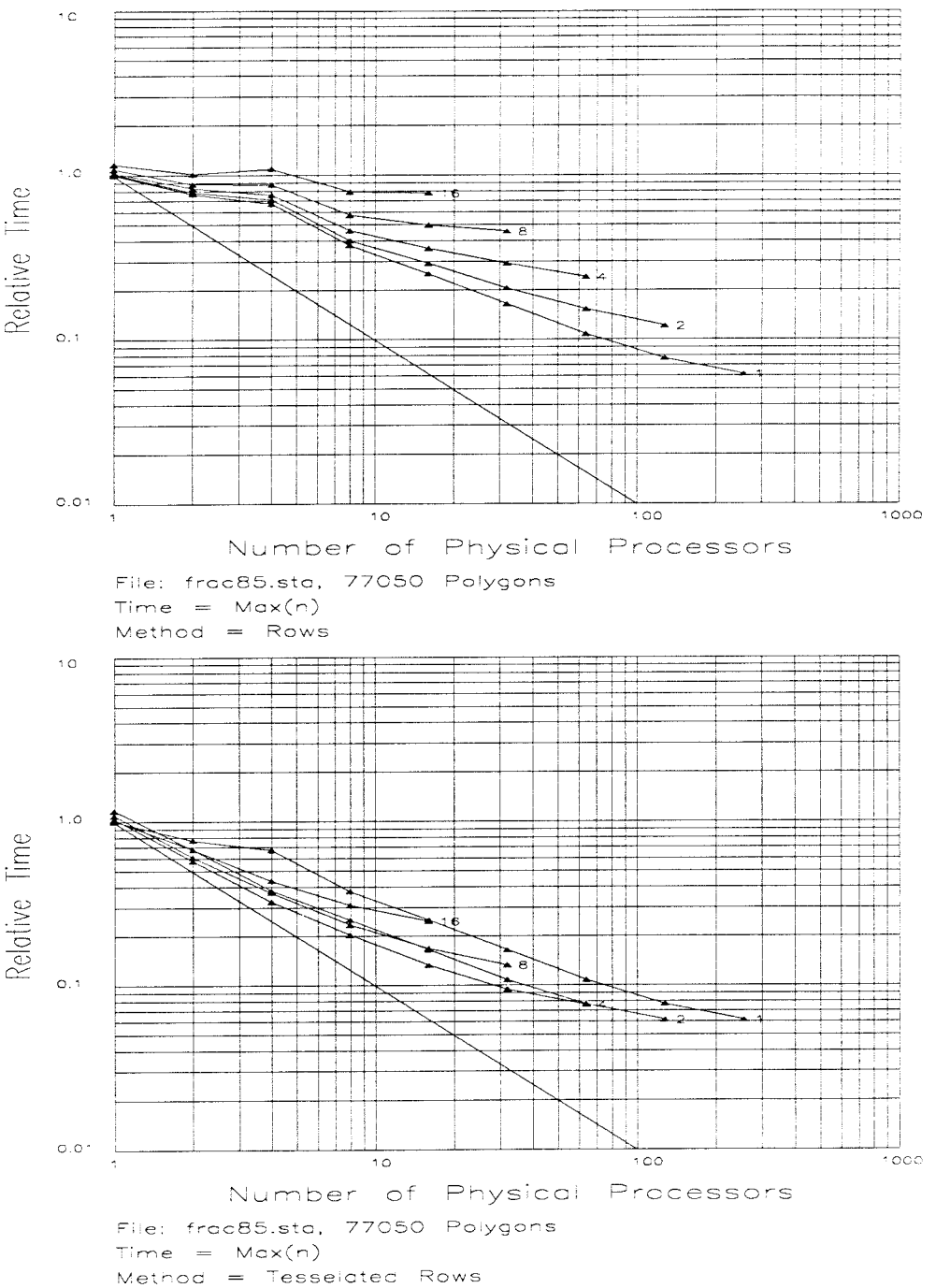


Figure A.33: Fractal Landscape Virtual Simulation Results (frac85)

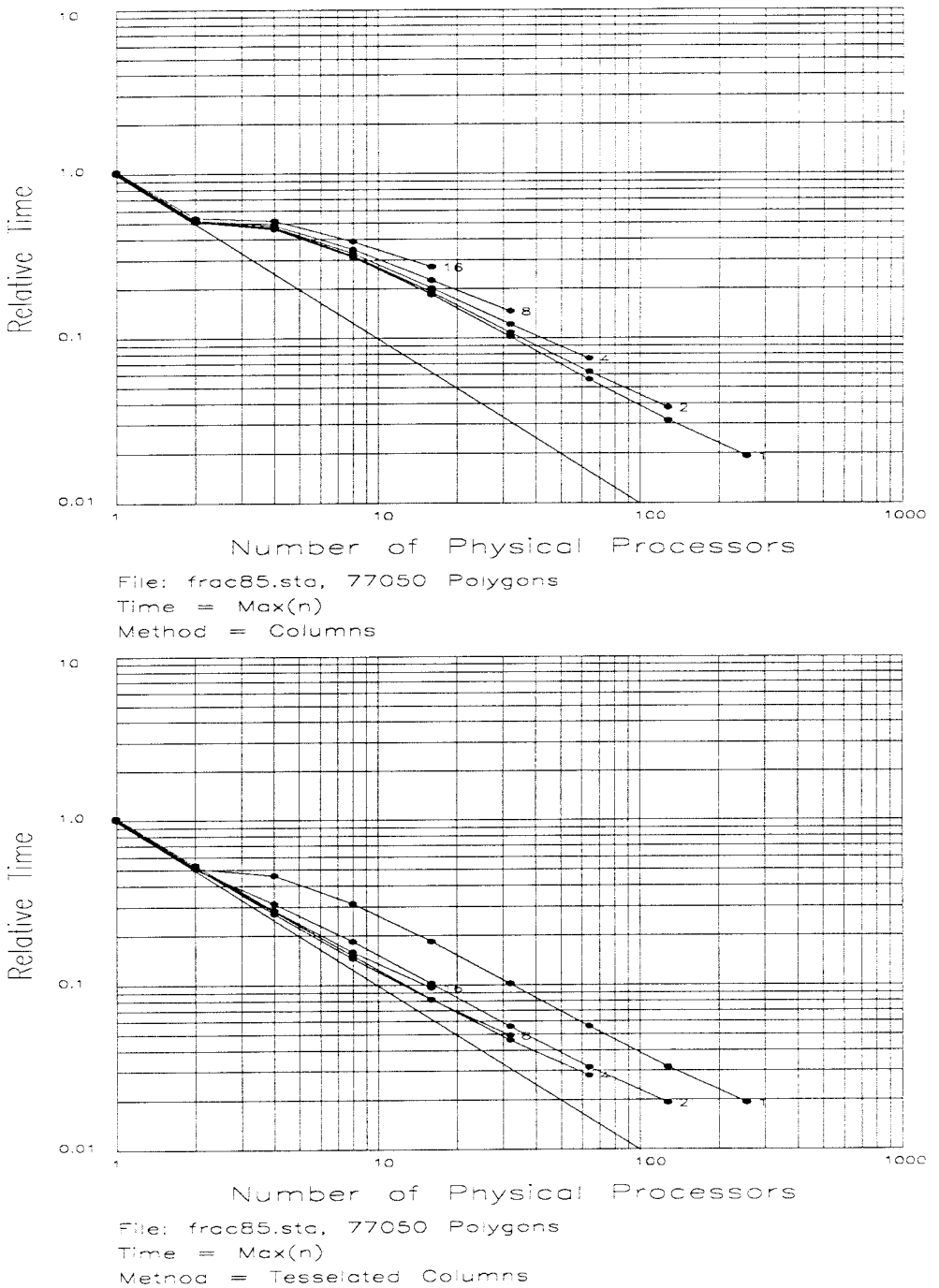


Figure A.33: Fractal Landscape Virtual Simulation Results (frac85) *(cont.)*

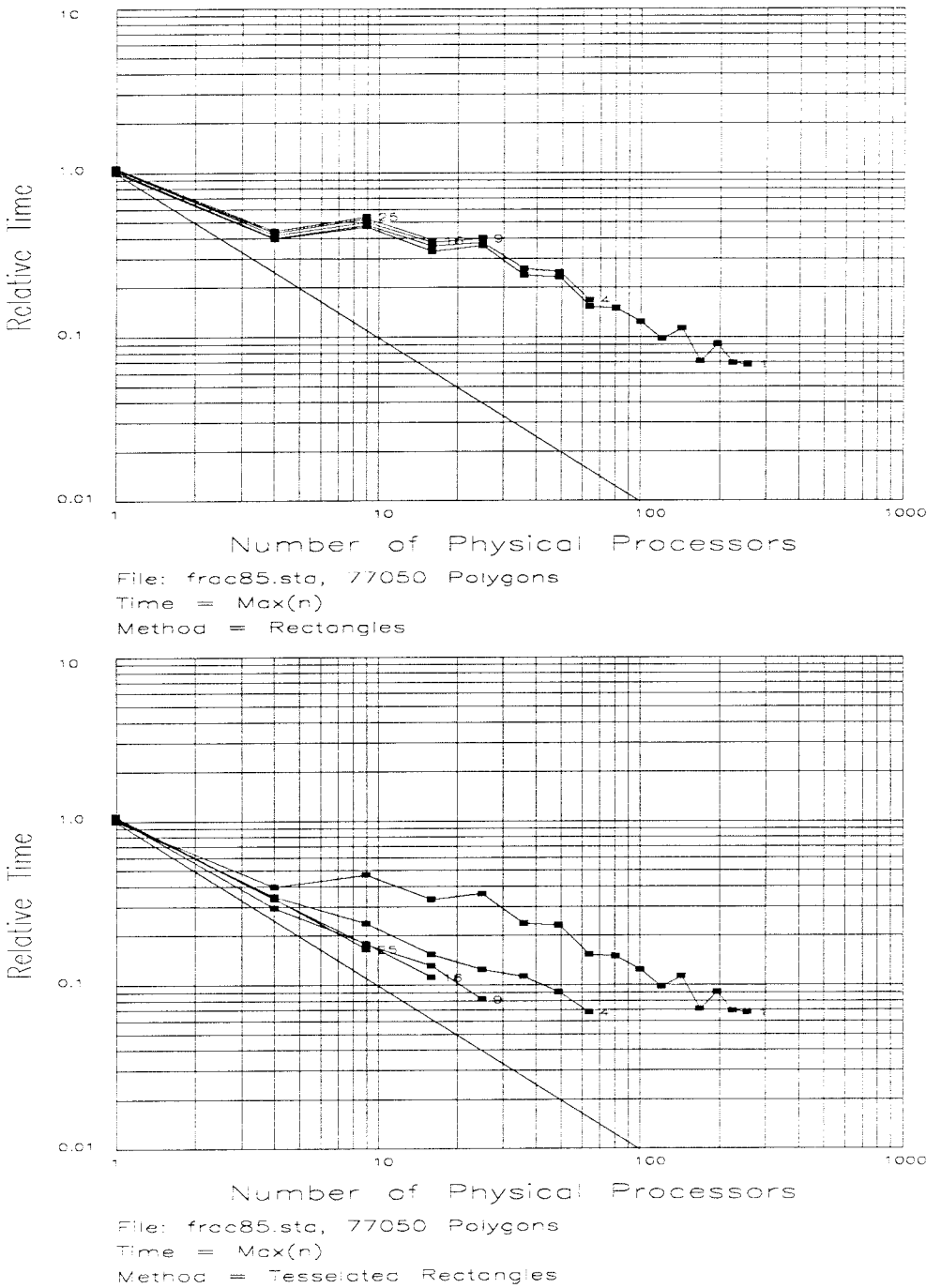


Figure A.33: Fractal Landscape Virtual Simulation Results (frac85) *(cont.)*

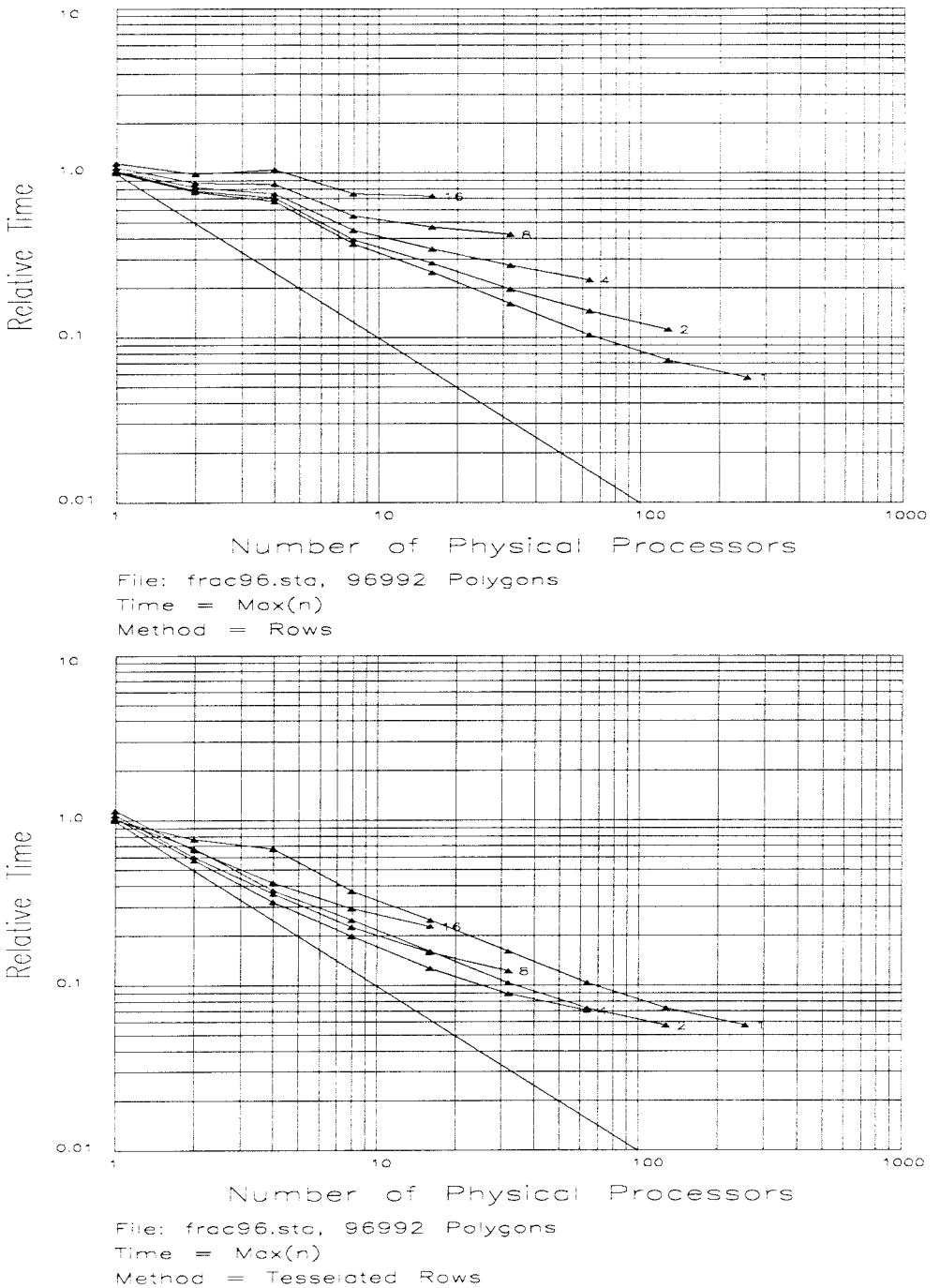


Figure A.34: Fractal Landscape Virtual Simulation Results (frac96)

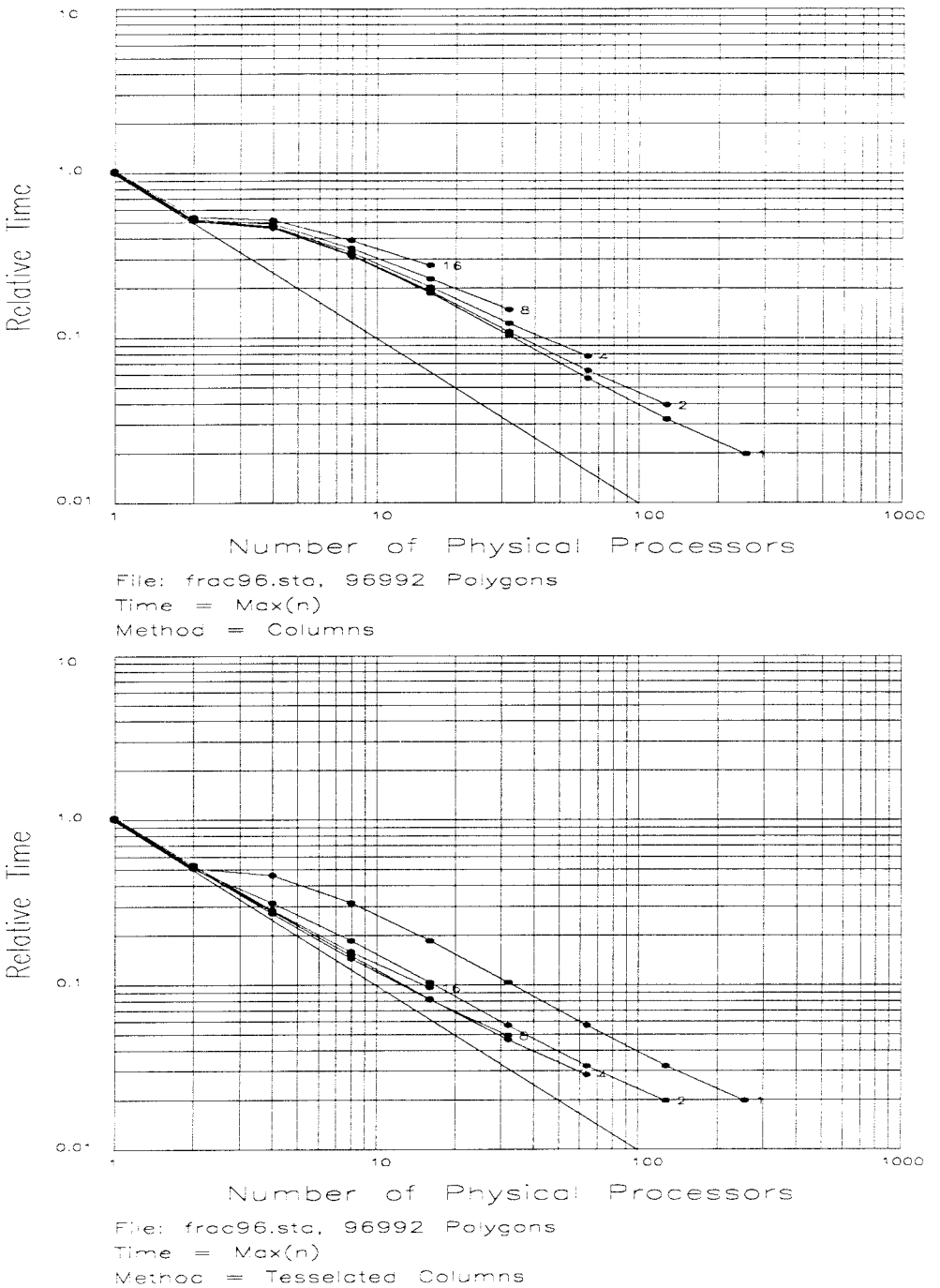


Figure A.34: Fractal Landscape Virtual Simulation Results (frac96) *(cont.)*

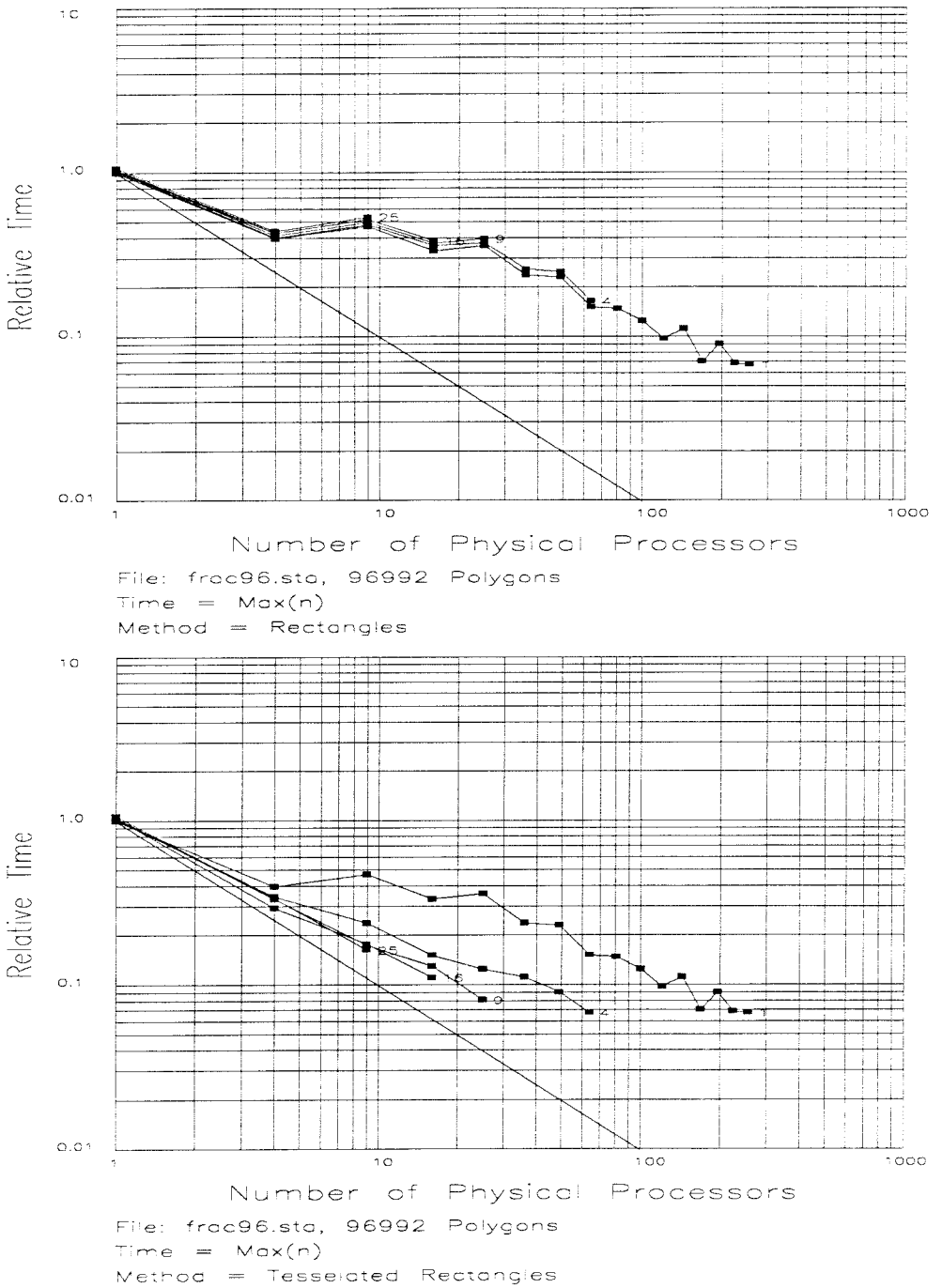


Figure A.34: Fractal Landscape Virtual Simulation Results (frac96) *(cont.)*

References

[APPEL68] Appel, A., *Some Techniques for Shading Machine Renderings of Solids,* SJCC 1968, Thompson Books, Washington, D.C., pp. 37–45.

[ATHERT78] Atherton, P., Weiler, K., and Greenberg, D., "Polygon Shadow Generation" SIGGRAPH '78 Proceedings, published as *Computer Graphics,* 12 (3), August 1978, pp. 275–281.

[BLINN76] Blinn, J.F. and Newell, M.E., "Texture and Reflection in Computer Generated Images," *Communications of the ACM,* 19 (10), October 1976, pp. 542–547.

[BLINN82] Blinn, J.F., Personal Communications, 1982.

[BROTMA84] Brotman, L.S. and Badler, N.I., "Generating Soft Shadows with a Depth Buffer Algorithm," *IEEE Computer Graphics and Applications,* pp. 5–12.

[BUI-TU75] Bui-Tuong, P., "Illumination for Computer Generated Pictures," *Communications of the ACM,* 18 (6), June 1975, pp. 311–317.

[CARPEN84] Carpenter, L., "The A-Buffer, an Antialiased Hidden Surface Method," SIGGRAPH '84 Proceedings, published as *Computer Graphics,* 18 (3), July 1984, pp. 103–108.

[CATMUL74] Catmull, E., *A Subdivision Algorithm for Computer Display of Curved Surfaces,* University of Utah Computer Science Department, UTEC–CSc–74–133, December 1974.

[CATMUL80] Catmull, E. and Smith, A.R., "3-D Transformations of Images in Scanline Order," SIGGRAPH '80 Proceedings, published as *Computer Graphics,* 14 (3), July 1980, pp. 279–285.

[CATMUL84] Catmull, E., "An Analytic Visible Surface Algorithm for Independent Pixel Processing," SIGGRAPH '84 Proceedings, published as *Computer Graphics,* 18 (3), July 1984, pp. 109–115.

[CLARKE80] Clarke, J. and Hanna, M., "Distributed Processing in a High-Performance Smart Image Memory," *VLSI Design*, 4th Quarter, 1980, pp. 40–45.

[COHEN80] Cohen, D. and Demetrescu, S., *A VLSI Approach to Computer Image Generation,* Technical Report, University of Southern California, Information Sciences Institute, Los Angeles, 1980.

[CROW77A] Crow, F.C., "Shadow Algorithms for Computer Graphics," SIGGRAPH '77 Proceedings, published as *Computer Graphics,* 11 (2), Summer 1977, pp. 242–247.

[CROW77B] Crow, F.C., "The Aliasing Problem in Computer-generated Shaded Images," *Communications of the ACM,* 20 (11), Nov. 1977, pp. 799–805.

[DEMETR83] Demetrescu, S., *High Speed Image Rasterization Using a Highly Parallel Smart Bulk Memory,* Computer Systems Laboratory, Stanford University, Technical Report No. 83–244, June 1983.

[DIPPÉ84] Dippé, M., and Swensen, J., "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis," SIGGRAPH '84 Proceedings, published as *Computer Graphics,* 18 (3), July 1984, pp. 149–158.

[ELECTR85] "Philips gives boost to performance of multimode fiber," ElectronicsWeek, 58 (18), May 6, 1985, pp. 25–28.

[FUCHS79] Fuchs, H. and Johnson, B.W., "An Expandable Multiprocessor Architecture for Video Graphics," Proceedings of the 6th Annual ACM-IEEE Symposium on Computer Architecture, 1979, pp. 58–67.

[FUCHS81] Fuchs, H. and J. Poulton, "PIXEL-PLANES: A VLSI Oriented Design for a Raster Graphics Engine," *VLSI Design*, 2 (3), 1981, pp. 20–28.

[FUCHS82] Fuchs, H., Poulton, J., Paeth, A., and Bell, A., "Developing PIXEL-PLANES, A Smart Memory-based Raster Graphics System," Proceedings of the 1982 MIT Conference on Advanced Research in VLSI, Artech House, pp. 137–146.

[GABRIE82] Gabriel, S., Personal Communications, 1982.

[GOURAU71] Gouraud, H., "Continuous Shading of Curved Surfaces," *IEEE Transactions on Computers,* C–20 (6), June 1971, pp. 623–628.

[KAJIYA81] Kajiya, J., and Ullner, M., "Filtering High Quality Text for Display on Raster Scan Devices," SIGGRAPH '81 Proceedings, published as *Computer Graphics,* 15 (3), August 1981, pp. 7–15.

[KAPLAN79] Kaplan, M. and Greenberg, D., "Parallel Processing Techniques for Hidden Surface Removal," SIGGRAPH '79 Proceedings, published as *Computer Graphics,* 13 (2), August 1979, pp. 300–307.

[KEDEM84] Kedem, G. and Ellis, J.L., *Computer Structures for Curve-Solid Classification in Geometric Modeling,* Production Automation Project, University of Rochester, May 1984.

[MEAD80] Mead, C.A., and Conway, L.A., *Introduction to VLSI Systems,* Addison-Wesley, 1980.

[MOHSEN79] Mohsen, A., "Device and Circuit Design for VLSI," Caltech Conference on VLSI Proceedings, January 1979, pp. 31–54.

[MOORE79] Moore, G.E., "Are We Really Ready for VLSI?," Caltech Conference on VLSI Proceedings, January 1979, pp. 3–14.

[PARKE80] Parke, F.I., "Simulation and Expected Performance Analysis of Multiple Processor Z-Buffer Systems," SIGGRAPH '80 Proceedings, published as *Computer Graphics,* 14 (3), July 1980, pp. 48–56.

[POULTO85] Poulton, J., Fuchs, H., Austin, J.D., Eyles, J.G., Heinecke, J, Hsieh, C.H., Goldfeather, Hultquist, J.P., and Sprach, S., "PIXEL-PLANES: Building a VLSI-Based Graphics System,"

Proceedings of the 1985 Chapel Hill Conference on VLSI, May 1985.

[RIDEOU81] Rideout, V.L., "Trends in Silicon Processing," Caltech Conference on VLSI Proceedings, January 1981, pp. 65–110.

[SCHACH83] Schachter, B.J., *ed. Computer Image Generation,* Wiley-Interscience, New York, 1983.

[SCHUMA80] Schumacher, R.A., "A New Visual Surface Architecture," Proceedings of the Second Interservice/Industry Training Equipment Conference, November 1980, pp. 1–8.

[SUTHER74] Sutherland, I.E., Sproul, R.F., and Schumaker, R.A., "A Characterization of Ten Hidden Surface Algorithms," *Computing Surveys,* 6 (1), March 1974, pp. 1–55.

[ULLNER83] Ullner, M.K., *Parallel Machines for Computer Graphics,* Ph.D. thesis, California Institute of Technology, 1983.

[WATKIN70] Watkins, G.S., *A Real-Time Visible Surface Algorithm,* University of Utah Computer Science Department, UTEC–CSs–70–101, June 1970, NTIS AD–762 004.

[WARNOC69] Warnock, J., *A Hidden-Surface Algorithm for Computer Generated Half-Tone Pictures,* University of Utah Computer Science Department, TR–4–15, 1969, NTIS AD–753 671.

[WEILER78] Weiler, K., *Hidden Surface Removal Using Polygon Area Sorting,* M.S. Thesis, Program of Computer Graphics, Cornell University, January 1978.

[WEINBE82] Weinberg, R. A., *An Architecture for Parallel Processing Image Synthesis with Anti-aliasing,* Ph.D. Thesis, University of Minnesota, 1982.

[WHELAN82] Whelan, D.S., "A Rectangular Area Filling Display System Architecture," SIGGRAPH '82 Proceedings, published as *Computer Graphics,* 16 (3), 1982.

[WHITTE80] Whitted, T., "An Improved Illumination Model for Shaded Display," *Communications of the ACM,* 23 (6), June 1980, pp. 343–349.

[WILLIA78] Williams, L., "Casting Curved Shadows on Curved Surfaces," SIGGRAPH '78 Proceedings, published as *Computer Graphics,* 12 (3), August 1978, pp. 270–274.