PARALLEL IMAGE GENERATION USING THE Z-BUFFERING ALGORITHM ON A MEDIUM GRAINED DISTRIBUTED MEMORY MODEL COMPUTER

By

ERIC MARTIN BLAZEK

Bachelor of Science

Oklahoma State University

Stillwater, Oklahoma

1988

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 1991

Thesis
1991
B61Ep
cop.2

PARALLEL IMAGE GENERATION USING THE Z-BUFFERING ALGORITHM ON A MEDIUM GRAINED DISTRIBUTED MEMORY MODEL COMPUTER

Thesis Approved:

Blayne E. Mayfield

Thesis Adviser

[illegible]

[illegible]

[illegible]

Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my gratitude to Dr. Blayne E. Mayfield for his constant advice and suggestions from the very beginning of this thesis. My thanks also go to Dr. K.M. George and Dr. George Hedrick for serving on my graduate committee. Lastly, I acknowledge the help of Dr. Keith Teague and Dr. David Miller for their assistance. All of their suggestions and help was much appreciated.

To all of my family, but especially my parents, Larry and Marietta Jo, my heartfelt thanks for all the many years of love and support. My thanks go to Larry Kraemer also for his time and experience in the many hours of proofing and typing. And finally, to Laura Wardius, soon to be my wife, my eternal thanks for her patience and companionship these last five years.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER I

INTRODUCTION

Computer Graphics has emerged from the dark and noisy arcade to become one of the most prominent areas of computer application in the world today. The movie industry increasingly relies on computer graphics for special effects and animation, while the medical world eagerly adopts new imaging techniques as soon as they are made available. Even the art community has been invaded by computer generated images, many of which challenge the most sophisticated viewer to determine whether or not the work is that of a human or a machine. Very few areas of computer science have had as large an effect on the everyday life of modern society.

Two Goals of Computer Graphics

In computer graphics, the current research pursues two goals: speed and realism. Unfortunately, these two goals are often at odds with each other. Achieving greater realism requires increasingly complex (and consequently more time consuming) rendering methods. Conversely, greater speed requires simpler, and often less realistic algorithms. As often happens, one goal is sacrificed for the other.

Realism

Advanced techniques like Ray Tracing [Whitted, Foley] and Radiosity [Goral] were developed to create photo-realistic images. These highly realistic rendering models actually simulate the generation of light rays from their source, the propagation of the light in the scene, and its reflection and refraction by the objects in the scene. Additional enhancements to these two methods such as shading, texture mapping, and volume rendering have increased overall realism. Unfortunately, the time it takes to generate one of these highly realistic scenes is often measured in hours, if not days.

Speed

For the computer scientist whose goal is speed, a typical project could be to generate ten seconds of a movie at 24 frames per second. For this assignment, time-intensive rendering methods are not acceptable. If photo-realism is the ultimate achievement in realistic imaging, real-time imaging is the ultimate goal of the speed methods.

There are two major approaches to this goal of increased speed: brute force and parallelism [Machover]. Brute force methods employ larger and faster computers, requiring larger and larger budgets, and newer and newer technology. Parallelism, on the other hand, often makes use of existing hardware technology to achieve the same speed as brute force methods. Can the twin goals of speed and realism be realized?

Z-Buffering Is The Answer

Z-Buffering is a rendering algorithm that balances the two goals of graphics, speed and realism. The algorithm itself is not complicated, is easily implemented in a variety of languages on many different types of hardware. At the same time, Z-Buffering generates highly realistic images, and supports a number of additional techniques that enhance realism. The research done for this thesis focuses on the Z-Buffer rendering method.

Parallel Z-Buffering, A Better Answer

Developing a parallel algorithm from a sequential one is intuitively expected to yield better results. This has proven to be correct on a number of different classical problems in computer science [Fox].

The goal of this thesis is to develop and implement Z-Buffering techniques on a parallel architecture in the hopes of increasing performance and overall speed.

Overview

In Chapter 2, background information is given on the development of rendering and the evolution of the Z-Buffering algorithm. The parallel environment is also discussed, with special emphasis on the iPSC/2 parallel computer on which a majority of this research was performed.

Chapter 3 reviews some of the literature leading to the development of the Z-buffering method, some of the additions that have been made to the technique, and some of the previous research in the area of parallel Z-buffering.

Chapter 4 discusses design choices necessary to implement the Z-Buffering algorithm on a parallel architecture. Many considerations should be carefully weighed before any sequential algorithm can be applied to a parallel architecture.

Chapter 5 details the four parallel implementations of the Z-Buffering algorithm researched for this project. The four methods presented illustrate a gradual evolution of parallel algorithm development and attempt to solve some of the problems unique to the parallel environment.

Results are discussed in Chapter 6, following a general discussion of various performance measures. Detailed accounts of the timing results from the four different methods are discussed and illustrated.

Finally, Chapter 7 contains the summary and conclusion of this research. The chapter also discusses a few of the problems encountered. Lastly, areas of future research are proposed and avenues of approach discussed.

CHAPTER II

BACKGROUND

All computer graphics methods which generate images from scenes, including Z-Buffering, can be termed renderers. Rendering is the process of translating a two or three dimensional scene description into a two dimensional image to be displayed on a graphical output device i.e. a Video Display Terminal (VDT) or Printer/Plotter.

Scene Components

Any scene can be broken up into three elements: the viewer, the view plane, and the primitives. Figure 1 shows the relationship between these three elements. The viewer provides information about the viewing position and direction. The view plane element of a scene contains the view plane, the 'window' through which the viewer sees the scene, the view plane orientation, the dimensions of the view plane in pixels, and the frame buffer. Primitives (discussed in greater detail in chapter 4) form the third element of a scene. The primitives are the objects displayed in the scene such as spheres, cubics, conics, and pyramids, and also the components of more complex objects such as trees and buildings. While a scene has only one viewer and view plane, the number of primitives may be as few as one or as many as tens of thousands.

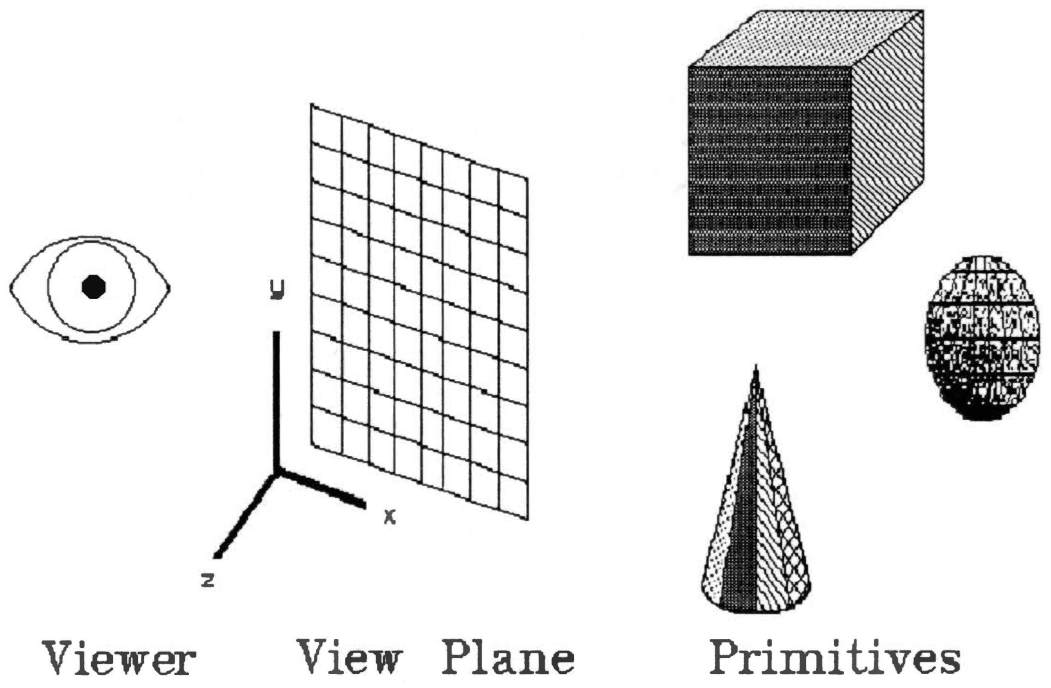

Figure 1: Components of a Scene

The frame buffer mentioned above is the central data structure for all renderers. Along with other information, a frame buffer stores the rendered image, whether color, gray-scale, or black and white. The frame buffer is also the structure that is displayed on a VDT, or written to an image file, for later viewing. For color and gray scale images, a frame buffer will typically have one array for each spectrum. Each element of each array is a number that corresponds to a light intensity value (red, green, blue, or white) for the corresponding pixel. Black and white images will typically have only a single spectrum, often a bit-map.

Tessellation

There are a variety of ways in which a particular method may render the primitives in the scene: directly from a mathematical description, through polygonal approximation, etc. Rendering methods that are aimed at greatest realism, such as raytracing, employ the mathematical description of the primitive directly. Other methods, like the Z-buffer method, approximate the primitive using a large number of polygons to achieve greater speed.

The process of approximating primitives using polygons is known as tessellation. As an example, take a regular octahedron having each of its eight points lying on the unit sphere. Replace each of the eight triangular polygons that makes up the octahedron by four smaller polygons, while maintaining all new vertices on the unit sphere. After a number of iterations of this process, a very realistic sphere can be generated. The process of iteratively generating a shape using the above method is also known as recursive subdivision. Figure 2, below, shows two iterations in the tessellation of a 2D triangle.

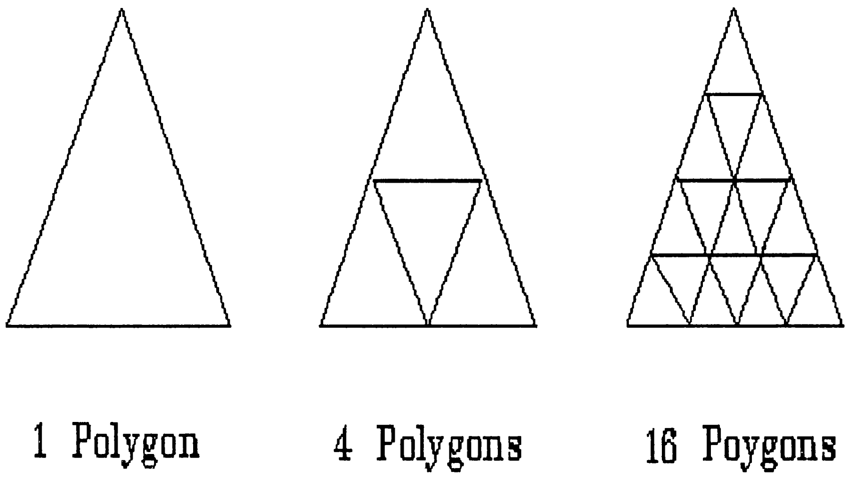

Figure 2: Tessellation

Tessellation is often preferred over direct calculation using a mathematical representation because it saves time. Fewer calculations are required to compute each pixel of every object in the image. A rendering method that is based on polygons rather than direct mathematical descriptions also has the advantage of being smaller in size because the program is only required to render one object, a polygon, rather than having to render each type of primitive.

Scan-Line Rendering

Now that the scene has been described, the methods of rendering primitives into the frame buffer can be examined. Before describing the Z-Buffer rendering algorithm however, it is necessary to look at the scan-line algorithm, upon which it is based.

The scan-line algorithm, developed by [Catmull] in the mid seventies, maps polygons into scan-lines, the horizontal rows of pixels on a VDT. For each scan-line, the algorithm first determines all the edge pixels -- the points where the scan-line intersects the polygon. For convex polygons, there will be 0, 1, or 2 intersections. Concave polygons are slightly more difficult due to the larger number of possible intersections. Next, the intersection list for each scan-line is sorted by x-values; all y-values are identical because they lie on the same scan-line. For convex polygons, this involves a single comparison. Standard sorting methods may be employed for concave polygons. The sorted list of edge pixels is then traversed, and all pixels falling between the points of intersection are added to the frame buffer. Algorithm 1 gives psuedo-code for the simple scan-line algorithm.

```
For each scan-line S in the frame buffer F
     Intersect P and S
     For each pair of intersections points, p1 and p2
          For each pixel X between p1 and p2
               Assign RGB to FB[S][X]
          Endfor
     Endfor
Endfor
```

Algorithm 1: Scan Line Algorithm

Figure 3 shows many scan-lines in the scan-line conversion of simple two dimensional polygons. Scan-line ten intersects the concave polygon in two places, point (13,10) and point (17,10). All pixels between those two points are rendered into the frame buffer. Three dimensional polygons are converted similarly.

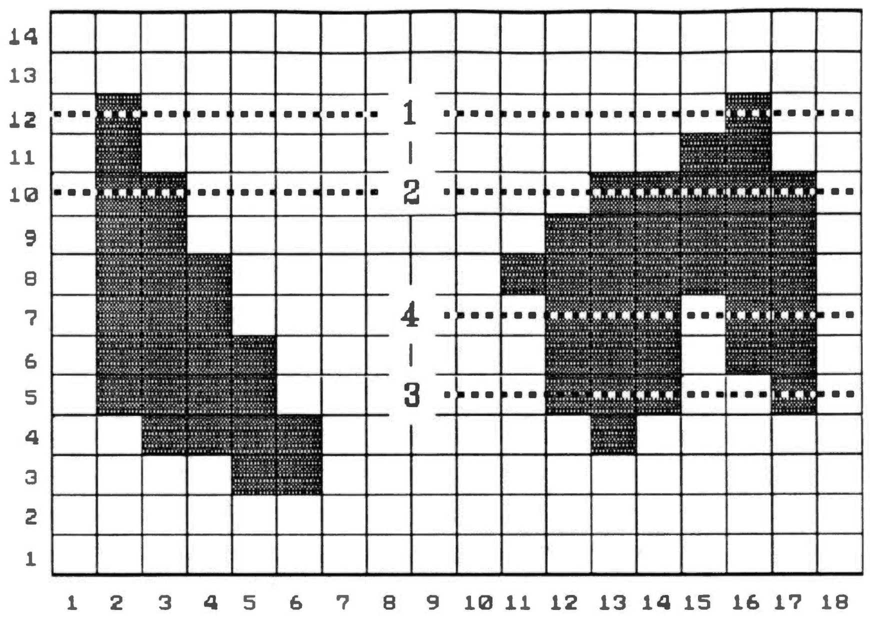

Figure 3: Scan Line of Convex and Concave Polygons

<u>Advantage of Scan-Line Rendering</u>

Speed is the greatest advantage of the scan-line renderer. This algorithm relies on the principle of coherence, the tendency of an object, and therefore its pixels, to maintain a high degree of similarity over a given space. Span coherence then is exploited by the algorithm in relying on the pixels of a given scan-line to be colored similarly within a single object. Edge coherence, the tendency of edges to be similar, is also exploited in computing edge pixels and in various shading algorithms [Foley]. The list of edge pixels can be computed quite rapidly given the vertices of the polygons, and the pixels can be computed equally fast with only a few operations.

In contrast, rendering methods that employ the mathematical description of the primitive directly often require hundreds of calculations per pixel, because each pixel is rendered independently of its neighboring pixels. In ray tracing for example, each ray must be intersected with every object in the scene. This is a highly computationally intensive step. Intersecting 512 rasters against 2000 polygons is much easier than 512^2 rays against 50 complex parametric shapes.

Disadvantages of Scan-Line Rendering

Despite its speed and simplicity, the scan-line algorithm is not without its problems. One weakness of the algorithm is exposed when two or more polygons overlap. Depending on the order in which the two polygons of Figure 4 are rendered, two very different images can be generated.

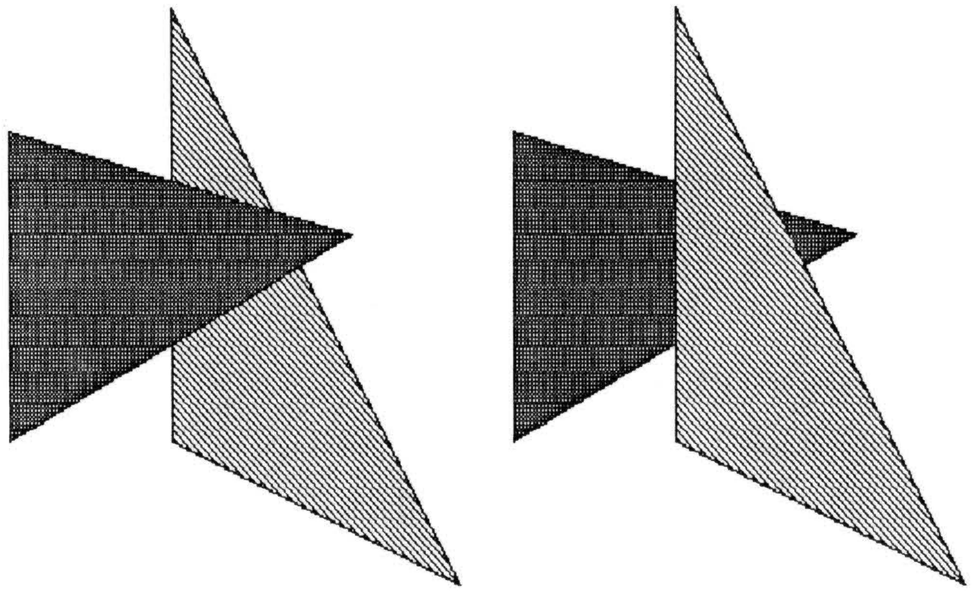

Figure 4: Problem of Overlapping Polygons

To solve the problem of overlapping polygons, a step to sort the polygons is incorporated into the simple scan-line algorithm [Newel]. This modified algorithm, which is known as the painter's algorithm, attempts to sort the polygons in such a way that those polygons farthest away from the viewer are rendered first, while nearer polygons are rendered on top of those farther away. In this way, overlapping polygons are correctly rendered into the image, since those polygons closer to the viewer are 'painted over' those that were rendered earlier.

The polygons can be sorted by any of the standard sorting algorithms. Usually, either the centroid or the minimal z coordinate of the polygon is used as the key for sorting. Computing centroids and finding minimal coordinates takes some computation, and the sorting process itself can be quite costly, especially with large scenes of thousands or hundreds of thousands of polygons.

Adding the sorting step corrects problems due to overlapping polygons, but it also greatly increases the complexity of the algorithm and decreases its overall speed. Because the polygons in the scene must be compared and sorted against one another, they are no longer independent as in the simpler scan-line algorithm. This consideration weighs heavily on the algorithm, especially when considering the parallel implications.

Although sorting solves problems of overlapping polygons in two or three dimensions, an additional problem occurs only in scenes of three dimensions: intersecting polygons. Figure 5 shows the example of a scene and two very different images, the possible results of rendering the intersecting polygons using the painter's algorithm. Depending on the sorted order of the polygons, one or the other image may result; both are wrong.

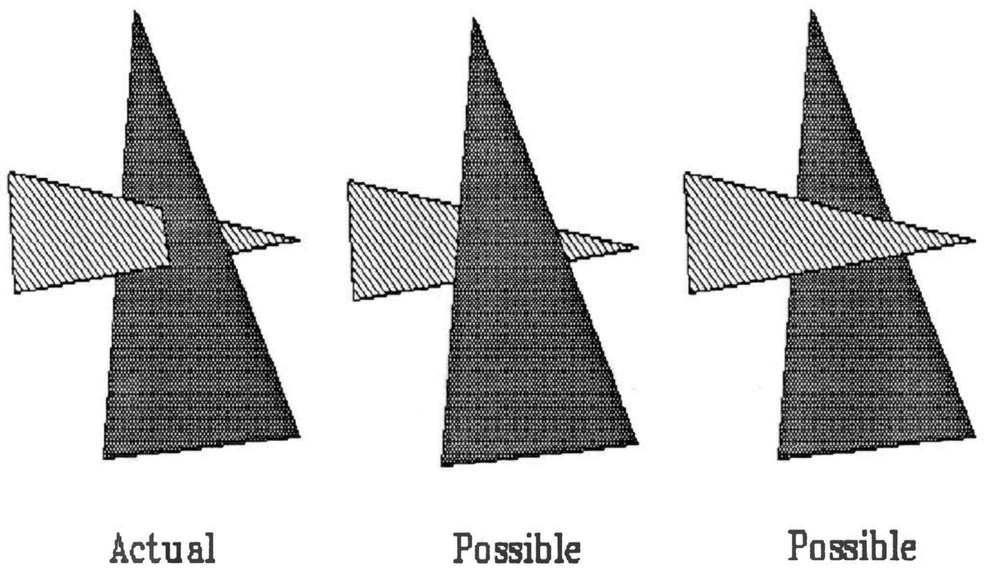

Figure 5: Problem of Intersecting Polygons

No sorted order will ever render this image correctly using the painter's algorithm because one polygon will always be added to the frame buffer before the other. To solve this problem, Newell added to the algorithm a series of tests to determine if one object intersects another. After all the tests and comparisons, if the problem does occur, the original polygon is divided into two or more sub-polygons at the point of intersection, thus allowing the sorting step to work properly.

It is not necessary to detail this method of resolution to see that it slows the algorithm even further. Other methods were then added to help restore the lost speed.

Binary Space-Partitioning Trees [Schumacker] are based on the idea that if a plane can be found that wholly separates one set of polygons from another, then the polygons that lie on the same side of the plane as the view point can obscure, but cannot be obscured by, the other polygons that lie on the other side of the plane. The use of bounding volumes [Rubin] is another method designed to increase the speed of the slowed algorithm. Instead of dividing the polygons into two groups separated by a plane, this method groups together objects within a three-dimensional volume such as a sphere. If two bounding volumes do not obscure each other, then no primitives within the bounding volumes can obscure each other. What started out as a simple algorithm has rapidly become a nightmare of complexity.

Z-buffer Rendering

The Z-Buffer rendering method takes the simplicity and speed of the original scan-line algorithm and combines it with the accuracy of the painter's algorithm, but without incurring additional computational costs. In the Z-Buffer algorithm, another spectrum (i.e., another array) is added to the frame buffer. This spectrum records the depth value associated with each pixel, which is based on the pixel's Z coordinate, hence the algorithm's name. To start out, the spectra is initialized to positive infinity.

Before a pixel is added to the frame buffer, its depth is first compared with the current depth recorded in the Z spectral element of the frame buffer. If the new depth value is less than the recorded value, meaning the new pixel lies closer to the view point than the recorded one, the new pixel replaces the recorded one. Algorithms 2 and 3 give psuedo-code for both the standard polygon renderer using the scan-line algorithm and the Z-buffer renderer which uses the scan-line algorithm in combination with the improved frame buffer.

```
For every object O having color RGB
     Tessellate O into polygons
     Sort polygons, resolving any conflicts
     For every polygon P
          For each raster R in the Frame Buffer FB
               Intersect P and R
               For each pair of intersection points,
                         (x1,R,z1) and (x2,R,z2)
                    For each pixel X from x1 to x2
                         Assign RGB to FB[X][R]
                    Endfor
               Endfor
          Endfor
     Endfor
Endfor
```

Algorithm 2: Standard Polygon Rendering Algorithm

```
For every Object O having color RGB
    Tessellate O into polygons
    For every polygon P
        For each raster R in the Frame Buffer FB
            Intersect P and R
            For each pair of intersection points,
                    (x1,R,z1) and (x2,R,z2)
                dz <- (z2-z1)/(x2-x1)
                z <- z1
                For each pixel X from x1 to x2
                    If (z < FB[X][R])
                        Assign RGB and Z to FB[X][R]
                    z <- z + dz
                Endfor
            Endfor
        Endfor
    Endfor
Endfor
```

Algorithm 3: Z-Buffer Rendering Algorithm

Advantages of Z-Buffer Rendering

The Z-Buffer rendering algorithm provides many advantages over the scan-line polygon rendering algorithm, and minimal disadvantages. (The primary advantage of Z-Buffering is the lack of the sorting step required by the standard polygon renderer.) Even employing the best sorting algorithms, which have O(n log n) time complexity, the sorting process may be more time consuming than the actual rendering portion of the algorithm.

Another advantage of the Z-Buffer method is that the polygons are once again independent of one another. The order in which the polygons are rendered does not affect the final image. Instead of relying on a sorted order to render pixels correctly, the Z portion of the frame buffer insures the closest object is seen in each pixel. The problems of overlapping and intersecting polygons in Figures 4 and 5 are correctly rendered the first time. The advantage of polygon independence will become more apparent when the parallel data decompositions are discussed in chapter 5.

Disadvantages of Z-Buffer Rendering

One noticeable disadvantage of the Z-buffer algorithm is the larger memory requirement due to the added Z spectrum. In the simple scan-line algorithm, only the RGB triple is stored; now RGB and Z must be stored. Unfortunately, the Z elements are often larger than the color spectra elements. Modern display hardware usually can handle only 8 bits per color spectrum element, or 24 total bits per pixel to represent an RGB image. For reasonable precision however, the Z elements must be at least 32 bits per pixel [Piol], corresponding to the single precision floating point variable type of many hardwares. In other words, adding the Z spectrum more than doubles the size of the frame buffer. A monochrome bitmapped 512 square image would require only 32,768 bytes, an 8 bit gray-scale image requires 262,144 bytes, a 24 bit RGB color image 786,432 bytes, while a 24 bit RGB&Z image requires 2,621,440 bytes. Typical image sizes are more often than not even larger then the 512 square example. The memory requirement of the Z-Buffering algorithm is serious a consideration to keep in mind.

CHAPTER III

LITERATURE REVIEW

As mentioned earlier, Z-Buffering was originally developed by Catmull [Catmull] out of the earliest scan-line algorithms [Sutherland]. This first algorithm was built on the work done for the scan-line algorithm [Watkins] and later improved on by additional work of Newell et.al. [Newell]. Since that time, many other additions have been made to the algorithm, and it has been used in many different applications.

When used to produce images of high complexity, Z-Buffering has two very noticeable drawbacks immediately apparent: aliasing and lack of transparency. Aliasing is the jagged edge, or stair-step appearance, of near-vertical and near-horizontal lines when rendered with traditional methods. Anti-Aliasing is the process of removing the aliasing, to produce a much smoother line. Transparency is the feature which allow some light to pass through objects, rather than reflecting all of it, to illuminate objects lying farther from the viewer. Both these features are absent in the standard Z-buffering method, though very apparent in the more realistic renderers.

A-Buffering [Carpenter], a descendant of Z-Buffering, solves the two problems of aliasing and transparency. It gets its name from an anti-aliased, area-averaged, accumulation buffer added to the frame buffer. To solve the aliasing problem, this method incorporates a 4x8 bit mask per pixel to represent subpixel coverage. This bit-mask is similar to work done by Fiume et.al.[Fiume]. Transparency is handled by dynamically maintaining a sorted list of pixel fragments associated with every pixel. A pixel fragment describes the transparency, color, area, and fill mask associated with each object visible in the pixel. From this information, an anti-aliased image with transparency can be generated. The additional computation slows the algorithm, but does a great deal to enhance the realism of the images.

Alternately, a two-pass Z-Buffer method [Williams] is highly effective in implementing shadows without suffering the penalties of A-Buffering. In this method, the first pass of a Z-Buffer algorithm creates a frame buffer from the viewpoint of the light source. This frame buffer will give distances from the light to the nearest object in the scene. The second Z-Buffer pass, from the viewpoint of the viewer, contains a slight modification of the standard algorithm. For each pixel to be added to the frame buffer, if the distance from viewer to the object is greater than the distance from object to the light source as recorded in the first pass, then the pixel lies in shadow. Additional light sources are handled by additional passes of the first step. Figure 6 shows two cubes viewed overhead and in perspective. The arrow in the overhead view is the viewpoint. The dotted line marks the shadow cast by the right cube onto the left in the scene.

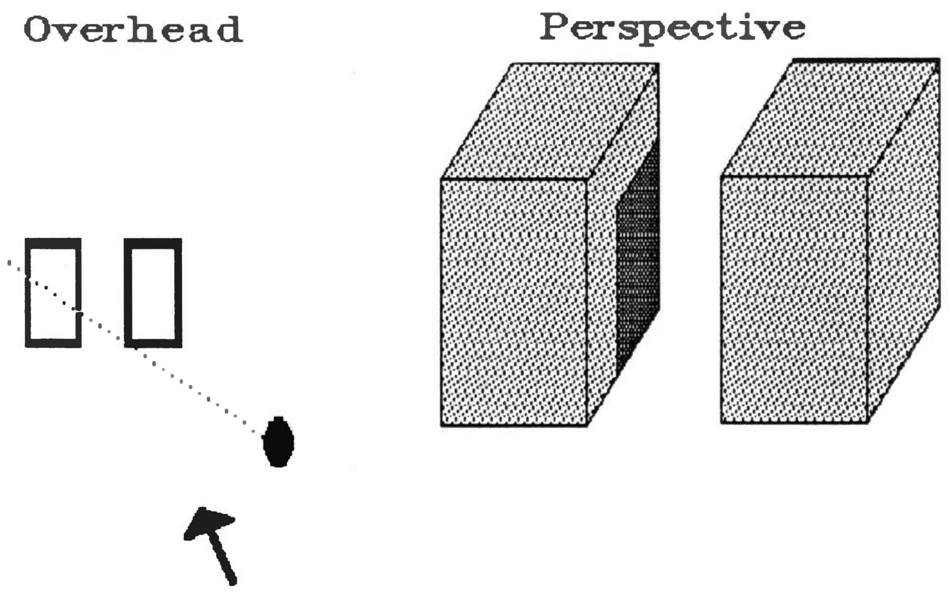

Figure 6: William's Shadowing Method

Constructive Solid Geometry (CSG) [Rossignac] is another addition to the Z-Buffer rendering method. CSG is a technique whereby complex shapes can be defined as a boolean expression of two or more simpler shapes. For example, the lense of a magnifying glass might be defined as the intersection of two spheres which barely overlap. The technique is useful in defining realistic shapes that are not easily built from simple primitives, greatly enlarging the set of objects capable of being rendered.

Shading is perhaps the easiest technique to add realism to a renderer. Shading Models [Bishop, Blinn] give even a simple object a very realistic appearance. The simplest shading method, Gouraud shading [Gouraud], is incremental in nature. The shade value of edge pixels at each scan-line are computed, and pixels lying between the edges are computed with a single addition. Phong shading [Phong] improves on Gouraud shading by linearly interpolating the surface normal at each pixel rather than relying on the edge pixels. The method is much more computationally intensive, but yields far better results.

The application of Z-Buffer rendering in parallel has largely been a hardware approach. One approach has been to vectorize portions of the algorithm on SIMD (Single Instruction Multiple Data) machines such as the Cray XMP-2/4 and Convex C-1 computers [Dyer]. The portions of the algorithm that can be vectorized include calculation of surface normals, edge determination, clipping, and shading. Overall performance speed-up is directly proportional to the amount of the algorithm that can be successfully vectorized.

Another hardware implementation of Z buffering uses VLSI approaches [Hu]. In this method, a number of processors, cooperate to render a single image into a single frame buffer. To minimize the memory bottleneck of multiple processors accessing a single frame buffer, a complex Omega Network [Hayes] is added to the design. This network can be thought of a large cross-bar switch, in which multiple paths exist from the processors to the memory, allowing simultaneous accesses. Input, in the form of polygons, is broadcast to each of the N processors, which each in turn render every Nth scan-line into the frame buffer.

CHAPTER IV

DESIGN CHOICES AND IMPLEMENTATION

Architecture of the iPSC/2

In parallel applications, the hardware available greatly influences the design of the software, especially on MIMD (Multiple Input Multiple Data) machines. The hardware and software work best together when matched in granularity. Granularity of a parallel system is defined as the ratio of computational power to communication power. Applying this definition to parallel hardware, a coarse-grain machine has very powerful processors but very costly communications. A fine-grained computer has communications and computational power that are equally matched. When applying granularity to parallel software, a process or task that communicates infrequently with other processes or tasks can be termed coarse-grain, while a process that must communicate often is termed fine-grain.

The iPSC/2 Hypercube concurrent computer system is a typical model of a medium-grained distributed memory parallel architecture. Distributed memory means that no processor can access the memory of another processor, all are separate. It is on this hardware that much of the research of this thesis was performed.

Hardware

The iPSC/2 computer is divided into two main subsystems, the System Resource Manager (SRM), often called the host, and the Tower, which contains all of the nodes. The SRM is a 386 microprocessor front-end connected by a high speed data line to a single node of the tower.

Each node board in the tower contains a 386 microprocessor, a Weitech floating point co-processor, and up to sixteen megabytes of random access memory. Nodes communicate with each other and the host through a proprietary interconnection scheme. In this interconnection scheme, the nodes are connected in a hypercube topology, having direct connection to $\log_2 N$ other nodes, where N is the total number of nodes. By definition, in a hypercube architecture, each node is connected to $\log_2 N$ other nodes, meaning the maximum distance a message must travel between any two nodes is N-1 hops.

Software

The software for most iPSC/2 applications is designed in two parts: the host program and the node program. The host program runs on the SRM and is designed to decompose the problem for distribution among the nodes, handle I/O functions, and collect any output from the nodes. The node program runs on each node and performs the actual work of the problem, all in parallel. All communication among the nodes and between the nodes and host must be expressly stated by the programmer using message passing. This architecture is ideal for medium-grained problems. Figure 7 shows the data flow of a typical program on the Hypercube.

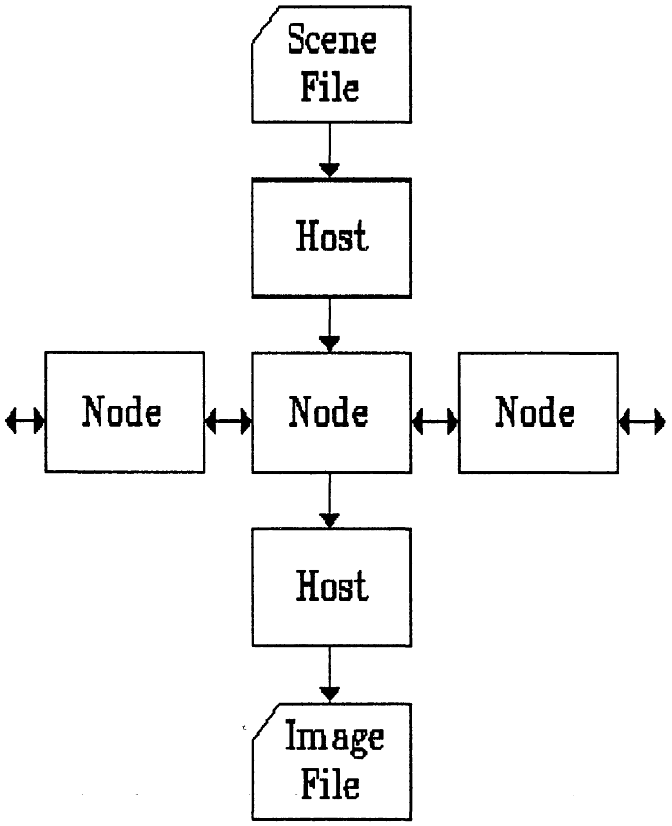

Figure 7: Data Flow

Communication

All communication on the iPSC/2 is handled by message passing. A message may be any number of bytes in length, up to the available memory on the receiving node. Both synchronous and asynchronous communication are available. In synchronous communication, the sending node waits for the message arrival to be confirmed, while the receiving node blocks processing till the message arrives. Asynchronous transmission does not wait for confirmation of arrival, and reception does not block processing. The particular path a message will take from one node to the next is determined by proprietary software running on each node. This path determining software is similar to other "worm-hole" methods in that a guaranteed unique minimal distance is traveled. As expected on a medium-grained machine, communication on the iPSC/2 is quite costly. Messages under one hundred bytes in length cost on the average 4 milliseconds per byte. Longer messages cost 275 ms plus 4 milliseconds per byte.

Hypercube Image Processor

Unfortunately, the Hypercube does not have any type of graphical display device directly connected to it. For that reason, the Hypercube Image Processor (HIP) [Daniel] can be used to display the generated images on a SUN 3/60 workstation. This software package was designed to perform standard image processing operations on image buffers.

These operations include convolution, threshold, read, write, etc. HIP also implements a display ability using UNIX sockets(2) [Sun], which solves the lack of display problem for the Hypercube.

The HIP display allows images of arbitrary size to be displayed very rapidly, within the physical pixel limit of the SUN workstation (1100x900 pixels). A 512x512 RGB image can be displayed in only a few seconds. Unfortunately, the SUN palette is limited to 256 colors displayable in any one image.

The images generated for this thesis did not need to be formatted for HIP. A number of different graphical formats could have been used, including GIF, HQX, or PIC. The output format was chosen to fit the available hardware. In fact, some other formats will generate smaller sized image files due to internal image compression using techniques such as run length encoding.

Primitives

Determining the set of primitives is a very important task because all output will be built on this set of objects. If the set of primitives is too small, few complex objects can be generated, and if the set is too large, the program's code segment may become too large for each node's finite memory. The number of primitives is only limited by the memory available. For this project, all primitives that can be tessellated into a finite number of polygons are applicable. This research uses conics, cubics, cylinders, and spheroids. Each primitive in this set is easily tessellated into polygons. This set of five primitives can then be combined to generate more complex objects. Figure 8 shows the collection of primitives used in this research.

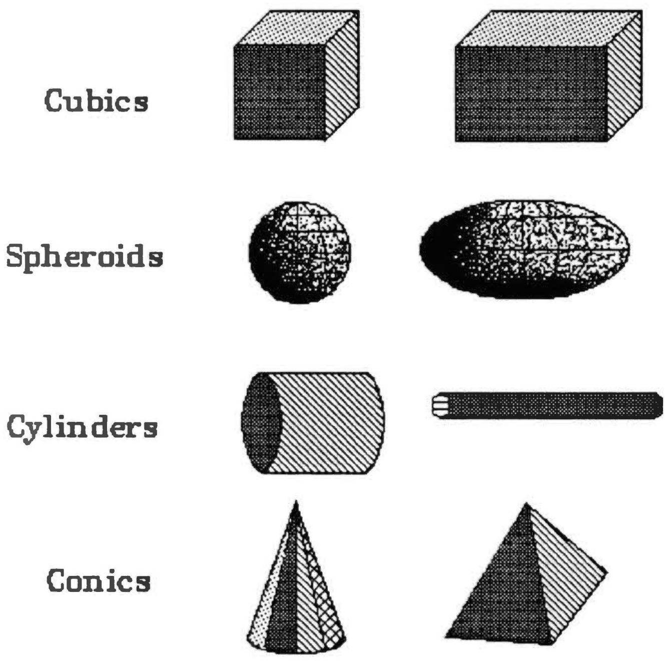

Figure 8: Primitives

Scene Description Language

To render any complex image, a scene description language must be implemented. The scene description language is used to define to the renderer the image to be generated. The scene description file may contain parameters for all the scene options and variables that correspond to the view plane and the viewer: viewing position, viewing direction, ambient light level, and position of light source(s). The file also defines the objects and their parameters, such as color, position, size, and object-specific variables, such as the maximum recursion level in recursively generated objects.

Scene description languages vary widely in flexibility and functionality. Some are fixed format, while others are absolutely free format. Some are truly languages in the strict sense, while others are merely a list of integer parameters. Renderman [Upstill] and PHIGS+ [Mallett] are two of the newest and most flexible scene description languages available today, implementing nearly every aspect of complex scenes.

The more flexibility that is built into a language, the larger the parser required. Smaller languages may be faster, but they lose functionality and extensibility. Ease of use is also highly desirable in these languages.

For this research, the scene description language consists of a flat ASCII file containing one definition per line. A definition is either a viewplane/viewer parameter or an object. An object is a list of tuples containing the parameter name and one or more values. There is no order imposed on the definitions by the parser. Figure 9, below, gives an example of a scene description file.

```
#
#    Z-Buffer datafile to generate
#    three medium sized red cubes
#    on a field of multicolored
#    stars with a randomly colored,
#    randomly positioned small sphere.
#
#    Viewer/Viewplane definitions
#
ambient  40
background black
light    -1,1,1
nodes    16
overlay  gray
outfile  testout
percent  1
#
#    Object definitions
#
obj cube rot 0,1.9,0 scale 0.4 num 3 color red
obj sphere scale 0.15 recursion 5 color random
```

Figure 9: Sample Scene Description File

Certain objects, such as the sphere in the figure, contain specific variables that can be defined by the user. In the sphere object, the parameter recursion corresponds to the number of iterations the tessellation step is to take.

CHAPTER V

PARALLEL METHODS

For computer graphics applications such as Z-Buffer rendering, there are a number of possible approaches to solving the problem in a parallel environment. Ultimately, these different approaches, or data decompositions, can be grouped into three categories, based on whether the algorithm decomposes the frame buffer, the object set, or both. For this research, four different decompositions have been implemented to perform Z-Buffer rendering on the iPSC/2 concurrent computer. The first two methods, host initiated and node initiated, decompose only the object set. The third parallel method, node initiated with object forwarding, decomposes the frame buffer as well as the object set. The fourth method decomposes only the frame buffer and not the object set. These four methods were chosen because they exemplify the three categories of data decompositions and because they show a natural evolution of the methods employed.

Host Initiated Method

The Host Initiated method is the first of four different decompositions which have been implemented to perform Z-Buffer rendering on the iPSC/2 concurrent computer. In this method, the host reads a datafile containing primitives and tessellates the primitives into polygons. The polygons are compiled into a message structure and distributed to the nodes in a round-robin fashion. Each node receives its polygon packets from the host, processes approximately the same number of total polygons, then renders the polygons into its own complete frame buffer.

After all packets have been received and processed, the frame buffers are then merged hierarchically. The hierarchical merge is a means whereby P processors can combine P frame buffers in N communication steps, where $N=\log_2 P$. A thirty-two node hypercube can therefore merge 32 separate Z-Buffers in five communication steps. At each step, half the active processors send their frame buffer to their next closest neighbor. The other half of the processors (the neighbors) receive the buffers and compare each received pixel's Z value to their own frame buffer's Z value for that pixel. If the received Z value is less than their own, the received pixel's RGB value is copied into the node's frame buffer. Psuedo-code for the merge step is given in Algorithm 4.

```
L_FB = local frame buffer
N = Log₂P
for S = 0 to N
     if ((1<<my_node_number) && S)
     then
          receive frame buffer R_FB
          for I = 0 to MAX_X
               for J = 0 to MAX_Y
                    if (R_FB's Z[I][j] < L_FB's Z[I][J])
                    then
                         L_FB's Z[I][J] = R_FB's Z[I][J]
                         L_FB's R[I][J] = R_FB's R[I][J]
                         L_FB's G[I][J] = R_FB's G[I][J]
                         L_FB's B[I][J] = R_FB's B[I][J]
                    endif
               endfor
          endfor
     endif
endfor
```

Algorithm 4: Frame Buffer Merge

Although this merge step seems time consuming, it is a constant time algorithm based on the size of the frame buffer and not the number of objects in the scene. The time to merge is relatively small compared to the total processing time, especially for scenes containing a large number of polygons. Figure 10 shows the hierarchical merge process on 8 nodes. Each line represents a communication, and numerical values represent the portion of the image resident on the node following the merge. This method maintains the greatest utilization of processors, minimizing total node idle time.

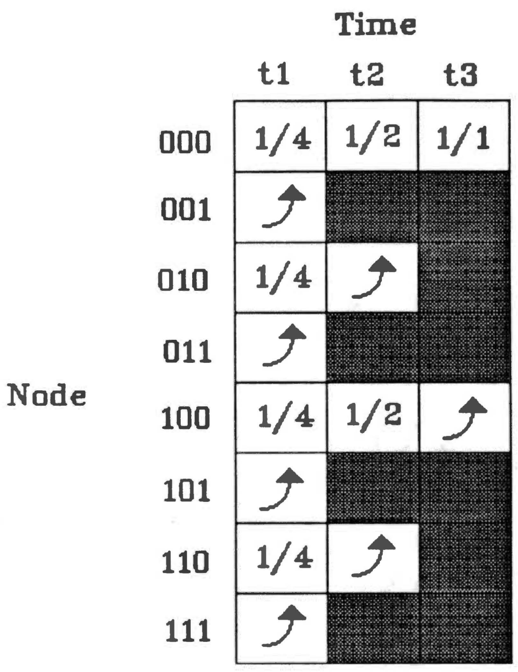

Figure 10: Hierarchical Merge

In the host initiated method, the size of the polygon packet is an important consideration. Smaller packet sizes require more total packets to transmit the same information. Larger packet sizes mean that some nodes will have to spend too much time waiting on a polygon packet from the host. Polygon packet sizes ranging from 10 through 10^3 polygons per packet are compared in this research.

Advantages

The advantage of this method is that each node carries only the code necessary for polygon rendering and not for tessellation, leaving more available memory for larger frame-buffers. Another advantage is that no modification to the Z-Buffer algorithm itself is needed.

Disadvantages

The disadvantages of host initiated processing is that the host-to-node communication link becomes a bottleneck. Because the host handles all the tessellation, the bottleneck worsens the size of the scene increases. Using packets containing 100 polygons each, the size of each message sent to the nodes is three kilobytes.

Variations

A variation of the host initiated method would be to decompose the frame buffer as well as the object set, distributing both over the available nodes. If N nodes are each responsible for 1/N of the frame buffer, the host could send a polygon packet directly to the specific node rendering that portion of the image. Some rendering of the polygons would be necessary to determine which node to send the packet to. This would eliminate the merge step, saving some time, though probably using more time in the partial rendering step to determine the destination node. Overall, this variation of the host initiated method would probably yield a negative overall effect, only adding to the already busy host process without taking advantage of any of the parallelisms of the architecture.

Node Initiated Method

The second parallel data decomposition tested moves the tessellation process from the host to the nodes. As before, the host reads a scene description file, but rather than sending a lengthy polygon packet, the host distributes only the primitive's description to the nodes in a round-robin fashion. The nodes then tessellate the primitives and render them into their own frame buffers, then merge them hierarchically as before.

Advantages

An immediate advantage of this method is that the host-to-node communication bottleneck is removed. The host is able to send out object description packets (only 100 bytes each) much faster than the nodes can tessellate and render them. Another advantage is that tessellation is now accomplished in parallel, making effective use of the parallel hardware.

Disadvantages

Unfortunately, the nodes must now contain code to tessellate all primitives. Though this is not critical, it could become a problem in fully implemented production renderers having much larger sets of more complex primitives. The limited memory of the nodes must be used efficiently.

Variations

One possible variation of this method is to make specific nodes responsible for the tessellation and rendering of specific object classes. For example, on a four node hypercube, node numbers zero and one might be responsible for the simplest class of primitives such as cubes and prisms, while node number two would handle the object class containing spheres, conics, and tori. Lastly, an object class of complex primitive such as fractals and other iteratively defined functions might be the responsibility of node number three.

Splitting the sets of primitives among the nodes would also provide natural load balancing. Hopefully, the distribution of code among the nodes would reflect the proportion of primitives in the scene. Normally, complex shapes are much more computationally intensive, but are greatly outnumbered by simpler shapes. Consequently more nodes should be allocated to simple shape generation.

Because scenes are individually unique, ideal node allocation would vary from one scene to the next. In any static object class distribution, this method introduces the possibility of load imbalancing, if the class distribution does not accurately reflect the scene's object distribution.

A dynamic distribution would be an even more ambitious undertaking. In this variation, the host program could dynamically allocate nodes at run time to reflect the proportions of different object classes present in the scene. Load balance could be monitored, and the node distribution changed mid-run.

Node Initiated with Object Forwarding Method

The third parallel method implemented in this project is designed to decompose the frame buffer as well as the object set. In the previous two methods, an entire frame buffer was maintained on each node. For even moderately sized images, this data structure could easily exceed the memory available to the node. A 1024x1024 pixel RGB image requires eleven megabytes, well in excess of a node's memory capacity. A way to decompose the frame buffer is needed so that each node maintains only 1/N of the total image.

The node initiated with object forwarding decomposition builds upon the simpler node initiated method but maintains only 1/P of a frame buffer per node, where P is the total number of nodes. For this implementation, the frame buffer is decomposed into strips, and each strip is X/P pixels in height by Y pixels in length, where X and Y are the dimensions of the final image.

Primitives are received from the host in a round-robin fashion similar to the previous two methods and tessellated as before. However, if a polygon (or a portion of a polygon) is found to lie in an different strip of the image than the one assigned to the node, then the object description is forwarded to the appropriate node. With a strip decomposition, as the strip grows narrower, a given object is more likely to extend beyond the boundaries of the node's strip. This object consequently requires forwarding to one or more other nodes, which can seriously effect performance, particularly with very large numbers of nodes, thus very small strip sizes.

Instead of forwarding the object description (as sent from the host to the node), the tessellated polygons that compose the object could have been forwarded. The communications cost of forwarding the polygons is measurably less than the time needed to communicate the primitive's description to the appropriate node, combined with the time needed to tessellate the forwarded description. In essence, implementing this change moves the bottleneck present in the host initiated method to the nodes.

After all polygons have been tessellated, forwarded, and rendered, the partial node images are pasted together rather than merged. This can be accomplished hierarchically if memory allows, or linearly by the host. The choice of horizontal strips rather than vertical rows does affect this operation. Because the programs for this project were written in 'C', arrays are implemented in row major order. Pasting a row-major frame buffer strip into the row-major array of the local frame buffer takes only one operation, rather than N operations, where N is the number of rows. Similarly, if the project were designed in FORTRAN, in which arrays are stored in column major order, a vertical decomposition would be best.

Advantages

The greatest advantage of this method is that it increases the size of the overall frame buffer the scene can be rendered into. Where before, NxN pixels were maintained, only Nx1/N pixels are now kept. The savings are quite substantial. A 1024x1024 frame buffer that is too large to fit on a single node using the node initiated method now fits comfortably distributed on four or more nodes using this node initiated with object forwarding method.

Host Broadcast Method

The final method researched explores the merits of decomposing the frame buffer only, rather than the objects set. In this method, the host reads the datafile and broadcasts all the objects to all the nodes, rather than distributing the objects in a round-robin fashion.

Similar to the object forwarding method, each node is responsible for an equal portion of the frame buffer. When a node receives an object from the host, it tessellates the object into polygons as with the other three methods. Polygons lying wholly outside of the node's portion of the frame buffer are discarded at this time. The polygons within the node's portion of the frame buffer are then rendered. After all primitives have been rendered, the frame buffer is pasted together hierarchically as before.

Advantages

Unlike the object forwarding method, this method does not suffer performance penalties as the number of nodes grows larger. This penalty was incurred in the previous method due to the communications costs of forwarding those objects lying on or outside the boundaries of the nodes partial frame buffer. As the number of nodes grew larger, more and more objects required forwarding, resulting in more communications. With this method, the objects not residing on the current node are simply discarding, incurring no penalty whatsoever.

Another advantage lies in the fact that the frame buffer is not required to be decomposed in a contiguous manner. The load balance becomes much more even when the frame buffer is decomposed into scan-lines rather than strips. The first node would receive scan-lines 0, N, 2N, 3N, etc. The second node would receive scan-lines 1, N+1, 2N+1, 3N+1, etc. For arbitrary images, the balance is greater with this method, yielding an even better overall performance.

Disadvantages

Compared to the object forwarding method, this method has a great deal more communications cost due to the initial broadcast step. This can be especially large for scenes with a great number of objects rendered on a large number of nodes. Some of this additional communications cost can be offset by taking into account the observation that broadcast messages can be sent hierarchically just as the frame buffer was collected in $\mathrm{Log}_2 P$ steps rather than N steps.

Other Considerations

A major consideration of this third method is load balancing. Load balancing is the relation of the work done on each node. For the best case load balancing, O/P primitives would be rendered per node, where O is the total number of Objects in the scene and P is the total number of nodes. In the worst case scenario, all objects lie within the single strip of the image and are assigned to only one node. If there are O total objects and P nodes, each node receives O/P primitives, and forwards every one of them to the single node N, the node assigned to the image strip where all objects lie. Node N must then tessellate and render the entire scene of primitives. The worst case would result in even poorer performance than rendering the scene on a single processor due to the communication cost of forwarding the objects. In actual practice, both the best-case and the worst-case distributions are rarely achieved. In graphics, scenes tend to be centered in the image, resulting in a load balance that resembles a bell curve, where nodes having central portions of the image do most of the work.

To help solve load imbalance, the decomposition by strips can be changed to a decomposition by grid. Each processor has X/N by Y/M pixels, where N*M=P is the total number of processors available. Any of a number of possible decompositions are available. For this application, the decomposition which minimizes the total perimeter of the nodes partial image is ideal, thus minimizing the number of overlapping objects requiring forwarding.

Another consideration of any parallel decomposition is in the image to be generated. A scene containing a great number of largely vertical elements, such as a sky-line of tall buildings, would render quite poorly on a horizontal strip decomposition. In practice images are made up of a number of different elements, and there is no way to customize the decomposition for the image.

In this research, the four decompositions are tested using scenes made up of uniformly random distributed spheres of identical size. This will provide a good sample data set, if perhaps a bit too-ideal of a case. The three methods have been tested on 1, 2, 4, 8, 16, and 32 nodes. The number of spheres tested are between 1 and 2^{11} (2048). Each sphere is made up of 5102 polygons. In the largest case, over two million polygons are tested on 32 nodes.

CHAPTER VI

RESULTS

Before detailing the results of the four data decompositions, we need to examine performance measures used to test the effectiveness of the approaches.

Performance Measures

Speed-up, one of the main measures of performance, is defined as the time a task takes on one processor divided by the time taken on n processors, i.e. Tp=T1/Tn. Perfect speed-up is achieved when N processors complete a task in 1/N the time it takes to complete the same task on only one processor. Most tasks do not reach this goal due to a number of factors including inter-node communication cost, uneven data decomposition, and poor load balancing.

Another measure of parallel performance is efficiency, the effective use of parallel resources. Efficiency is calculated as the product of speed-up and the number of processors, i.e. E = Sp*P. Perfect speed-up is also 100% efficient.

When a problem can be decomposed in such a way that no inter-node communication is necessary, it is said to be perfectly parallelizable. An example of a perfectly parallelizable problem would be integer summation: summing all integers between zero and some large integer I on P processors where P < I. The host passes each node a single message containing I. Each node sums its portion of the problem, from I/P*N to I/P*(N+1), where n is the node number from 0 to P-1. The resulting partial sums are returned to the host and totaled before being displayed. No inter-node communication is required, resulting in a perfectly parallelized problem.

Most of the interesting problems in parallel research today require a certain amount of inter-node communication. As expected, these processes yield less than perfect speed-up. It is even possible that some parallel implementations on certain architectures do take longer to execute on multiple processors than on single processor architectures. In such cases, communication costs outweigh computational savings to such a degree that it is not advantageous to decompose the problem into the parallel environment. Most parallel applications fall somewhere between these two extremes of perfect speed-up and actual speed degradation. On a speed-up curve, linear positive slope is good, suggesting that speed-up is directly proportional to the number of processors. Peaked performance shows on a speed-up curve as a local maximum, after which, adding processors to the task may actually decrease performance. Figure 11 below is a speed-up graph showing a number of differing levels of performance.

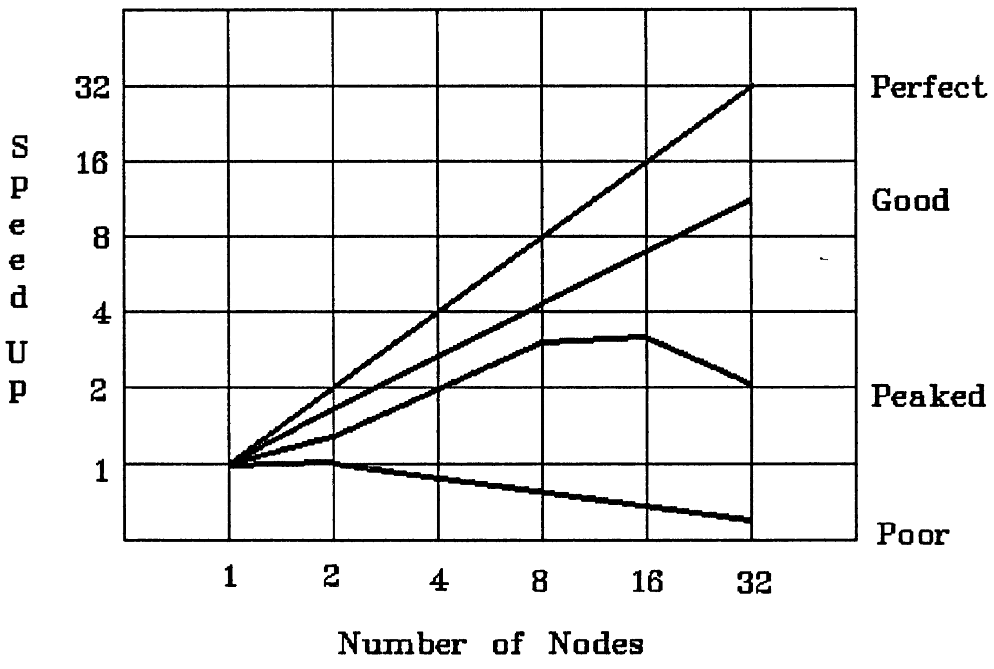

Figure 11: Speed-up Curves

Having presented the two basic performance measures, speed-up and efficiency, we can now examine the results from the four methods implemented for this research.

Host Initiated

As expected, the host initiated decomposition yields very poor results due to the host-to-node communications bottleneck. Figures 12 gives the speed-up curve of the run with the best overall time, while Figure 13 shows the efficiency curves of all runs tested. The host cannot send polygons packets to the nodes as fast as they are able to render them into the frame buffer. As the number of nodes increases for any given number of polygons, both the efficiency and speed-up decrease due to the increased time spent waiting on the next polygon packet to arrive. As the number of polygons increase, this wait-time comes to dominate the time spent rendering. The variation in polygon packet size also yielded minimal change in the overall results. Testing this method on machines with even more nodes, or scenes containing even more polygons, would be expected to show even poorer performance measures.

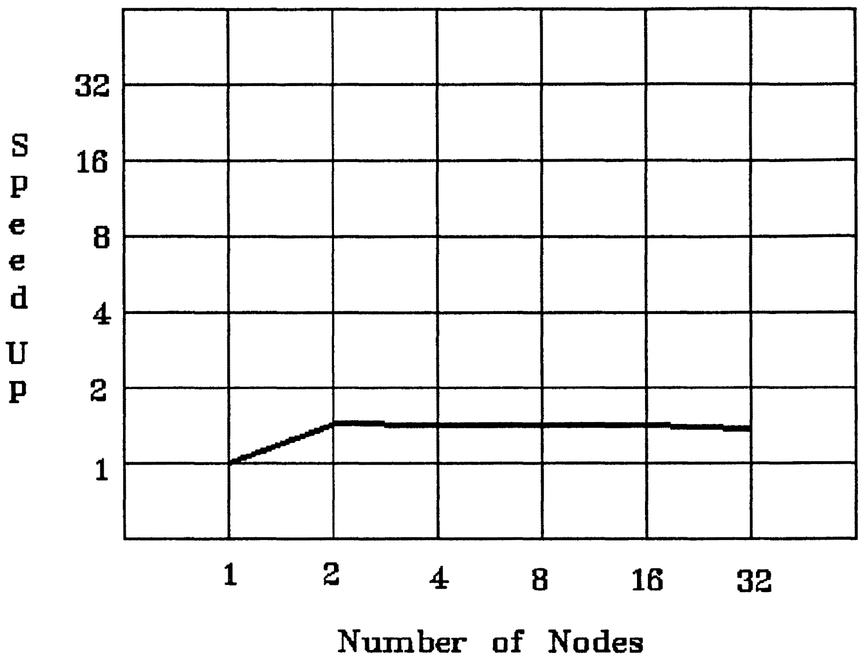

Figure 12: Speed-up of Host Initiated

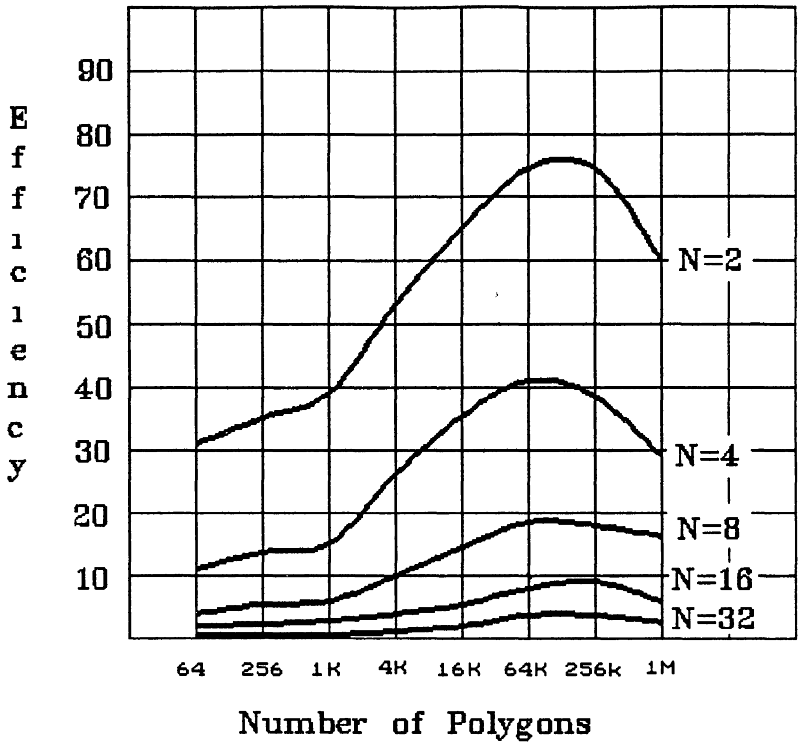

Figure 13: Efficiency of Host Initiated

Node Initiated Method

The node initiated method yielded the best results of all four methods. The speed-up and efficiency curves, given in figures 14 and 15, respectively, are more in line with the optimal speed-up and efficiency curves. The results come from a minimization of overall communications and an absolutely equal load balance. Higher numbers of nodes and larger numbers of polygons would maintain the high results.

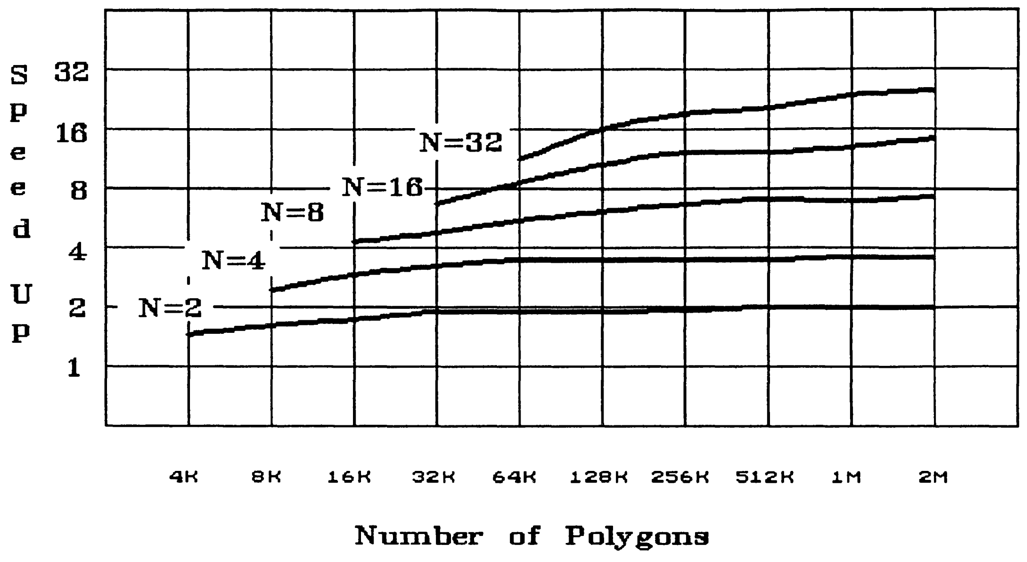

Figure 14: Speed-up of Node Initiated

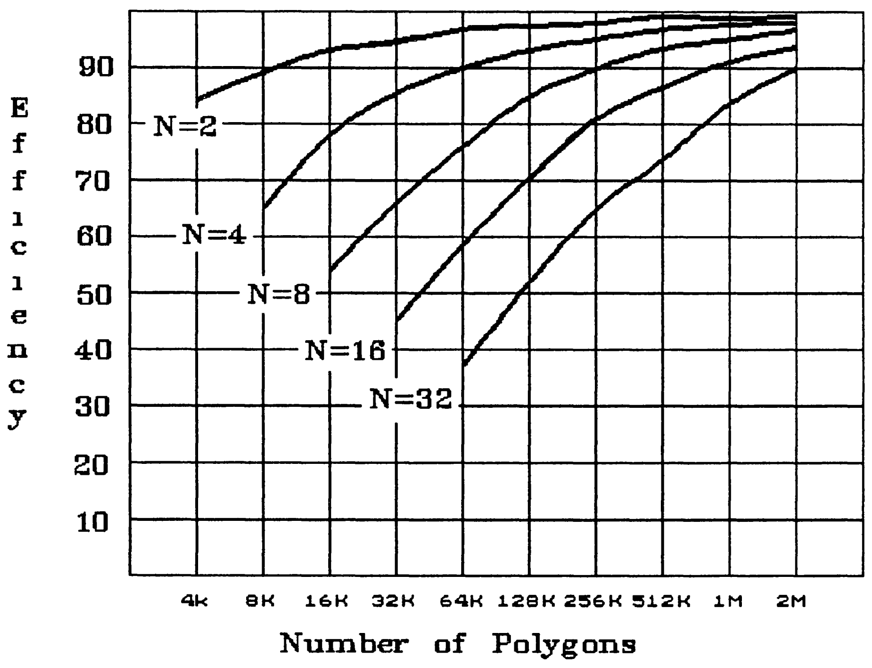

Figure 15: Efficiency of Node Initiated

Node Initiated with Object Forwarding Method

Node initiated with object forwarding failed to yield the expected results. The communications cost of object forwarding weighs very heavily on test runs with large number of nodes, thus having narrow strips of the frame buffer. Figures 16 and 17 show the speed-up and efficiency curves, respectively, of all runs. The performance curves wander quite a bit, most likely due to the random nature of data distribution over the nodes. From one test run to the next, a relatively small change in the percentage of the objects require forwarding can have a significantly high change in timing results, due to the high cost of communications. Averaging the run times of many tests would smooth out the curves, but it is certain that the results would be well below optimal.

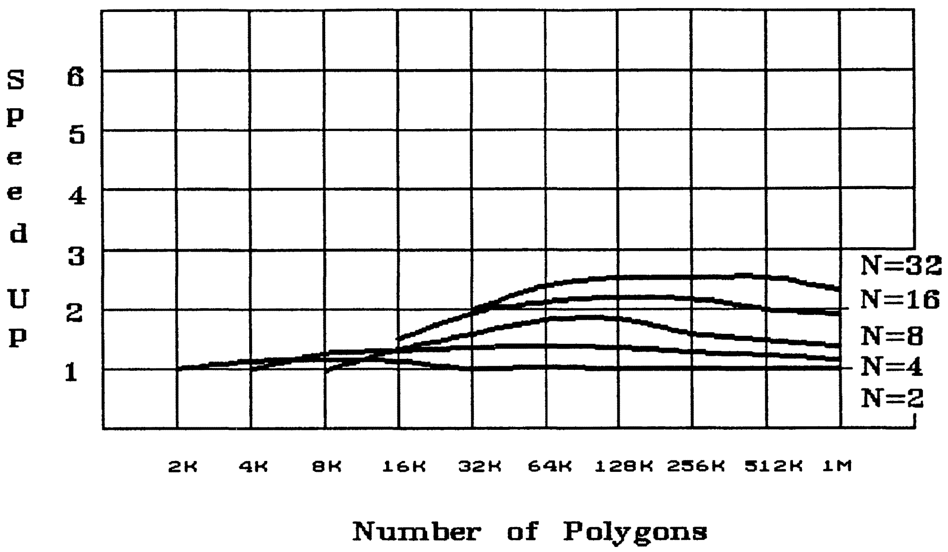

Figure 16: Speed-up of Node Initiated
with Object Forwarding

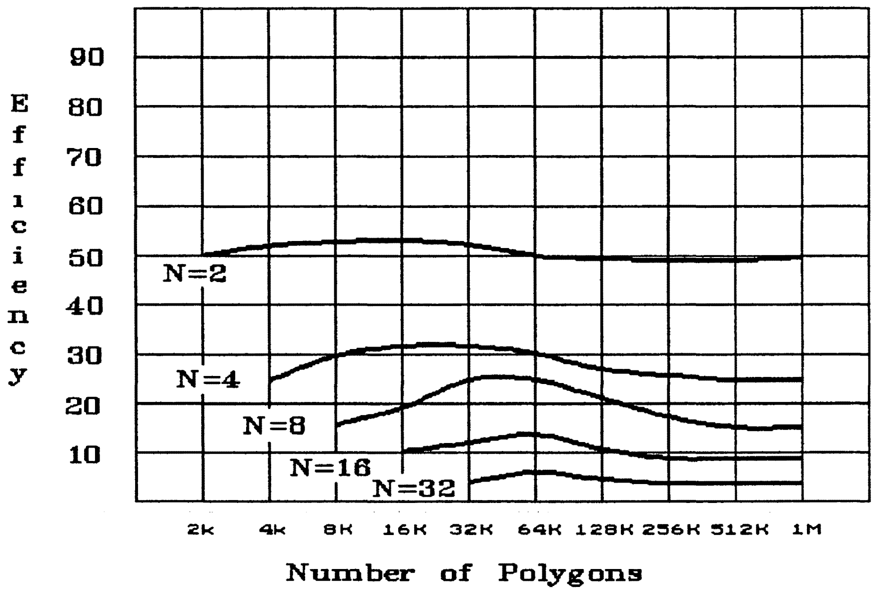

Figure 17: Efficiency of Node Initiated
with Object Forwarding

Broadcast Method

The broadcast method yielded very good results, though not quite as good as the node initiated method. Figures 18 and 19, respectively, show the speed-up and efficiency curves respectively for this method. As the number of nodes generating a particular scene increases, the time to tessellate the broadcast objects remains constant, while the time spent rendering decreases.

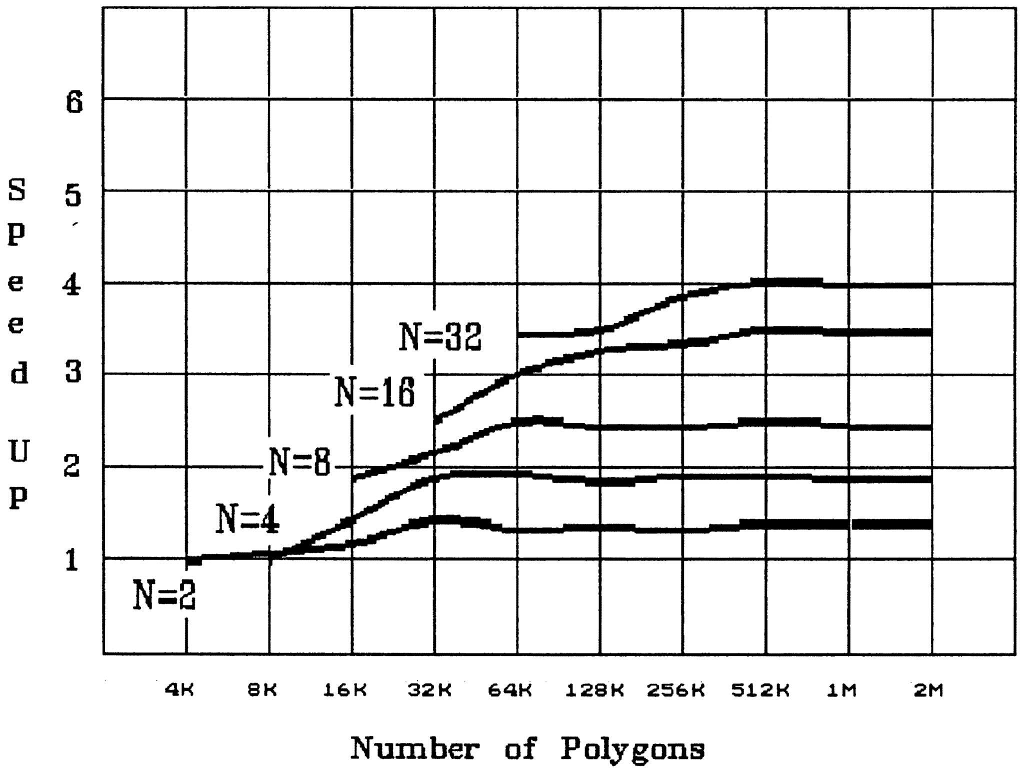

Figure 18: Speed-up of Broadcast Method

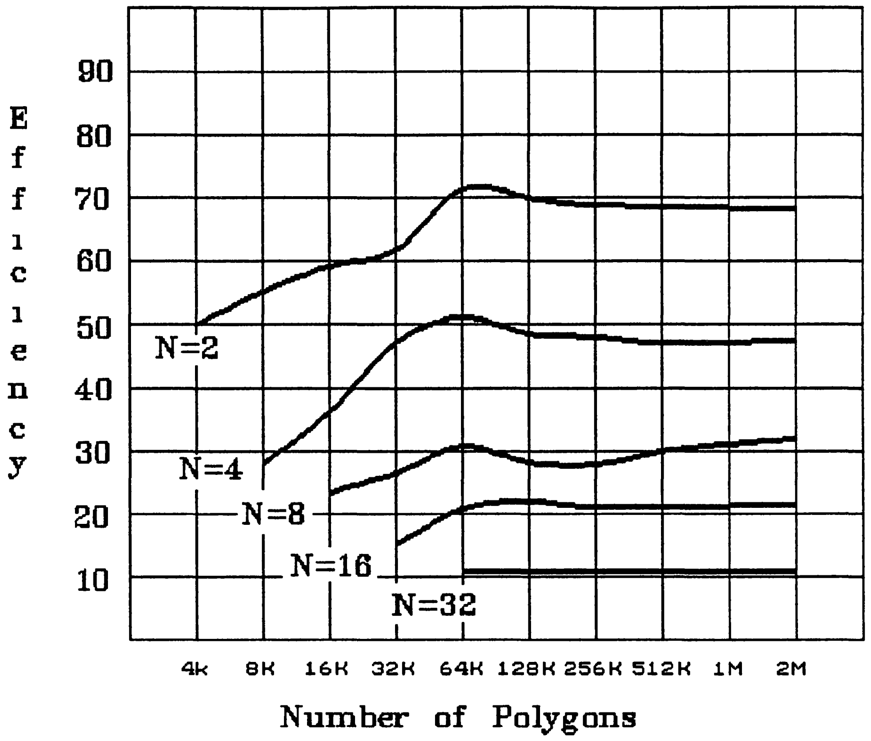

Figure 19: Efficiency of Broadcast Method

CHAPTER VII

SUMMARY AND CONCLUSIONS

Conclusion

The previous chapters have shown that complex image rendering techniques can indeed be successfully implemented on medium-grain parallel architectures. Two of the four data decompositions researched, the node initiated method and the broadcast method, yield excellent results. With the other two methods, host initiated and node initiated with object forwarding, further research in certain areas may bring them up to more acceptable levels. Applying an algorithm such as Z-Buffering to parallel processing using four different data decompositions yielded four different results. Unlike some other areas of programming which may seem more straight-forward, if not actually cut-and-dried, developing parallel data decompositions still require that creative approach that appears almost to be art.

Future Work

A great deal of work can still be done in the area of Scene Description. RenderMan and PHIGS+ are by no means the final word in this area. As scenes become larger and larger, and component objects become more and more complex, scene description languages are going to have to evolve as well toward greater flexibility and ease of use.

The four decompositions researched in the previous chapters can all be classed static decompositions, since the decomposition does not change during execution of the scene rendering. Another possibility, less well researched, is the dynamic decompositions. These dynamic decompositions are also another method of load balancing. In the fourth decomposition, the broadcast method, if a poor load balance is determined to exist (as in the case of objects clustered at the center of the image), the dynamics could be changed so that the nodes responsible for the central portions of the frame buffer handle proportionately less work, thus equalizing the overall load balance.

Load balancing is the most visible area of this research that has the greatest effect on performance. A poor load balance can destroy even a good decomposition. If any kind of reasonable performance is to be realized from any decomposition, either static or dynamic, a suitable method of maintaining proper load balancing must be implemented.

The scene sizes tested in this research (on the order of a few million polygons) may be large today, but in a few years, they will seem small by comparison. Graphics techniques, especially parallel techniques, must come to deal with larger scene sizes.

All the areas of future research mentioned above are aimed at advancing graphics technology toward the twin goals of speed and realism. Better scene descriptions yield larger scene sizes, having greater realism. Advanced data decomposition methods yield faster rendering of the larger scenes. Load balancing ensures maximum speed-up and efficiency on the parallel architectures. As these areas are improved, both the speed and realism of image rendering graphics techniques will be advanced.

LITERATURE CITED

Catmull, Edwin, "Computer Display of Curved Surfaces," Proceedings of the Conference on Computer Graphics, (IEEE Cat. No. 75CH0981-1C), 1975, pp. 11-17.

Daniel, Ron Jr., Michael Carter, Keith Teague. "A Parallel Processing System for the iPSC/2", Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications (HCCA4), Vol. II, March 1989, pp. 1075-1078.

Denault, Damien, Eric Ryherd, John Torborg, Robert Tosi, and Ross Werner, "VLSI Drawing Processor Utilizing Multiple Parallel scan-Line Processors," Advances in Computer Graphics Hardware II, Record of Second Eurographics Workshop on Graphics Hardware, 1987, pp. 167-182.

Dyer, Scott and Scott Whitman, "A Vectorized Scan-Line Z-Buffer Rendering Algorithm," IEEE Computer Graphics & Applications, Vol 7, July 1987, pp. 34-45.

Foley, van Dam, Feiner, and Hughes, Computer Graphics: Principles and Practice, 2nd Edition, Addison-Wesley, 1990.

Fiume, E., A Fournier, and L. Rudolph, "A Parallel Scan Conversion Algorithm with Anti-Aliasing for a General-Purpose Ultracomputer," Computer Graphics, vol. 17, no. 3, pp. 141-150, July 1983.

Fox, Johnson, Lyzenga, Otto, Salmon, and Walker, Solving Problems on Concurrent Processors, Vol. I., Prentice Hall, 1988.

Ghosal, Dipak and L.M. Patnaik, "Parallel Polygon Scan Conversion Algorithms: Performance Evaluation on a Shared Bus Architecture," Computers & Graphics, Vol. 10, No. 1, 1986, pp. 7-25.

Goldsmith, Jeff, and John Salmon, "A Hypercube Ray-tracer", Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications (HCCA3), Vol. II, Jan 1988, pp. 1194-1206.

Goral, C.M., K.E. Torrance, D.P. Greenberg, and B. Battaıle, "Modeling the Interaction of Light Between Diffuse Surfaces," SIGGRAPH 84, pp. 213-222.

Hayes, John P., Computer Archıtecture and Organızation. 2nd Editıon, McGraw-Hill, 1988. pp. 660-662.

Intel, The iPSC/2 User's Guide. Intel Scientific Computers, Beaverton, OR; 1988.

Machover, Carl, "Advanced Computer Technology: The Experts' Views," Electronic Digest, Vol. 32, May 3, 1984, pp. 212-213.

Mallett, Mark, "The Challenge of Realism: Comparing the PHIGS+ and RenderMan Graphics Standards," MIPS, April 1989, pp. 101-104.

Newell, M.E., R.G. Newell, and T.L. Sancha, "A New Approach to the Shaded Picture Problem," Proceedings of the ACM National Conference, 1972, pp. 443-450.

Pıol, Alessandro, "Parallel Architecture Tackles Graphics and Image Processing," Computer Design, Vol. 26, September 1987, pp. 65-68.

Rubin, S. and T. Whitted, "A Three-Dimensional Representatıon for Fast Rendering of Complex Scenes," Computer Graphıcs, Vol 14, No. 3, July 1980, pp 110-116.

Schweıtzer, Dino and Elizabeth S. Cobb, "Scanline Rendering of Parametric Surfaces," Computer Graphics, Vol. 16, No. 3, July 1982, pp. 265-271.

Schumacker, R., B. Brand, M. Gilliland, and W. Sharp, Study for Applying Computer-Generated Images to Visual Simulatıon, Technical Report AFHRL-TR-69-14, NTIS AD700375, U.S. Air Force Human Resources Lab., Aır Force Systems Command, Brooks AFB, TX, September 1969.

Sutherland, Ivan E, Robert Sproull, and Robert Schumacker, "A Characterızation of Ten Hidden Surface Algorithms", Computıng Surveys, Vol. 6, No. 1, March 1974, pp. 3-55.

Upstıll, Steve, The RenderMan Companion: A Programmer's Guıde to Realistıc Computer Graphıcs, Addison-Wesley, 1990.

WIllıams, L., "Casting Curved Shadows on Curved Surfaces," SIGGRAPH 1978, pp. 270-274.

VITA

Eric Martin Blazek

Candidate for the Degree of

Master of Science

Thesis: PARALLEL IMAGE GENERATION USING THE Z-BUFFERING ALGORITHM ON A MEDIUM GRAINED DISTRIBUTED MEMORY MODEL COMPUTER

Major Field: Computer Science

Biographical:

Personal Data: Born in Los Angeles, California, August 30, 1966, the son of Larry and Marietta Jo Blazek.

Education: Graduated from Sapulpa High School, Sapulpa, Oklahoma, in May 1984; received Bachelor of Science Degree in Computer and Information Sciences from Oklahoma State University in December 1988; completed requirements for the Master of Science Degree at Oklahoma State University in July, 1991.

Professional Experience: Research Assistant, Departments of Computer Science and Electrical Engineering, Oklahoma State University, January, 1988, to January, 1991. Research Scientist (Junior Grade), Amoco Production Company, Tulsa Research Center, May 1988 to August 1988, May 1989 to August 1989.