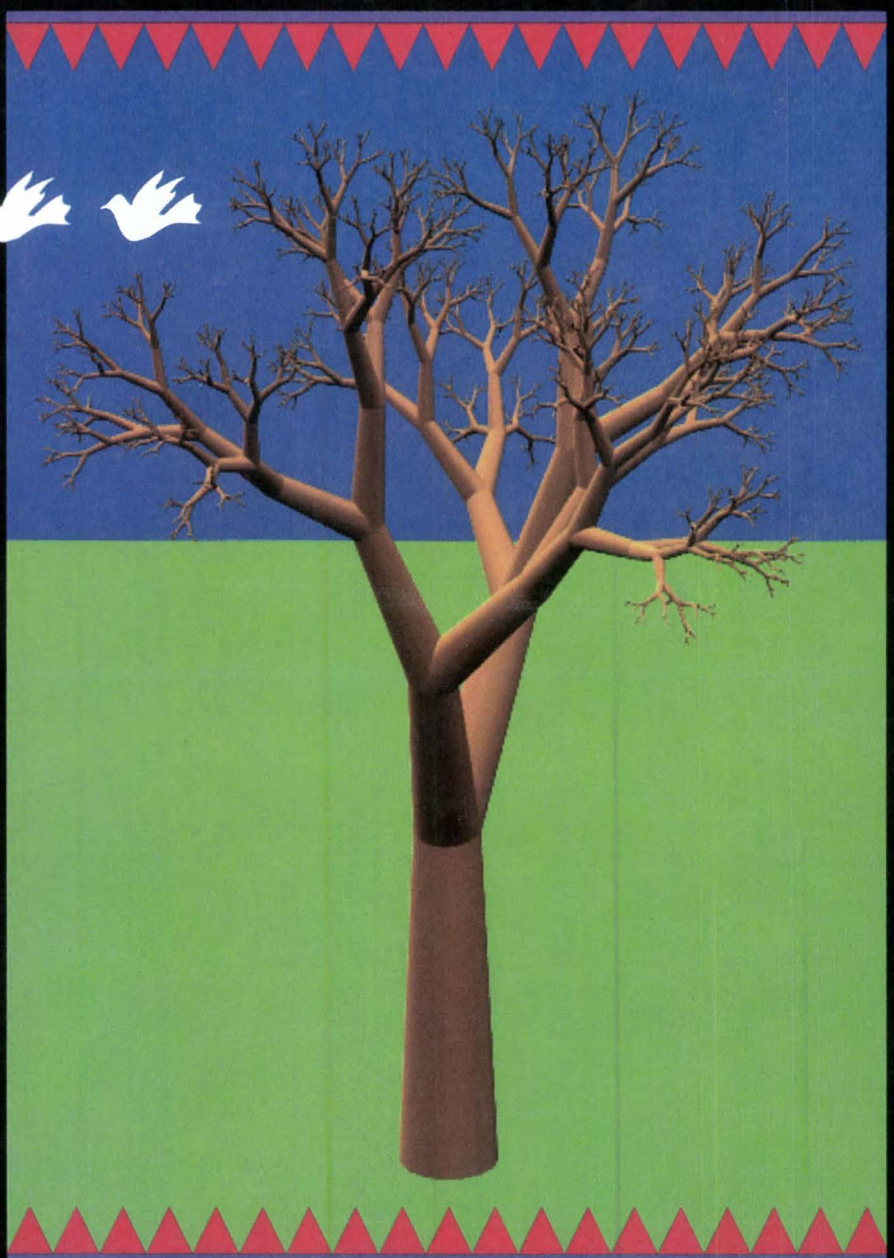
MULTIPROCESSOR METHODS FOR COMPUTER GRAPHICS RENDERING

SCOTT WHITMAN



Multiprocessor Methods for Computer Graphics Rendering

Multiprocessor Methods for Computer Graphics Rendering

Scott Whitman

Lawrence Livermore National Laboratory
Livermore, California

Jones and Bartlett Publishers
Boston London

Editorial, Sales, and Customer Service Offices
Jones and Bartlett Publishers
20 Park Plaza
Boston, MA 02116

Jones and Bartlett Publishers International
P.O. Box 1498
London W6 7RS
England

About the cover: The tree dataset was generated using Eric Haines' SPD database and rendered at a resolution of 640 by 484 pixels. The tree contains approximately 850,000 polygons and was rendered on 96 processors of a BBN TC2000 in 8.8 seconds (including specular highlights and stochastic sampling for anti-aliasing).

Library of Congress Cataloging-in-Publication Data

Whitman, Scott.
Multiprocessor methods for computer graphics rendering/ Scott Whitman.
p. cm.
Includes bibliographical references and index.
ISBN 0-86720-229-7
1. Computer graphics. 2. Microprocessors. I. Title.
T385.W54 1992
006.6'6--dc20 92-3810
CIP

ISBN 0-86720-229-7

Printed in the United States of America

96 95 94 93 92 10 9 8 7 6 5 4 3 2

To Carol,

who was not there to share the past,
but with whom I shall enjoy the future

Table of Contents

Preface

Parallel computing and computer graphics are currently two of the hottest topics in computer science. It is only natural that a merging of these two fields has now occurred in hardware architectures as well as software algorithms. This text explores a number of methods which can be used on current generation commercial multiprocessors to perform computer image synthesis. The emphasis here is on image space rendering methods since these types of algorithms will likely get the most use in the day to day work environment.

The subject matter of computer image synthesis is over 20 years old, dating back to Warnock's and Watkins' rendering algorithms, along with Gouraud's lighting model. Since then, many refinements have been developed which use advanced hardware and software techniques to hasten the rendering computation. The availability and price/performance ratio of commercial multiprocessors makes them attractive for development of general purpose computer graphics algorithms. When parallel computer architectures became commercially available in the mid-1980s, the sequential programs that had been previously developed for computer graphics rendering were in need of a re-evaluation for a parallel context. In addition, it was questioned whether new and completely different rendering programs would be required for use on these computers. In this book, we examine previous and current solutions to the computer image generation problem presented by a variety of researchers. Several of these solutions, along with a number of newly developed algorithms by the author, are analyzed according to their performance on a scalable multiprocessor.

The problem of quickly generating three-dimensional synthetic imagery has remained challenging for computer graphics researchers due to the large amount of data which is processed and the complexity of the calculations involved. For instance, in a multiprocessor, one needs to minimize the communication of data between processors so that the majority of the execution time is spent on computations. The large datasets inherent in computer graphics scenery do not lend themselves to ease of partitioning among processors. Tradeoffs between synchronization, load balancing, and communication must be made during algorithm development and refinement in order to effectively utilize the resources available in the system. These issues are discussed in detail in this text with regard to the parallel algorithms which were implemented on the BBN Butterfly family of

computers. Although the algorithms were developed for these machines, they could be modified with minimal effort to work on any general purpose multiprocessor. Unfortunately, the time to modify and test the code on a variety of machines would be prohibitive, especially to the degree used in the latter part of this book. It is hoped that the insights presented here along with the various issues raised, and will be informative as both a guide for implementation and a reference to methods of attacking this problem.

In the first chapter of this book, an overview of computer graphics rendering is provided, and the issues that are of importance to the fields of computer graphics and parallel processing are noted. The second chapter provides a historical reference to previous efforts in this field. Each of these is categorized into a taxonomy to indicate what algorithmic methods each work has utilized. Most of this research involved simulations of parallel environments whereas this book provides an analysis of actual implementations on general purpose commercially available multiprocessors. The third chapter analyzes the various multiprocessor architectures with regard to graphics rendering algorithms. In chapter 4, the basis parallel algorithm is presented, along with the procedures used for testing and performance analysis. Chapter 5 includes descriptions and analyses of each of the work decomposition methods which were implemented. The analysis is a scrutiny of a given program's parallel performance which provides information to the reader on exactly why each algorithm performed the way it did. There were two main choices for storing the graphics data in main memory, and these are analyzed in chapter 6. The first is a shared memory paradigm while the second, although using shared memory, takes advantage of local memory on each processor to reduce latency. The results for all of the algorithms are compared on a variety of imagery to convince the reader that the results presented are representative of real world expected performance.

Acknowledgments. This book was originally a doctoral dissertation written and researched while I was at The Ohio State University. My dissertation committee of Richard Parent, P. Sadayappan, and D. Jayasimha helped to guide me through the difficult phases of my research. I am indebted to my committee for the countless hours of useful discussions and comments that they provided me on my dissertation. Others who helped to make my stay at Ohio State that much more rewarding include Scott Dyer, Doug Roble, Manas Mandal, as well as my other colleagues in the Computer and Information Science Department, the Advanced Computing Center for Arts and Design, and the Ohio Supercomputer Center.

This research was conducted over a period of several years and utilized literally thousands of hours of computer time. The author would like to acknowledge the institutions and staff at BBN Advanced Computers, Inc., Argonne National Laboratory, and Lawrence Livermore National Laboratory for allowing their machines to be used for benchmarking and testing purposes. Individuals deserving special recognition for their assistance include: Ed Forbes, John Price, Linda Woods, and Eugene Brooks.

Scott R. Whitman

1

Introduction

High quality computer graphics imagery is used in a wide variety of fields in society today. Most people are familiar with the entertainment uses of computer graphics which span the artistic realm and include two-dimensional imagery using paintbox systems, three-dimensional surreal scenes for aesthetic prints, and 2D and 3D animation sequences for use in the video and film industry. There are many major motion pictures which rely on computer graphics rendering to achieve cost effective special effects. The quality of this imagery has risen to such a high level that the public is accustomed to seeing on a regular basis computer generated commercials of photorealistic caliber. In addition, applications such as CAD/CAM, finite element modeling, flight simulation, and molecular modeling use computer graphics to aid in the visualization of scientific and industrial data. The demand for higher quality images from these applications has grown as computer time has become less expensive. Even though faster computers are now available in reference to the past, the time to generate a typical image has not really decreased due to the more elaborate imagery required. Deering [Deer88] noted that "an increase in graphics performance is more likely to cause users to display more complex objects, rather than the same objects faster." A computer graphics display algorithm must be able to

handle this highly complex imagery in an efficient manner. One solution to this problem involves utilizing parallel computer architectures to render the graphics image. If an efficient software algorithm is employed on this type of machine, performance will increase with the number of processors added to the system.

This book examines techniques which utilize parallel processing to accelerate the computations necessary for rendering three-dimensional computer graphics scenes. The most promising algorithms are developed and quantitatively compared under a variety of circumstances to ascertain which has the highest performance.

The basic problem in computer image synthesis of 3D scenes is outlined in the first section of this chapter. Here, the components of a computer graphics display algorithm are described. The terms *hidden surface removal* and *rendering* are defined in the context of a computer graphics display program. The second section presents a brief overview of the research which has been done in the area of developing parallel computer graphics rendering techniques. This research can be broadly grouped into two categories: hardware and software based solutions. The hardware based solutions typically involve designing custom VLSI chips to transform and display data in near real-time. *Real-time* is a term used to describe calculations which can proceed within the update rate for a single frame on a CRT monitor, typically 30 frames per second. Software solutions use high performance advanced computer architectures to achieve fast computer graphics renderings. The goals of each of these methods are described in some detail in this section. The third section outlines the area of research which this book covers. In this section, the context of this work in both the parallel processing and computer graphics communities is stated. Finally, the fourth section provides an overview of the rest of the text.

1.1. Problem Description

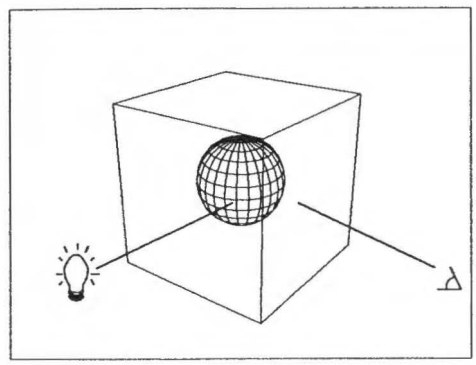
Computer graphics imagery can serve many purposes, but the basic computer program used to generate these images is the same, regardless of the intended application. The input data consists of a set of objects which are described both geometrically and topologically, using a polygonal format. Various scene parameters are also input to describe the lights, shading, color, and other information regarding how the objects should appear in the computer synthesized scene. All data is input as x, y, z floating point variables. The input datasets are assumed to contain closed planar polygons. The output of a

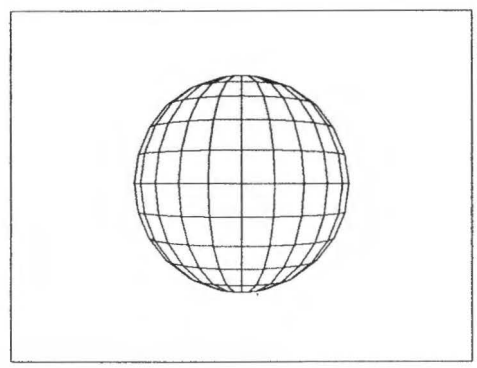
graphics display algorithm is a rendering of a three-dimensional scene, taking into account realistic lighting and object attributes. This output is an image in the form of picture elements (pixels) which may be displayed immediately on a frame buffer color monitor or stored on hard disk for later display. A *frame buffer* is a dynamic memory collection of pixels containing red, green, and blue components. Each pixel component is usually 8 bits deep allowing for a choice of 256^3 or approximately 16 million colors.

In general, a computer graphics display algorithm which generates images of three-dimensional data consists of the following phases:

1. Read-in polygonal data from disk.
2. Transform data from object space to eye space.
3. Clip and perform perspective projection of the data.
4. Remove hidden surfaces so only displayable surfaces are seen.
5. Render surface data using an illumination model.
6. Calculate special visual effects such as anti-aliasing or texture mapping.
7. Write pixel data to the frame buffer for display or to a file for storage.

The overall algorithm is shown in detail in figure 1.1. The top diagram indicates the world space three-dimensional view of a sphere dataset composed of quadrilateral polygons. The sphere is initially described in its own 3D coordinate system (object space). The scene as a whole consists of a collection of objects in 3D space relative to each other. In addition, an eye point and light sources are present in the scene (world space or eye space). In order for the graphics program to display the scene on a two-dimensional screen, each object must be transformed to screen space and (if necessary) clipped to the borders of the screen. The middle diagram is the same sphere after 3D to 2D transformations, clipping, and perspective operations have been applied. The bulk of the work in the program occurs in the rendering phase. This amounts to taking into account the position of the eye and each light source in relation to the objects in the scene, and then accurately displaying the objects according to their surface geometry. Incorporated here are such operations as hidden surface removal, illumination modeling, anti-aliasing, and other visual effects.





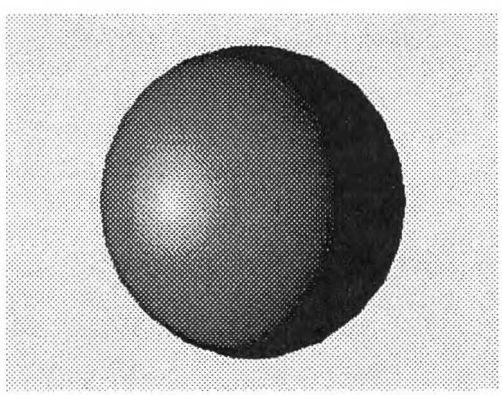


Figure 1.1: Graphics rendering pipeline

These operations are elaborated upon in the sub-sections that follow. This process is shown in the bottom diagram as the final rendered and shaded image which takes into account the location of the light source, object, and eye position.

The problem domain of this book focuses primarily on the *tiling* portion of a graphics display algorithm (steps 4, 5, and 6). Methods to speed up both the front end (reading in, transforming of data) as well as the back end (writing out pixels) of the program are also investigated. The assumption here is that the nature of the input and output is unique to each application, while tiling is the same for the majority of applications. Because the tiling operations constitute the bulk of the computation in this type of program, it is worthwhile to concentrate one's efforts on this section of a graphics rendering algorithm. Steps 4, 5, and 6 are described in more detail next.

1.1.1. Hidden Surface Removal

Hidden surface removal consists of determining which surface element in the synthetic 3D scene is closest to the observer for each pixel on a CRT screen. There are a variety of techniques for solving this problem, and most take advantage of some form of coherence in the image in order to reduce the amount of computation. In Sutherland, Sproull, and Schumacker's landmark paper [Suth74], graphical coherence is defined as "the extent to which the environment or the picture of it is locally constant." For instance, scan line coherence refers to the fact that successive lines of pixels do not differ greatly in the data displayed, so that incremental calculations can be used to achieve faster processing. Sutherland et al. point out that "all of the [display] algorithms capitalize on various forms of coherence to reduce to manageable proportions the work of sorting." The exploitation of image coherence in a parallel setting poses a challenging problem. The use of coherence reduces the amount of computation in a sequential machine by using results from previously computed parameters when generating new values. The independent parallel generation of these parameter values in the image implies redundant recomputation and the loss of coherence. The tradeoff between parallelism and coherence is an important issue that is studied here.

1.1.2. Rendering and Special Effects

Rendering is a method for displaying polygonal or bicubic patch surfaces on a frame buffer monitor so that the overall surface geome-

try is approximated and lighting in the scene is taken into account. Rendering techniques include illumination models such as Gouraud [Gour71] and Phong [Phon75] shading, which are used to simulate smooth surfaces. Using Lambert's law and approximations to the normal vector of the surface at each pixel, the data can be displayed accurately on the screen. Computer graphics special effects add realism to a computer generated scene. Some of these include: calculating refractions of transparent objects, modeling of wrinkled surfaces (bump mapping), applying texture to a surface (texture mapping), and accounting for shadows in a scene.

Anti-aliasing is another added visual effect which removes the jagged or staircase edges which appear at surface boundaries due to discrete sampling of the analog dataset. Both rendering and special effects are closely tied because the addition of visual features normally occurs during the rendering process. Current display methods incorporate advanced rendering and visual effects as an integral part of the algorithm. The complexity of these computations in most cases overrides those necessary for hidden surface removal. Any techniques used to speed up the image generation process must concentrate heavily on the rendering and visual effects stages.

In the next section, the background on a number of hardware and software graphics techniques is given.

1.2. Overview of Accelerated Rendering Techniques

Although significant work has been done in the past regarding the sequential computer graphics image generation problem, it is necessary to re-investigate this problem to see what changes or alternate approaches are necessary for parallel implementation. Work in this area has centered around both hardware based graphics workstations and software solutions for parallel machines.

Numerous companies have developed graphics superworkstations which incorporate special purpose chips along with multiple processors to achieve a high performance visual computing system. Initial developments in this area involved the use of special purpose graphics terminals which manipulated wire-frame images in real-time. Wire-frame imagery only shows the outline of the dataset surfaces and makes use of phases 1 through 4 given in the beginning of this section. Using today's technology, more sophisticated machines can generate smoothed surface representations in near real-time to aid in visualizing data. The term "real-time" generally refers

to an update rate of at least 10 frames per second (fps). Standard video update rate is 30 fps while film is 24 fps.

Commercial machines of this type include the Apollo DN10000VS [Kirk90], the Silicon Graphics 4D VGX [Haeb90], the Stellar Graphics Supercomputer GS1000 [Apga88], and the Ardent Titan [Died88]. All of these machines support parallelism with typically up to 4 processors, while the Stellar and Ardent architectures employ parallel processing at both the MIMD (multiple instruction, multiple data path) and SIMD (single instruction, multiple data path) levels. MIMD refers to the fact that each processor is executing a set of instructions asynchronously from other processors. SIMD refers to a central processor controlling execution or to a vector pipeline architecture. In addition, fast rendering engine processors are coupled with the frame buffer in these machines to achieve high speed generation of images. The Silicon Graphics and Apollo machines support anti-aliasing and texture mapping. The Apollo DN10000VS uses quadratic interpolation to help alleviate Mach bands, although this technique is not quite as good as true Phong shading. Mach bands (see [Roge85]) can occur when the smooth surface interpolation in the illumination model is not an accurate representation of the actual surface. The Silicon Graphics 4D VGX has what is called an "accumulation buffer." This buffer allows such features as motion blur, soft shadows, depth of field, and anti-aliasing to be performed on a polygonal database. Motion blur smooths out the motion of fast moving objects in a scene. Soft shadows provide a smoothing effect to the shadow that simulates a penumbra rather than the typical quick cutoff that is apparent in conventional shadow algorithms. Depth of field simulates the way a camera lens focuses. Other hardware approaches including new chip designs from Schlumberger [Deer88] and IBM [Ghar88] promise high graphics performance for the future.

An example of using a hardware architecture to solve the radiosity problem is given by Baum and Winget [Baum90]. Radiosity is a very computationally expensive technique for visualizing 3D scenes. It is essentially an n^2 problem which involves calculating the diffuse inter-reflection of all surfaces against one another so that the light reflectance of the entire scene is taken into account. In their algorithm, Baum and Winget use the hardware capability of the Silicon Graphics IRIS workstation to perform real-time radiosity. Their algorithm exploits the hardware by using the Z-buffer rendering feature of the IRIS to calculate the form factors in parallel. The Z-buffer is a contiguous memory which holds the Z coordinate value of the closest surface to the viewer for each pixel on the screen. Additional work by Garlick et al. [Garl90] using the IRIS workstation

allows one to manipulate very large databases in real-time. This algorithm works by using parallel processing to perform clipping operations necessary to observe the dataset. Although both of these implementations are useful, they do not deal directly with the problem of image generation.

Two architectures which are primarily intended for fast image processing as well as 3D rendering are Pixel Planes and the AT&T Pixel Machine. These machines are designed to offload the graphics calculations from a host computer; they are not intended for use as workstations.

Fuchs et al. [Fuch85] introduced their hardware approach to solving the visualization problem in 1985. Fuchs' team designed Pixel Planes, a parallel architecture containing a processor at every pixel, and a binary tree of adders optimized to solve the equation $F(x,y) = Ax + By + C$ at each pixel. This machine also has hardware support for calculating anti-aliasing, shadows, and texturing.

The AT&T Pixel Machine [Potm89] contains a high performance network of processors with a fine-grained interleaved frame buffer. That is, the frame buffer memory is scattered throughout the processors. This alleviates contention while providing sufficient throughput. It is typically used as a graphics engine which offloads complex rendering calculations from a host computer. With a full configuration of 64 rendering processors, 820 MFLOPS peak performance is attainable. Since this machine is a general purpose graphics machine, software algorithms can be used to take advantage of its characteristics. Although the Pixel Machine can be programmed to handle a number of different graphics display methods, its versatility is limited as a general purpose computer primarily because of the small amount of memory available at each processor (only 64kbytes).

The solutions described above involve integrating a special purpose graphics rendering engine into a high performance workstation or using a hardware assisted graphics accelerator. The first approach yields a near real-time update of polygonal based scenes, which is useful to designers and engineers. The second approach offloads the host for external graphics processing. Even within this realm, the designs suffer limitations. For instance, if anti-aliasing and other features are used, performance degrades dramatically. Quantitative measures of the degradation which occurs when applying anti-aliasing to polygonal models in these machines are not available. True Phong shading is not present in the hardware of any of these machines. Most of the hardware methods employ a Z-buffer type of hidden surface removal algorithm but the data must be

stored in the memory of the machine prior to loading into the graphics pipeline. A *Z-buffer* [Catm74] is analogous to the frame buffer except that the z coordinate for a polygon at the given pixel is stored in memory. This is a simple technique used for hidden surface removal. Extremely large datasets are not able to fit into the physical memory of the machine and consequently performance suffers as a result of disk access. As a result of these limitations, a hardware approach is only adequate for interactive use with a small to medium size dataset (typically 10,000 polygons or less). To achieve reasonable performance on large datasets, a parallel software approach to solving the rendering problem is warranted.

The use of a general purpose multiprocessor computer is more cost effective than the specially designed architectures, since this type of machine can be used for non-graphics applications as well. The software method may not have the capability for real-time calculations, but this is not needed in many applications. In addition, a graphics workstation is not capable of the high performance general computing required by applications which demand supercomputer cycles. By integrating the graphics rendering with the application and using the same computer for both simultaneously, it is unnecessary to send the data to a separate machine for graphics rendering. Taking this a step further, we expect that future generation multiprocessors may in fact offer the capability to achieve real-time computer graphics rendering. Following is a description of how this might be used.

For real-time interaction with a complex illumination model, the user is generally limited to a small number of polygons on even the most advanced graphics workstations. With the recent interest in scientific visualization, scientists would like to be able to see their scientific data using real-time interaction, while adjusting their simulation simultaneously. The simulation portion of the code is usually run on a supercomputer class architecture machine. Example applications which require this level of computer power include: molecular dynamics simulations, 3D finite element simulations, and global climate modeling. Massively parallel architectures hold great promise for being able to support applications of this type. In addition, the capability to support real-time interaction of a dense database containing perhaps a million elements is beyond the scope of even the most powerful graphics workstations. Consequently, it is natural to incorporate the graphics rendering operations along with the simulation program in the same computer so that the coupled system can output the graphics image in real-time. This desired interactive environment has come to be known as *simulation steering*.

It is expected that massively parallel architectures will provide the capability to accomplish steering before the end of the 1990s [Upso89]. This book gives insight into how graphics rendering programs will be developed for massively parallel architectures to incorporate this desired feature in the near future. Already, some researchers are looking into using SIMD architectures for such a purpose [Smal89], [Schr91]. Although these machines are likely to provide decent results, it is generally believed that the long term prospects for real-time interactive simulation steering can only be achieved by future generation high performance MIMD computers.

There have been numerous software algorithms presented in the past that have been designed for many different types of advanced architectures. An overview of algorithms of this type is provided by Whitman and Parent [Whit88]. In addition, Crow [Crow88a] provides insight into commercial ventures and other interesting methods used for designing parallel software approaches to display computer graphics images.

Previous work in software algorithms for parallel graphics rendering has primarily concentrated on software simulations or simple ad-hoc solutions. Little work has been done in this area to fully exploit parallel processing at a high level. Some parallel graphics display solutions have dealt with a graphics rendering technique known as ray tracing [Whit80]. *Ray tracing* is a technique which involves sending rays from the observer through each pixel to intersect the objects in the scene.

The advantage of ray tracing is that features such as reflections, refractions, shadowing, motion blur, and depth of field are very easy to implement. On the other hand, an image generated by ray tracing takes several orders of magnitude more time to compute than one which is generated by a conventional image space graphics display algorithm such as the scan line Z-buffer or Watkins' algorithm [Roge85]. Badouel [Bado90] and Green [Gree89] both present a fairly good treatment of ray tracing in parallel on a message passing multiprocessor. Although more work could be done in this area, the analysis in this book is restricted to the more efficient image space rendering algorithms. However, some ray tracing algorithms are presented in the next chapter to illustrate the work that has been done in this area. In the next section, a description of the context of this book in the fields of computer graphics and parallel processing is given.

1.3. Research Context

This text presents an analysis of the most efficient methods for the generation of computer graphics imagery on multiprocessors. Past approaches to parallelizing graphics display algorithms were not designed to take full advantage of the machine architecture. In this book, a variety of techniques are investigated which exploit parallelism in computer graphics image generation. The intention here is to evaluate high performance solutions which perform well on a massively parallel computer. Previously developed serial algorithms are examined for potential parallel extensions. In addition, new parallel approaches to generating computer graphic images are studied to evaluate the methods most suitable for implementation.

A number of different memory referencing strategies are also compared and analyzed on a parallel computer. Jamieson [Jami87] discusses a variety of algorithm and architecture characteristics, and presents guidelines for determining how to fit an algorithm to an architecture. To ascertain the appropriate choice of architecture for implementation purposes, a number of commercial parallel machines are compared in the context of developing a graphics rendering program. A particular computer graphics algorithm may not be well suited to all architectures, however. Therefore, different approaches are categorized according to a number of characteristics in order to obtain a suitable mapping of algorithm to architecture. In evaluating an implementation of a parallel graphics algorithm on a given architecture, various factors that degrade program performance are quantified. These factors help the reader to understand the characteristics specific to the different algorithms.

Most of the previous work in this area by computer graphics researchers involves one of the following procedures: analysis of parallel architectures for graphics, simulation of a parallel machine in software on a von-Neumann architecture, or presentation of an initial software study on a multiprocessor architecture. This text extends the work of others by including detailed comparisons of a number of different algorithmic techniques as implemented on an existing commercial multiprocessor.

1.3.1. Graphics Context

Some issues with regard to this subject matter that various computer graphics specialists have addressed in the past include: 1) SIMD ap-

proaches [Dyer87], [Crow88b], [Smal89], [Theo89a], 2) coherence vs. parallelism [Kapl79] (for spatial subdivision algorithms), 3) methods of spatial subdivision [Whel85], [Kapl79], [Hu85], and 4) effect of larger datasets [Whel85]. This is by no means a complete list of the research that has been done in this area. In fact, Burke and Leler [Burk90] present a fairly thorough examination of previous work in this field. The first of these items is not addressed in this text since we are primarily interested in MIMD algorithms here. In chapter 2 we present in more detail the choice of architecture and defer a discussion on this matter to that point. The second item, coherence versus parallelism, is mentioned only briefly in various papers, but has not been analyzed extensively to see to what degree it is worthwhile maintaining graphical coherence in a parallel algorithm. The third item, spatial subdivision methods, has been looked at by the most researchers, but there are other methods that have not been considered. Also, most of the previous work is based on simulations rather than actual implementations. The fourth item, large datasets, is treated fairly completely by Whelan, although his work involves a simulation rather than an implementation.

By comparing implementations, one can determine which parallel task decompositions provide good performance on an actual machine. In these implementations, various parameters such as the number of tasks assigned per processor can be varied to see how performance changes in practice. Memory partitioning and referencing schemes that have not been addressed in any previous work are discussed as well. In a data intensive program such as a graphics display algorithm, the most efficient method for data storage and access cannot be easily determined. This may relate strongly to the type of architecture that the algorithms are implemented on, and should be taken into account as well.

The effect of other parameters such as image complexity and machine characteristics are analyzed. Very little work has taken place on developing a scalable parallel display algorithm. While it may be true that some modifications may be necessary to obtain high performance on upwards of 1024 processors, the goal here is to effectively utilize as little as 2 and as many as 100 or more processors, with no modification in the code.

The work presented in this text can also provide hints to other programmers developing graphics algorithms for parallel environments. For instance, little work has been done in the past regarding radiosity or volume rendering on parallel architectures. Both of these rendering techniques involve large datasets and random memory access. The decomposition and memory referencing schemes

developed here may be suitable for extension to these types of applications.

1.3.2. Parallel Computing Context

In relation to how this work fits into a parallel computing context, one should note the following factors. A graphics display program is a fairly detailed program which typically has 5000 or more lines of code. The complexity and size of the data structures used in this application make it fairly difficult to deal with in a straightforward manner. One must determine if a given data item is to be:

1. Read-accessible to all processors.
2. Read-accessible to a limited number of processors.
3. Write-accessible to all processors.
4. Write-accessible to a limited number of processors.

For example, there are several storage methods and issues relevant for a read-accessible data item. The item may be copied to all processors that may reference it, but the overhead of copying as well as excessive memory use may prohibit the usefulness of this approach. It might be possible to regenerate the item each time it is needed on a given processor, but there is a cost associated when this is done. Another alternative could be to remotely reference an item stored in shared memory, but this causes latency problems. These issues are investigated in chapter 5 of this book.

These alternatives bring to light one of the key issues in any computer program: what is the balance between storage and speed that can be best utilized in this implementation? The use of multiple processors re-opens this issue to a whole new set of potential problems. A graphics application uses data items that fit into all four categories, theoretically requiring a decision regarding storage and access for each data item. In reality, it is fairly straightforward for most data to see what storage method would be best. On the other hand, the algorithms do impart some characteristics on memory referencing which may force different design decisions. A balance must be struck in order to obtain good performance under a variety of circumstances.

An in-depth analysis is presented in chapter 4 regarding different issues which relate to parallel programming in the context of this application. These include overheads encountered in a parallel program that are not present in a serial version such as: contention, use of virtual memory in parallel, scheduling, communication, and

synchronization, to name a few. These are quantified as to their effect on the overall program performance.

While numerical applications such as LU decomposition and matrix multiplication may involve large amounts of data movement, typically somewhat simpler data structures are used than those required for a graphics display algorithm. The data structures in numerical applications are usually multi-dimensional arrays. In a graphics display algorithm, the polygonal data is initially read into array data structures. After this initial phase, though, the data needs to be maneuvered into complex hierarchical data structures. These can consist of objects, polygon information lists, active edge lists, edge pair data structures, and many other intricate storage mechanisms. For a sequential environment, there is not general agreement among the graphics community about which type of data structure is the most efficient for a particular algorithm. There is certainly room for discussion as to the most suitable data structures for a parallel environment. The parallel architecture influences the decision as to the choice of data structures as well. This decision is especially crucial in a graphics algorithm where there may be a large amount of data needed for any given task. The memory resource in a parallel computer is not infinite, so the data structures must be time and space efficient as well.

One issue not encountered in numerical parallel algorithms is that of parallelism versus graphical coherence. While this is typically a graphics issue, it can be thought of as a parallel computing issue as well, in that different overheads are incurred depending on the task granularity chosen. These overheads are basically due to the lack of coherence induced by separating tasks for execution in parallel. As such, one must investigate to what degree this overhead affects performance insofar as determining the number of tasks to generate.

1.4. Document Overview

In chapter 2, a framework is developed for analyzing parallel graphics display algorithms. A taxonomy of parallel graphics display algorithms is generated in which the possible parallel approaches are categorized into image and object space methods. Previous researchers' work is fit into this taxonomy, and a number of new approaches are examined which could be used to solve the problem. By considering a number of issues relating to parallel algorithm development, it is shown that several types of approaches are worth further consideration.

In chapter 3, a number of multiprocessor architectures are presented in order to determine the one most suitable for implementation. It is easiest to use a previously developed serial display algorithm as a basis for the parallel implementation due to the nature of the architecture chosen. An evaluation of the different MIMD programming models is discussed, including the programming paradigms available on the BBN Butterfly, on which the algorithms were implemented.

Chapter 4 discusses the overall basis graphics display algorithm and how it applies to the chosen architecture. The design decisions which are common to all the implemented parallel algorithms are described. In this way, one can see that it is easy to compare parallel approaches since they are all based on the same code. Finally, the testing procedures which are used in timing the various algorithms on the GP1000 are described.

In chapter 5, a number of different parallel image space subdivision algorithms are presented, based on the pixel decomposition scheme. In this chapter, only the tiling section of the algorithms is compared in order to evaluate the maximum parallelism attainable. For comparison purposes, the size and number of areas is varied. In addition, three different task partitioning schemes are compared. The results are scrutinized, and the overhead percentage factors are determined through experiments in the performance of each parallel algorithm.

In chapter 6, several shared memory storage and referencing schemes are examined. A global storage and referencing scheme is compared to a software caching scheme. Due to the fact that the algorithms for task decomposition affect the memory storage and reference scheme, the overheads involved in implementing each scheme are examined. The various algorithms which are scrutinized in chapter 5 are compared again, but this time the setup cost is included, not just the tiling section cost. Based on this comparison, the task adaptive algorithm utilizing the locally cached memory referencing scheme resulted in the best timings. The issues of parameter variation are investigated in actual implementations of these algorithms. Both the machine and scene parameters can vary, and these variations can change the algorithms' performance.

Chapter 7 presents overall conclusions and discusses future research possibilities.

2

Overview of Parallel Methods for Image Generation

In this chapter, a number of methods which can be used for parallel graphics rendering are discussed and evaluated for their applicability to multiprocessor architectures.

In the first section, a number of factors are presented to serve as a basis for a quantitative analysis of potential algorithms for implementation. The second section presents a historical overview of previous work in the area of parallel graphics algorithms, and these algorithms are categorized and presented in a taxonomy. In addition, new methods are also described and fit into the taxonomy as well.

2.1. Criteria for Evaluation of Parallel Graphics Display Algorithms

A number of parallel approaches to graphics rendering have been developed in the past, and more are certain to be presented in the future. In order to effectively evaluate these different approaches, it

is worthwhile analyzing them in terms of a number of important issues including:

1. Level of granularity of task sizes.
2. Nature of algorithm decomposition into parallel tasks.
3. Utilization of parallelism in the display algorithm without significant loss of coherence.
4. Load balancing of tasks.
5. Distribution and access of data through the communication network.
6. Scalability of the algorithm on larger machines.

The interrelations of each of these issues and how they effect the overall parallel algorithm are shown in figure 2.1. These issues are investigated in the context of a number of previous approaches to the parallel image generation problem. They are described in more detail next.

2.1.1. Load Balancing

Load balancing refers to the idea that each processor is used as effectively as its neighbors. This means that in the ideal case, each processor has exactly the same amount of work and will finish its work at the same time as the others. Researchers typically address this issue by developing task partitioning schemes which attempt to create an even load among the processors in one of two ways: either

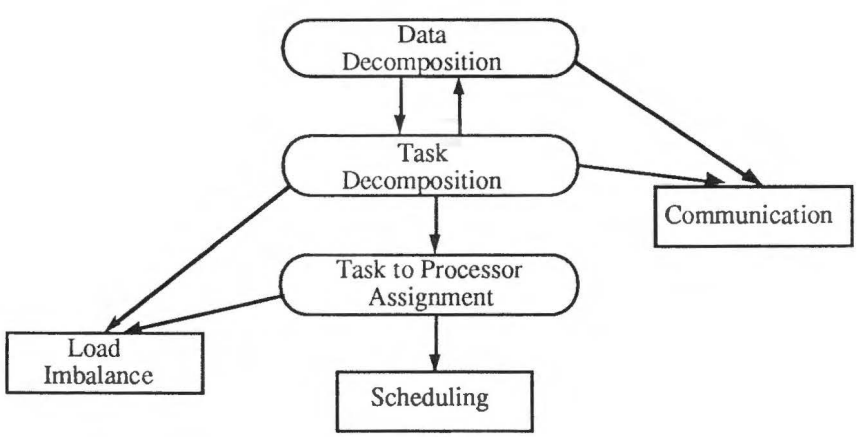


Figure 2.1: Relationship of decomposition methods to parallel overheads

by static assignment of large tasks or by dynamic assignment of smaller tasks.

In the static method of task decomposition, the number of tasks T is typically equal to the number of processors P, and all of the tasks are estimated to take approximately the same amount of time. This requires some additional overhead prior to starting a parallel environment, but the hope is that an even work distribution will result. The advantages of this method are: communication overhead percentage is small due to larger task sizes, task startup overhead is minimized, and scheduling overhead is reduced.

The second method of attacking load balancing is the dynamic approach. In this method, T is determined to be much greater than P, and task assignment to processors proceeds during runtime. A processor continues to work on tasks until no more work is available, at which point it remains idle until all of the processors complete their work. As a result of small task sizes, the idle time is small and will have minimal impact on overall performance. Previous research has shied away from this approach in hardware designs for graphics rendering because it was deemed that the context switching resulted in too much overhead. In a software algorithm, this is not a consideration unless the granularity of these tasks is too fine. If too fine a task granularity is used, it is possible that the time to obtain a new task is too high a percentage of the task execution time, degrading overall program performance. The advantages of the dynamic approach include: 1) task execution time does not need to be determined a priori; 2) load balancing is solved in a dynamic manner; and 3) the time needed to determine what the work will be for each task is much smaller than in the static method. Note that some methods employ a static scheduling of ($T > P$) tasks as well. However, load balancing is not handled directly in this type of algorithm.

2.1.2. Levels of Granularity

As discussed previously, the parallel decomposition of a computer graphics algorithm can occur at many different levels of granularity. It is necessary to determine the potential number of parallel tasks to identify independent calculations which can be performed in parallel. The partitioning of a display algorithm may be performed in terms of either image space or the object space. A single display algorithm can use any combination of any of these levels to partition the computation. The different levels of granularity are given in table 2.1.

If an algorithm is divided into tasks that are too coarse grained, load balancing will suffer since not enough parallelism is introduced. On the other hand, if too fine a level of granularity is used, then too much context switching will occur which adds time to the parallel program. It seems clear that a medium grain approach is the most viable since it strikes a balance between providing good load balancing and minimizing context switching.

2.1.3. Nature of Parallelism

There are two principal types of methods for decomposing algorithms for a parallel computer--data and functional parallelism. *Data parallelism* refers to dividing up the data among the processors and processing different data segments in parallel. *Functional parallelism* usually involves different threads of control and can be further broken down into operational and procedural levels. *Operational parallelism* refers to concurrency at the basic operations level such as assignment, etc. *Procedural parallelism* is achieved by decomposing the algorithm into sections which are assigned to different processors. *Pipelining* is a form of parallelism that combines features of data level parallelism with functional level parallelism. Although data parallelism is normally associated with SIMD architectures, an MIMD approach can also employ data parallelism in

Table 2.1: Granularity levels in parallelism and computer graphics

Granularity	Program Constructs	Graphics Entities
Very Coarse	programs running on different machines via network	calculation of separate images on different machines at the same time
Coarse	execution of P modules in parallel on P processors	sub-division of scene into objects or groups of objects
Medium	execution of N modules on P processors in parallel where $(N >> P)$	sub-division of image into sections or sub-division of objects into faces
Fine	parallel computation of loop iterations in SIMD pipeline	parallel processing of groups of pixels or span segments assigning one group per processor
Very Fine	hardware parallelism at instruction level	assignment of processors to calculations at the pixel level

which work is partitioned into parallel components according to the input dataset. Alternative schemes involve using functional parallelism or some combination of functional and data parallelism. The type of parallelism evident in each algorithm is identified as each is discussed since different stages of an algorithm may use different levels of parallelism.

2.1.4. Usage of Graphical Coherence

Recall that graphical coherence is the use of incremental operations rather than recomputation of parameters to hasten the speed of graphics calculations. A major component of every three-dimensional computer graphics display algorithm is sorting data elements in some combination of the x, y, and z directions in three-dimensional space. The advantage of using coherence in this type of algorithm is that sorting can usually be reduced to incremental calculations rather than recomputation of various parameters. Coherence can be examined within computer graphic images at the pixel level, scan line level, area level, or frame level. For example, *scan line coherence* refers to the fact that edges of polygons intersect a number of adjacent scan lines. When edge parameter values such as color or surface normal are calculated for the initial scan line which the edge crosses, the incremental values can be computed and used to update the parameters from one scan line to the next. This can also be used in a sorting context in hidden surface removal algorithms such as Watkins' algorithm [Roge85] to update which polygon span segments are in front of each other for a given set of scan lines. Other uses of coherence rely on knowledge obtained earlier in the computation to reduce calculations in the generation of the image.

In parallel computing, the approach usually taken for task decomposition is to partition the computation among different processors. This would mean that one could not necessarily rely on values calculated earlier in the computation for later use, as is usually done when exploiting coherence. If coherence is not exploited, redundant calculations are performed and the overall computation time will increase. In order to solve this apparent paradox in a parallel environment, it is worthwhile to investigate possible methods of parallelizing computer graphics display algorithms which maintain coherence. In the taxonomy in section 2.2, the type of coherence which each algorithm exploits in a serial implementation is noted, and we determine the method most suitable for a parallel implementation.

2.1.5. Data Access

One of the key issues in a parallel graphics display algorithm concerns movement of data between memory modules, as well as to and from the disk. Graphics display algorithms use a huge amount of memory, and memory management is important to the overall performance of the algorithm. Remote access of shared data will slow down an algorithm, so data locality should be taken into account when possible. Most algorithms developed in the past were based on simulations rather than implementations on actual multiprocessors, and little attention was placed on data access. In some cases, the given algorithm enforces a certain type of access pattern, but in general, the algorithms can be modified to use any particular type of memory access.

Since datasets representing complex graphics scenes are generally large, it is not feasible to copy the entire dataset onto each node of a multiprocessor. Besides the fact that space may be a limitation, it would not necessarily be desirable to copy all of the data since the time taken to do so on a massively parallel machine would be rather lengthy. Although such a complete replication of data is potentially feasible for read-only data through a one-time broadcast, simple replication cannot be used for read-write data. An example of a read-write data structure in a graphics application is the frame buffer memory used to store the pixel color information. This type of data structure must be partitioned among the memory modules. Of course, one could duplicate this data structure on every processor and perform a parallel merge operation at the end. This would require much more memory for implementation than partitioning the data, in addition to the time required for the merging operation. Shared memory multiprocessors provide a uniform view of the processors' data space, with each memory location being accessible from any processor. In the case of shared memory multiprocessors, the memory latency for data on a non-local memory module is significantly higher than that for a reference to the local memory module. Hence, judicious distribution of data among the memory modules can have a significant impact on realized performance.

2.1.6. Scalability

One issue that has not been dealt with in the past is the ability of the algorithm to provide good speedup on large processor configurations. Some algorithms in the past have been designed with a set multiprocessor configuration in mind, and optimization is limited to

this particular size. Due to the rapidly decreasing cost of microprocessors, very large parallel processors will be available in the future. Already, Ncube has a 4096 processor machine and BBN's TC2000 is capable of supporting up to 512 processors using a shared memory paradigm. While these algorithms cannot be tested on such a large machine at the present time, they can be evaluated for their potential performance on massively parallel architectures.

2.2. Taxonomy of Parallel Graphics Decompositions

In this section, a taxonomy is presented of parallel approaches which can be used to partition a parallel graphics rendering algorithm. The usefulness of each of these approaches for MIMD machines is analyzed in an effort to narrow down the choice of algorithms. The criteria for implementation is based on the issues raised in the previous section. The taxonomy includes possible new decompositions that have not yet been developed as well as results obtained by previous researchers. Figure 2.2 illustrates the overall structure of the taxonomy.

In the subsections that follow, different parallel approaches to graphics rendering are reviewed within the structure of the taxonomy. A number of approaches devised in the past were intended as special purpose architectural designs. In some cases, these algorithms could also be used for a multiprocessor and they are discussed here, noting that the original design was for a hardware implementation. Although it would be preferable to include all work that has been done in this subject area, only representative examples of each of the categories in the taxonomy are presented. Other related work is quoted and references are given to provide as complete a listing as possible.

A large contingent of ray tracing algorithms has been developed for parallel implementation. Since this book focuses primarily on fast graphics rendering algorithms, and ray tracing is typically an order of magnitude slower than a conventional tiling algorithm, this approach was analyzed in the tests described here. Some ray tracing designs are still worthy of note due to their unique methods of task partitioning or memory usage, so a selection of these are described in the taxonomy. A paper which provides a good synopsis of parallel approaches to a variety of graphics algorithms is [Burk90].

In the following subsections, brief descriptions of the various algorithms which fit into the various categories of the taxonomy are given.

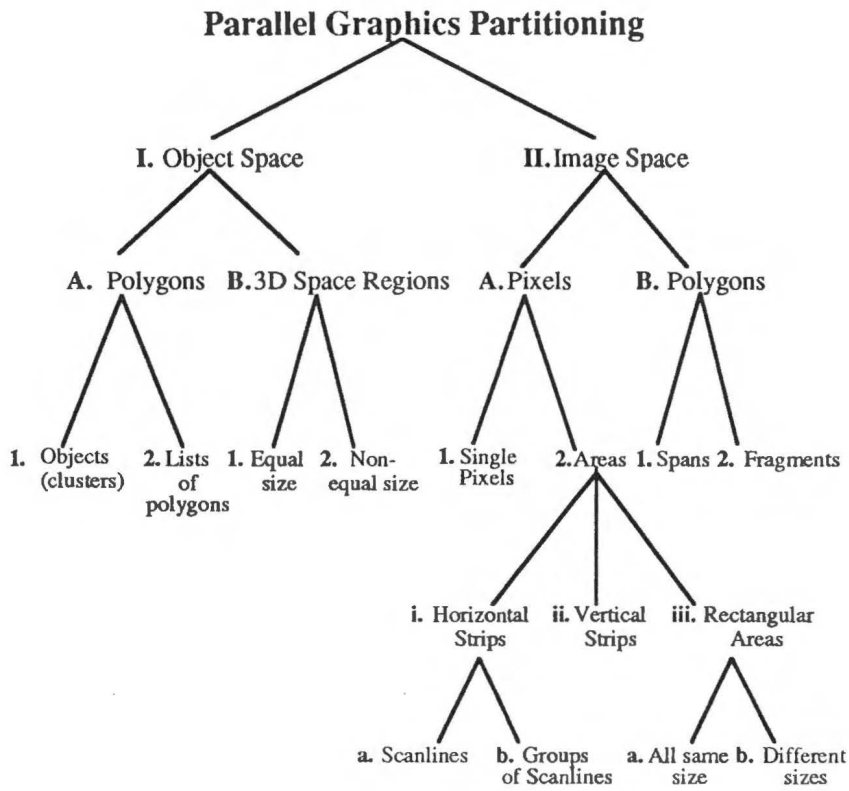


Figure 2.2: Taxonomy of approaches to parallel graphics partitioning

2.2.1. Object Space

Parallel object space decompositions are rare because there has been very little development of object space graphics rendering algorithms. The principal advantage of an object space algorithm is that the hidden surface removal calculation can be computed at arbitrary accuracy. In general, though, the computations in an object space algorithm are inefficient and are more difficult to program in comparison to image space methods. Nevertheless, some researchers have chosen to go this route for a parallel rendering algorithm; these are described next.

2.2.1.1. Polygons

Partitioning tasks based on polygons can be accomplished in a number of different ways: clusters of objects or sub-objects and lists of polygons.

Abram

Abram [Abra86] used Weiler and Atherton's hidden surface removal algorithm, but instead of using their concave polygon clipper, he implemented a fairly simple convex polygon clipper such as the one described by Sutherland [Suth75]. More clipping operations were required than in the concave clipping approach, but the code was simple to implement and did not contain unruly pathological cases such as are present in the Weiler-Atherton clipper. The rest of the algorithm is basically the same as the serial Weiler-Atherton approach, with extensions to facilitate a parallel approach designed for a hardware implementation. As the clipping procedure recursively builds inside and outside lists of polygons based on a clip polygon, a tree structure of lists is created. The tree depends strongly on the input data, but it is built up rather quickly. In Abram's design, the tree is laid out onto a linear pipeline architecture with nodes of the tree mapped to processors in a pipeline. This section of the algorithm only solves the hidden surface removal problem, however. Abram suggests that the tiling problem can then be solved by attaching tiler processors which take the input of visible polygon fragments from the pipeline section and perform the actual illumination and scan conversion of pixels which are then output to a frame buffer. Although Abram's algorithm is specifically tailored as a hardware design, it could easily be mapped to a commercial multiprocessor.

Kankanhalli and Franklin

In a recent paper, Kankanhalli and Franklin [Fran90] present a completely different approach to object space parallelism that deals

not only with lists of polygons, but also with edge lists and areas on the screen called *cells*. The algorithm is basically a parallel version of Franklin's [Fran80] object space hidden surface removal algorithm. The algorithm involves constructing a grid which is overlaid on the scene and then determining the covering faces within the grid cells. There are numerous stages of the algorithm, and each stage is a setup to the next stage. Synchronization is required after each stage of the algorithm which can degrade overall performance. This algorithm was implemented on a Sequent Balance with 15 processors, and the hidden surface removal performance was analyzed for two small images. The authors note that the speedup is different for the hidden surface removal section than it is for the visible region reconstruction section. Next, a brief summary and analysis of each of the object space algorithms is presented.

Summary

In the case of Abram's and Kankanhalli's algorithms, the added complexity of the hidden surface removal sections presents a more difficult programming task, in addition to the fact that efficiency in these approaches is not that high. In fact, Kankanhalli calculates a speedup factor of 10 for just the hidden surface removal in his algorithm utilizing 15 processors, resulting in an efficiency of only 0.67. Speedup is a measure of parallel algorithm performance in comparing the time on 1 processor versus the time on P processors (in this case, $P = 15$). Efficiency is speedup divided by P. More detail is presented on these measurements in chapter 4. The speedup for the visible region reconstruction portion of the algorithm is only 6 on 15 processors, which gives an efficiency of only 0.4. Since the total performance of the algorithm is bottlenecked by its slowest part, in addition to the synchronization required between sections, this algorithm does not provide performance which is adequate enough for high performance on large processor configurations.

The tiling section is a separate add-on task to both of these algorithms. Tiling dominates the total display calculation time these days, especially when Phong shading and anti-aliasing are added to the rendering phase. Neither of these researchers has developed an adequate method of solving the tiling problem in parallel because the focus of their work was restricted to the hidden surface removal section. Franklin and Kankanhalli's algorithm is based on functional parallelism in addition to data parallelism. The sections of the algorithm are divided into segments, each of which is applied in parallel to the data. Unfortunately, the synchronization required

after each segment limits the potential speedup due to the load imbalance incurred at each synchronization point.

2.2.1.2. 3D Space Regions

Regions of three-dimensional space can be partitioned and assigned as tasks. This method has primarily been used in parallelizing ray tracing, and although none of the methods described here serves as a basis for further analysis, an illustration of the algorithms serves to provide an insight into a unique method for partitioning. Ray tracing may be referred to as an image space algorithm since the hidden surface removal is based on a ray shot through a pixel on the screen; however, the actual intersection and illumination calculations are performed in object space. In this instance, the parallelism is devised from a division of the object space.

Cleary *et al.*

Cleary [Clea83] developed a ray tracing algorithm which involves assigning regions of 3D space to each processor. A processor handles rays as they traverse into its region, and then sends the results in ray packets out to the appropriate neighboring processors as they leave the region. Load balancing is not handled directly; rather, it is assumed that the rays traverse through the different parts of the scene in a random manner such that the processors each have approximately the same amount of work. This assumption is not very accurate and hence can lead to poor performance, especially for large processor configurations. A better approach which provides more direct load balancing for ray tracing is given next.

Badouel *et al.*

Badouel [Bado90] presents three approaches to parallel ray tracing, one of which is called the "ray dataflow" approach and is similar to Cleary's algorithm. The others are described in section 2.2.2 on image space partitioning. Badouel attempts to load balance 3D regions by clustering together equal size smaller regions depending on their expected time complexity. The regions are clustered together so that the clusters themselves have approximately the same time complexity as each other. The initial time complexity of a region is found by shooting a small group of rays within each small region and recording the calculation time of these rays. The clusters are mapped onto processors statically, and rays are passed through the system as in Cleary's algorithm.

The advantage of both of these algorithms is that the database is distributed statically and does not need to be replicated in each pro-

cessor. Although Badouel's algorithm exhibits better load balancing characteristics than Cleary's approach, this static method of load balancing is not adequate enough for good overall performance.

Caspary and Scherson

Caspary and Scherson [Casp89] developed a ray tracer which is also similar to Cleary's approach for use on a hypercube multiprocessor. A portion of the database is duplicated in each processor, while the bulk of the data is scattered among the processors' memory. By using two processes per processor, load balancing of the work is facilitated. One process handles intersections with the hierarchical database at a high level, while the other one performs intersections between rays and the actual bounding volumes and objects within the local processor. This method handles load balancing, in addition to dealing with memory management effectively.

Challinger

Challinger [Chal91] developed several approaches to parallel volume rendering. The first approach is a parallel extension to object space rendering using the well known *projection* method. The second method is described under the image space processor-per-pixel heading. The parallel implementation of the projection method is an order dependent approach based on which view of the volume cube is seen from the observer's point of view. A visibility graph is constructed which allows one to move voxels into the ready list for parallel rendering. The cells in the ready list can be processed in parallel, but the visibility graph must be updated afterward. This method constitutes a large amount of overhead, but is a unique look into a rendering technique that is quite new.

2.2.1.3. Analysis of Object Space Methods

Object space methods are typically inefficient when compared to image space algorithms. This is especially true of the ray tracing solutions, which is the reason these are not implemented. If the accuracy of the non-ray tracing object space methods is needed for a particular reason (such as to allow changing of the illumination after the hidden surface calculation), then these methods may be worthy of implementation. This is not a concern for most everyday applications, however.

2.2.2. Image Space

Parallel image space partitioning methods are much more prevalent in the literature than the object space methods. They are more

suitable for hardware implementation, and there are many adaptations of this type of algorithm. The image space algorithms can be divided into two subsets: those based on pixels or groups of pixels, and those based on polygons or polygon fragments as noted in the taxonomy. An important point to note here is that most of the previous work in this area specifies only how the image is divided up, not how the underlying algorithm is implemented nor how the memory referencing technique is employed. In addition, the algorithms mentioned were simulated rather than implemented on a multiprocessor. The primary reason for this is that very few researchers have had access to such machines until recently. The methods presented here at best only indicate their expected performance since the results have only been theoretically analyzed. The only actual implementations on commercial multiprocessors presented in the literature are those by Theoharis and Roble.

2.2.2.1. Pixels

Parallel display algorithms which are based on pixels are the most popular type of image space decomposition. The principal reason for this is that the pixel calculations are completely independent of one another, so no synchronization is required and the order of task execution is irrelevant. Algorithms which assign a single pixel as a task are typically designed for hardware implementation. This task size is too fine a granularity for implementation on a general purpose MIMD machine since context switching would severely degrade performance. Several of the parallel approaches which use this level of granularity are described next. Another type of pixel decomposition involves tasks which represent areas of adjacent pixels grouped together in one way or another. These methods are described immediately following the discussion of processor-per-pixel decomposition designs.

Processor-per-Pixel Designs

Fuchs *et al.*

Fuchs' [Fuch85] Pixel-Planes 4 system is a good example of a processor-per-pixel hardware architecture. Each pixel contains a small one-bit ALU in addition to a binary tree of one-bit adders designed to efficiently compute the equation $F(x,y) = Ax + By + C$. This equation is used to test for polygon containment as well as calculation of visibility and illumination. Polygons are sent to all processors, and each pixel processor then determines if the polygon covers its area. If the polygon covers a given processor's pixel,

visibility and shading calculations are performed. The system is somewhat inefficient since each processor must check every polygon in the dataset. Fuchs' recent extension to this system called Pixel Planes-5 alleviates some of the inefficiencies in the first system and is described in [Fuch89].

Whitman and Dyer

Whitman and Dyer [Dyer87] developed a vectorized version of a scan line Z-buffer algorithm. This program was designed for an SIMD vector architecture and featured pipelined pixel processing for the shading and visibility calculations. Although the algorithm is too fine-grained for an MIMD architecture, it could serve as a basis for an algorithm which would be suitable for a multiple processor SIMD architecture.

Plunkett and Bailey

Plunkett and Bailey [Plun85] developed a vectorized version of a ray tracing algorithm that processes rays independently. This algorithm is also designed to run on an SIMD pipeline architecture. Rays are placed into a queue, and when the queue fills up, all of the rays are intersected in pipeline fashion. Any new rays generated are attached to the end of the queue for future processing.

Challinger

Challinger [Chal91] has designed a parallel volume rendering approach based on ray tracing. The results seemed to indicate that assigning a pixel per task used significant overhead, while assigning a scan line per processor (as in the processor-per-area approach elaborated upon next), achieved better performance.

Processor-per-Area Designs

Parallel algorithms which work on groups of adjacent pixels represent the widest variety of partitioning methods that have been researched. The different categories in which these algorithms fall include: horizontal strips, vertical strips, and rectangular areas of pixels. Algorithms which are based on horizontal strips can be divided into two sub-categories: those based on single scan lines and those based on contiguous groups of scan lines as tasks. These groups of scan lines as tasks are referred to as *blocks*.

Kaplan and Greenberg

Kaplan and Greenberg [Kapl79] simulated two different hidden surface algorithms and analyzed them according to their usefulness on a parallel architecture. A Watkins' [Roge85] scan line algorithm is subdivided into P groups of s scan lines, where each group forms a

different task for a processor. Their design relies on a central control scheduling mechanism, whereby a task is assigned to a processor as it becomes free. The number of groups or the number of regions can be much larger than the actual number of processors available, allowing dynamic load balancing. Shared memory is not a consideration in their simulation; each processor is assumed to have in its local memory all of the information it needs to perform calculations for its portion of the scene.

Another parallel algorithm due to Kaplan and Greenberg is an adaptation of Warnock's [Roge85] algorithm. A static area mesh to the image space and each task is assigned to one region of the mesh. The Warnock algorithm is executed serially within each region. The mesh is applied at both low (16 x 16) and high resolution (32 x 32) to discern the differences in speed. As might be predictable, the finer grain mesh resulted in a more uniform time/area than the coarser mesh. Both the Watkins' and Warnock decompositions are illustrated in figure 2.3.

The authors suggest three considerations which should be taken into account when deriving a parallel implementation of a hidden surface algorithm: *partitionability*, the method of dividing the computation among independent tasks such that communication is kept to a minimum; *coherence*, the reduction of visible surface calculations by basing them on previously obtained results; and *computational efficiency*, the ability of the parallel processor system to schedule tasks. In addition, the authors believe that characteristics such as image area, image complexity, edge complexity, and how the

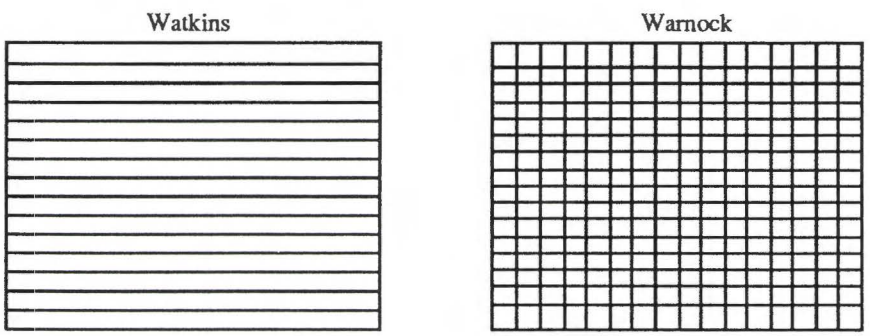


Figure 2.3: Two types of decompositions applied by Kaplan and Greenberg

image relates to the algorithm all affect the resultant performance of the algorithms. They also suggest that utilizing a good heuristic task scheduling algorithm is very important in obtaining good load balancing and high performance in the system.

In the Kaplan-Greenberg simulations, a static decomposition approach is applied in dividing up tasks which are then assigned dynamically to processors by the scheduler. Coherence is maintained in a region in their first method (Watkins approach) within scan lines and pixels. In their second method (Warnock approach), area coherence is used within a region of image space. Utilization of the processors is good, especially when the number of regions subdivided is small enough to allow a large number of tasks to be dynamically assigned. Load balancing is also good only when the regions are small enough due to the dynamic task assignment method which is used. As long as the number of regions created is not too large[1], the granularity level of these algorithms is suitable for implementation on a general purpose MIMD machine. The algorithms have good scalability if the number of regions created is adaptable to different processor configurations. The authors state that memory access is local since each processor will contain the data it requires. No clue is given as to how this might be accomplished, though.

Kaplan and Greenberg's algorithms are one of the first efforts in the area of parallel algorithm design for graphics. Their simulations are designed mostly to analyze the difference between two different parallel approaches, not to extrapolate to real world performance. Still, their idea of creating more tasks to achieve better load balancing seems natural. It seems reasonable then to further evaluate their ideas, especially with regard to memory referencing. In any case, this rectangular approach is further investigated in tests described in chapter 5. It is not clear from their paper how many rectangular regions are optimal, nor what type of memory partitioning algorithm should be used. Therefore, the descriptions in chapter 5 include an analysis of methods which can be used to determine these factors.

Chang and Jain

Chang and Jain [Chan81] have simulated a distributed multiprocessor version of Watkins' scan line algorithm. Their idea is to distribute the data among the processors in three-space with either horizontal cross-sections cutting the screen into P horizontal regions or a division of the scene into P cubic regions. This decomposition is shown in figure 2.4. The first method is essentially the same type of

[1]This was not quantified by the authors.

decomposition as Kaplan and Greenberg's technique, while the second method divides the areas in a more rectangular fashion. Chang and Jain's algorithm is somewhat different than Kaplan and Greenberg's approach, though, because the polygons are actually clipped in three-space within a single parallel task. Although this might seem to be a 3D decomposition, it is essentially similar to a 2D partition in which the perspective and clipping operations are performed in parallel.

In either of Chang and Jain's decomposition methods, each processor is responsible only for the polygons in its region, allowing parallel data processing. It is not clear from their paper, but one can infer that each processor gets a copy of the entire dataset. This is inefficient since redundant work (in addition to the extra space required) is necessary for the perspective and clipping calculations. Coherence is lost between adjacent regions, and each region has to perform additional three-dimensional clipping, which, as the authors observed, can override the hidden surface calculations if the regions become small enough. The paper considers only a limited number of polygons, and therefore their results cannot be applied to today's imagery. The authors state that due to the independent processing of polygons, each processor must initialize the scan conversion process for its region since there is no coherence between regions. In addition, if polygons are not uniformly distributed among the processors, the resultant time is degraded by the slowest processor. As a possible solution, the authors suggest breaking down the screen according to dataset density so each processor is able to finish close to the same time as the others. This was subsequently implemented by Whelan in his Median-Cut algorithm described later in this chapter.

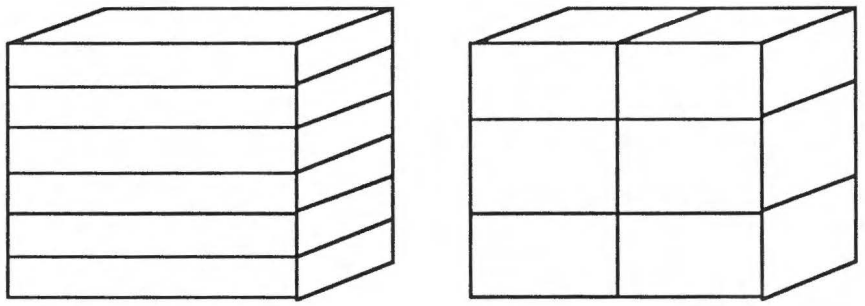


Figure 2.4: Chang & Jain's decomposition method

Chang and Jain's algorithm is a similar decomposition to Kaplan and Greenberg's method, except that clipping is part of each parallel task as well. The only unfortunate aspect of this is that polygons must be initially stored in all regions (or at least available to all processors) and in the parallel processing phase; the polygons may be clipped multiple times. In most cases, a polygon will not be displayed in a processor's region, but a trivial clip must be done anyway to check for this situation.

Hu and Foley

Hu and Foley [Hu85] analyzed one dynamic and two static distribution methods based on block size variations on a scan line. Their analysis determines to some degree the effect of coherence on parallelism. The static distributions analyzed were denoted the *static contiguous method* and the *static interleave method*. The static contiguous method exploits vertical coherence within a single task, while the static interleave does not. *Static contiguous* refers to a partitioning scheme in which the screen is broken down into P horizontal regions, each containing $y\text{-}resolution / P$ scan lines. The static interleave method involves partitioning the scan lines among the processors in such a fashion that each processor i would process all scan lines i, $i + P$, $i + 2*P$, $i + 3*P$, etc., as is illustrated in figure 2.5.

This technique could have been extended to interleave in the horizontal direction as well, but Hu and Foley chose just to deal with scan

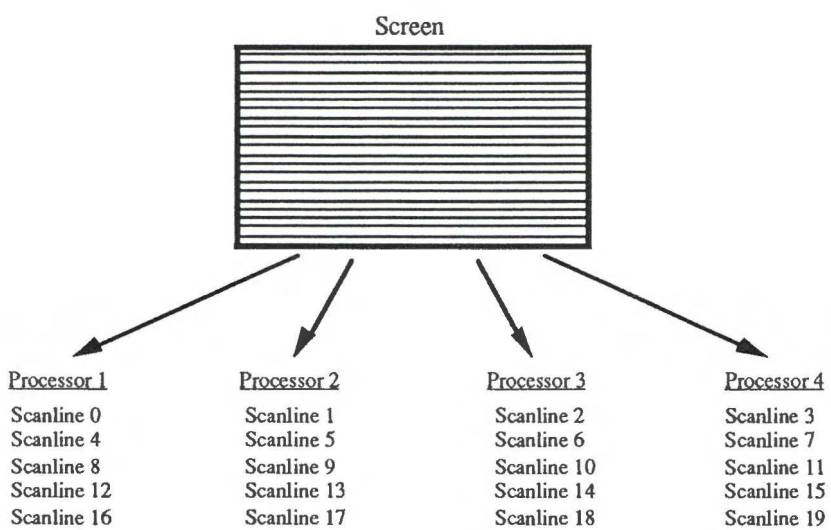


Figure 2.5: Example of scan line to processor assignment in Hu & Foley's interleaving algorithm

lines. The way in which these two static methods attempt to achieve load balancing is different because each tries to minimize different factors. The static contiguous method attempts to capitalize on vertical scan line coherence, a time saving technique used by most sequential algorithms. The downfall of the contiguous method is that it relies on a uniform distribution of the geometric elements in the scene across all blocks of scan lines; an unlikely occurrence. Their interleaving scheme is based on the fact that the geometric elements are not likely to be distributed across scan lines uniformly. Each processor will have nearly equal work since they deal with successive scan lines, but this comes at the expense of vertical scan line coherence. Finally, the *dynamic method* assigns processors to single scan lines in a dynamic scheduling fashion. The dynamic method is similar to Kaplan and Greenberg's idea, except in this case, each task is a single scan line rather than a group of scan lines. The dynamic method follows along the lines of the static interleave approach, except that task to processor assignment is resolved during runtime in the dynamic method, while it is done prior to tiling in the static method.

All static partitioning schemes have one inherent advantage over a dynamic scheme: no scheduling of tasks needs to occur at runtime. In a hardware design which Hu and Foley intended for their algorithm, this can be an important factor. In a software algorithm for a general purpose multiprocessor, this factor is minimized since scheduling must occur for all tasks; therefore, the number of tasks generated is the only overhead. Still, though, the parallel programming method used can have some impact on scheduling overhead. In other words, generating more tasks takes additional time, but this time is small enough to be negligible compared to the running time of a given task (assuming task size is large enough). The main difference between Hu and Foley's dynamic method and their static interleave method is in the task assignment to processors. The dynamic method is implemented (in a simulation on a VAX) at the scan line level by Hu and Foley and obtained the highest performance of the three parallel scan line designs based on their results. Their research involves a simulation of the algorithm on a von Neumann machine since their intention was to build a hardware architecture. Their graphs indicate very good expected performance for the dynamic algorithm when each processor contains the entire dataset. This is not a realistic situation for large databases, so memory storage strategies need to be investigated. If a different memory referencing strategy is implemented, this dynamic technique might provide good speedup and is therefore worth investigating further.

Ghosal and Patnaik

Ghosal and Patnaik present a scan line parallel algorithm that is somewhat similar to Hu and Foley's approach [Ghos86]. They describe several approaches, but their best algorithm is based on processing the scan lines for the y-extent of a single polygon in parallel. Overall parallelism is limited due to the small number of scan lines within the y-extent. In addition, synchronization is necessary after each polygon is finished. Hu and Foley's algorithm seems more general purpose than Ghosal's algorithm, since theirs is not based on the size of the polygon.

Whelan

Whelan [Whel85] compares several different image space task partitioning strategies: a horizontal strip method, a vertical strip method, and a rectangular region method. Whelan's rectangular region method is almost the same as Kaplan and Greenberg's Warnock approach, except that Whelan does not state what serial algorithm is used to tile a single region in his mesh. These methods are simulated to see which exhibits the best overall performance. Although the horizontal and vertical strip schemes might sometimes result in faster times, the rectangular region method is resistant to differences in the imagery and provides the most consistent results. These decomposition methods are illustrated in chapter 5 in figures 5.7, 5.8, and 5.9.

Crockett and Orloff

An algorithm which also uses the horizontal strip method was recently developed by Crockett and Orloff for the Intel iPSC hypercube [Croc91]. This algorithm involves extensive work to take advantage of the message passing architecture of the iPSC/860. Triangles are distributed evenly among the processors, and shading, transforming, and clipping are all performed by the local processor. Each processor is responsible for a region of the frame buffer, so it must receive the triangles from other processors which belong to its area. The processors then take turns passing triangles to the appropriate processor for rasterization, as well as performing the actual rasterization. A conventional Z-buffer is used so that the communication of the triangles can be overlapped with the rasterizing operations. There is a tradeoff between spending time rasterizing triangles and thus not sending out triangles to other processors, and vice versa.

Although the authors present extensive performance analysis for the algorithm and even give a model for the work, they do not focus on the load balancing success of their work decomposition strategy. The

bulk of their work seems to be the method by which the communication is done asynchronously within the same processor as the rasterization. This is the primary value of the authors' work since that is a unique problem on this type of machine. In fact, this research could be extended to generalize the data decomposition scheme for any graphics display algorithm on a message passing architecture. Crockett and Orloff also state that the algorithm can be modified for a shared memory architecture. It is clear though, that the modifications given for the latter case represent such a departure from their original design that it should not even be considered the same algorithm.

Parke

Parke [Park80] uses a technique which is based on the traditional Z-buffer. He distributes portions of the image space to processors arranged in a tree structure. Essentially, a hierarchy of regions is created and divided among the processors, with the complexity reduced as the tree is traversed. The output of a parent splitter is the input of the child, and so on, until the content of a region is sent to a single processor. This is illustrated in figure 2.6. Parke uses a Z-buffer which is partitioned among the processors, with each processor handling a portion of the Z-buffer to avoid contention for common memory. This design was intended to be a special purpose machine; however, a simulation is described in Parke's paper.

Parke also describes Fuchs' approach to the problem, in which a central broadcast controller distributes the input data and the Z-

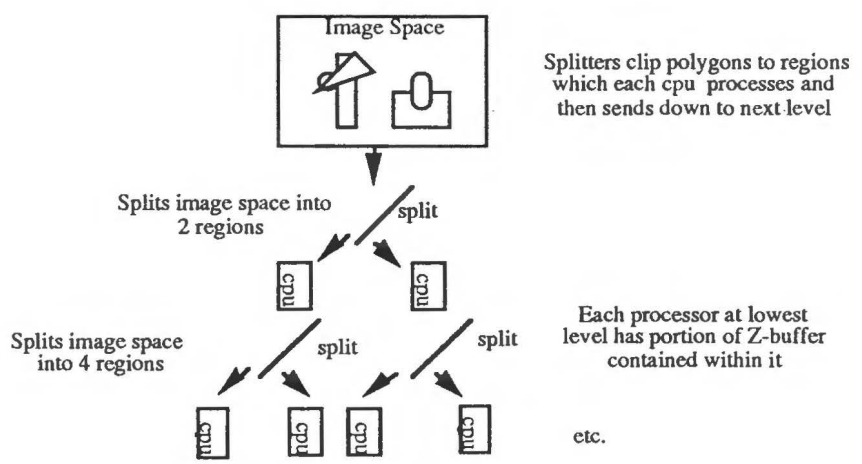


Figure 2.6: Parke's splitter tree of processors

buffer memory is segmented in an interlace fashion rather than a contiguous one as in Parke's original design. A hybrid of the two algorithms is suggested as the best possible alternative since this would alleviate the under-utilization problem.

Parke's initial algorithm is a static decomposition and relies on a uniform distribution of objects in the scene, so each processor will be just as busy as its neighbors. Assuming Parke's hybrid algorithm could be implemented, load balancing and utilization might be optimized. Communication can become a bottleneck in his system due to the passing of polygons from level to level in the tree. The algorithm is a standard Z-buffer algorithm, which means it suffers from the aliasing problem that is inherent in that methodology. It could be extended to be solved with any of the anti-aliasing methods common today, however. The principal limitations of the algorithm are the large amount of communication and the lack of adequate load balancing. This makes Parke's method unsuitable for implementation on a general purpose multiprocessor machine.

Theoharis

One unusual parallel implementation of a hidden surface algorithm is by Theoharis [Theo86] for use on a network of Inmos Transputers. Theoharis' method uses a variation of Parke's splitter mechanism. This algorithm assigns portions of the computer graphics display pipeline to different processors, and passes the information from one processor to the next until a scan conversion processor handles the actual rendering section. Each transputer handles a polygon and performs clipping, hidden surface elimination, and scan conversion in a pipeline format. The transputer has very fast context switching between processes, which makes it ideal to support fast changes as polygons come down the pipe. Clipping is performed via the Sutherland-Hodgman (see [Roge85]) polygon clipping algorithm, with each clipping plane forming a stage of the pipe. Then, multiple scan converters run in parallel, accepting polygons and generating pixel lists of those pixels covered by that polygon. A buffer routine forms the last stage of the pipe, which runs a standard Z-buffer hidden surface removal algorithm for the allocated image partition. Once all pixels have been handled, the frame buffer is displayed. Parke's splitter mechanism is employed to further limit the number of polygons handled. The algorithm is illustrated in figure 2.7. The pipeline does not consist of that many stages, so it needs to be expanded out in a tree fashion (steps 8 & 9 in the figure). The splitter mechanism accomplishes this by creating a tree of processes running

in parallel which can form their own pipes and keep the available processors busy.

Theoharis' scheme has the disadvantage that he assumes as Parke did that the image is uniformly distributed between all of the split planes. If this is not the case, some processors will have less work to do than others. He mentions that this problem can be alleviated by random splitting of non-contiguous areas in order to achieve load balancing, but this has not been investigated. Theoharis' algorithm uses a functional parallel decomposition, and it is possible that the communication between processors might limit the speed of the program. Processors performing the clipping and transformations will almost certainly be faster than the processors performing the scan conversion, leading to a bottleneck in the system. In this case, the load will not be universally balanced among all of the processors. In addition, some processors might be assigned sections of the scene which are far less complex than other sections. Although the algorithm illustrates a novel approach to the problem of a parallel hidden surface method, the solution given may not be able to be applied to a general purpose multiprocessor which does not have the communication properties of the Transputer. The real limitations in the algorithm are the assumed uniformity in the image and the large amount of communication, which are the same problems from which Parke's algorithm suffers. These algorithms might be suitable for hardware implementation, but are not appropriate for use on a conventional multiprocessor.

Whitman

The author of this book previously developed a parallel version of an area coherence scheme similar to Warnock's method [Warn69]; it

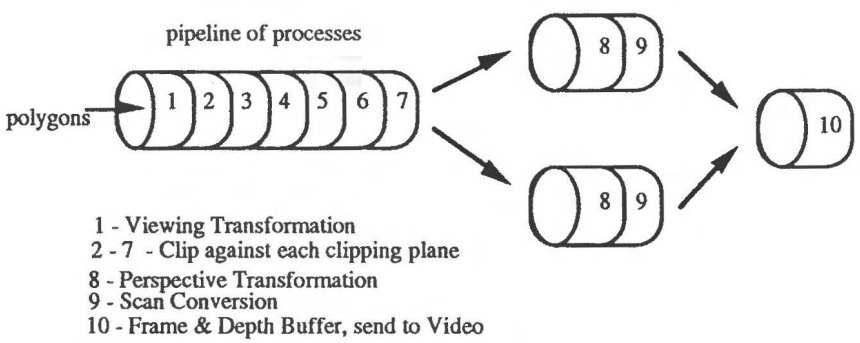


Figure 2.7: Theoharis' pipeline of processes for image decomposition

is illustrated in figure 2.8. This methodology employs a dynamic decomposition whereby a region is subdivided if it is too complex to compute. Instead of being recursive as in Warnock's original design, the algorithm assigns the subdivisions to separate processors, and the same tests are performed again within these subdivisions. If the region is too complex again, more subdivisions are created. Processes are assigned to subdivisions, and as a processor becomes free, it is assigned to a region. Coherence is maintained via the area method, which can be taken down to the pixel level if necessary.

There are several problems with this implementation of the Warnock algorithm in parallel. This algorithm is excellent for hidden line removal, but if it is used for hidden surface removal, the algorithm is not well suited for tiling polygons. One method would involve tiling each region at the point of hidden surface removal, but this creates a huge number of tasks. In addition, edge lists and other data structures need to be built for each small region, involving a lot of overhead. If the approach suggested in Warnock's paper is used, the tiling would be a separate operation. Synchronization needs to occur prior to tiling, and then a visible region reconstruction algorithm similar to that of Kankanhalli needs to be performed. This extra synchronization degrades performance, in addition to the fact that another entirely separate technique is required for tiling the visible regions in parallel.

The granularity of tasks created using the Warnock method is too fine for a general purpose parallel machine. The high context switching creates too much overhead in this approach since there are so many tasks created and the execution time of each task is very

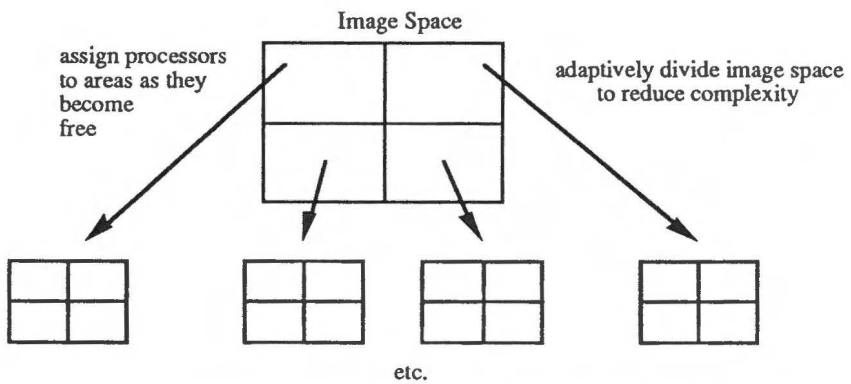


Figure 2.8: Whitman's parallel variation on the Warnock subdivision algorithm

small. As a result, the performance of the algorithm in parallel is not very good. Secondly, while the Warnock algorithm is adequate for hidden line removal, it is fairly slow compared to other image space algorithms for hidden surface removal. For very large polygons it might provide reasonable results, but the datasets which are typical in today's imagery are large, meaning that the average polygon size is smaller than when Warnock developed his algorithm. These two factors indicate that this algorithm is not a good choice for implementation.

Painter's Algorithm

The painter's algorithm due to Newell, Newell, and Sancha [Roge85] might make an interesting candidate for a parallel algorithm. The problem with converting the painter's algorithm to a parallel environment is the requirement for a specified order of tiling the polygons. This might be alleviated if regions could be specified as tasks and the painter's algorithm could work as a serial approach within each task. In fact, any hidden surface algorithm could be implemented as a serial task within any of the area based approaches because they do not rely on a functional decomposition. This is because of the independent nature of these tasks.

A generalized implementation of the Newell, Newell, and Sancha algorithm in parallel has not been presented in the literature, and it is easy to see why. The synchronization necessary to make sure that pixels are not overwritten in incorrect order will limit the potential speedup of the algorithm.

Adaptive Algorithms

While the methods described to this point all involve decompositions without regard to the data input set, several approaches have been developed that attempt to take into account the input scene when partitioning the work. These schemes are outlined next and take into account the work in a given area of the screen to estimate a priori how to divide the work among the processors. Whelan uses the centroids of polygons as their locations, and attempts to assign an equal number of polygons to each processor. Roble similarly tries to assign an equal number of polygons per processor, but he uses a bounding box and the regions are determined differently.

Whelan

Whelan [Whel85] is one of the first researchers to suggest a scheme based on non-equal size areas. This method (which is distinct from his other approaches) is called the *Median Cut* algorithm and proceeds as follows. The idea behind the algorithm is the creation of a

median line across a given region, in which half of the polygons are in one sub-region and half in the other sub-region. To achieve this, the image space is divided recursively, based on the centroids of the data elements. At each recursion level, the median of the centroids of all polygons in the region is used as a dividing line, alternately in the horizontal and vertical directions. This process of subdividing is repeated until the number of subdivisions equals the number of processors. Although it is not an optimal partitioning scheme, it can produce very favorable results on a variety of data input sets. The unfortunate drawback is that determining the location of the partitions involves sorting the centroids many times, and this overhead is hard to overcome in the performance of the rest of the algorithm.

Whelan's results indicate that his Median Cut algorithm has the potential for high performance, but it exhibits a significant amount of start-up overhead. Since this approach was not deemed viable by Whelan, his rectangular area approach, which is a generalization of Kaplan and Greenberg's parallel Warnock method, holds the most promise. This latter method could be adapted to any number of processors and still have a minimum overhead.

Roble

Roble [Robl88] has developed a scan line Z-buffer algorithm which is designed to exploit load balancing prior to the tiling stage. It is similar to Kaplan and Greenberg's area approach and was implemented on the Intel iPSC hypercube. Roble's idea involves counting the number of polygons sent to each processor under a given partition. If there is a strong discrepancy between the processors as to the number of polygons handled, the cube manager re-partitions the scene again so a nearly uniform distribution is achieved. This is a fairly dynamic solution since the tasks are updated during runtime. It is essentially the same as Badouel's clustering technique described in section 2.2.1.2, except that no prior work is required since the number of polygons is used as a heuristic to indicate the amount of work in a region. The decomposition is based on the input polygons, and load balancing is partially solved with this method. Memory contention is not an issue since once tasks are divided, each processor independently solves the hidden surface problem. This is a good solution for a multiprocessor with a small (< 50 or so) number of processors, but as the number of processors is scaled up, the region size is smaller and there will be more overhead.

Roble divides the screen space into P equal sections and passes polygons to the processors for each section. If the number of polygons

in certain sections creates a situation where some nodes have more work to do than others, the sections are merged and divided in an attempt to create an equal computational load for each processor. This type of approach is just a variation on the rectangular region decomposition theme, except that the region sizes are different depending on the amount of work present. Roble had some success with this approach, and Whelan showed his Median Cut algorithm to provide the best overall solution among his comparisons. Both authors state that the overhead can be quite costly and can override any performance gains. It seems worthwhile that if the overhead can be limited, then this type of algorithm will provide good performance in an implementation.

Analysis of Algorithms for Pixel Decomposition

It seems clear that the processor-per-pixel architectures and algorithms involve a very fine grain solution which is not applicable to implementation on this type of machine. The primary reason is that the task size is too small and context switching would dominate the computation. On the other hand, the processor-per-area designs are better suited to implementation on a general purpose multiprocessor. This is because the task size in these designs is large enough to eliminate context switching problems, yet it can be varied to handle load balancing in a variety of ways.

Other algorithms seek to distribute data to processors in a static manner so that no further communication takes place between the processors. After the graphics space is divided up, the hidden surface removal and rendering calculations are performed within a single processor for each section. Chang and Jain use this approach by statically dividing up three-space and assigning P processors to P regions. The disadvantage of this approach is that good load balancing is not achieved since uniformity of the image is not a realistic scenario. Whelan and Roble attempt to directly solve this by using a static decomposition which determines to some degree the amount of work assigned to each processor.

2.2.2.2. Polygons

Z-buffer

One of the more interesting sequential graphics display algorithms is the Z-buffer due to Catmull [Catm74]. This algorithm could be modified to process individual polygons as tasks. A parallel version of the Z-buffer might work as follows. A full screen Z-buffer memory is stored in globally shared memory and scattered

throughout the system. Each processor scan-converts a single polygon as a task and writes the pixel value into the scattered frame buffer if the value of the Z-buffer is greater than the z-value of the polygon at that pixel. To handle anti-aliasing, a very large Z-buffer and frame buffer could be used and post-filtered down to the desired output resolution. Although Parke and Theoharis ultimately use a Z-buffer, their decompositions are screen space subdivisions, although Theoharis' has features of both. This method involves parallelism by polygon with a shared Z-buffer.

The parallel Z-buffer algorithm suffers from the problem of contention for a shared resource. This method would require constant referencing of the Z-buffer array, and collisions in remote memory access would likely occur, slowing down the algorithm tremendously. This solution might be adequate for small processor configurations, but would not be suitable for a large MIMD machine.

Allison

A slightly different version of a Z-buffer algorithm which has been implemented on the BBN Butterfly TC2000 is due to Allison [Alli91]. His algorithm involves a parallel decomposition in which each object is sent to a different processor for scan-conversion. The limitation of this approach is that the algorithm is limited in its parallelism by the number of objects in the scene. For scientific datasets, this may not be that bad since most scientific programs use hundreds if not thousands of objects. Another problem, as noted above, is the contention for the shared Z-buffer. Synchronization is accomplished by a lock on each pixel. Objects which cover a large portion of the screen tend to slow the algorithm down, presumably because of blocking of pixel access for other objects. This algorithm is an initial stab at using the TC2000 for parallel processing of graphics rendering. The only problem is that the success of the algorithm depends to a large extent on the composition of the scene.

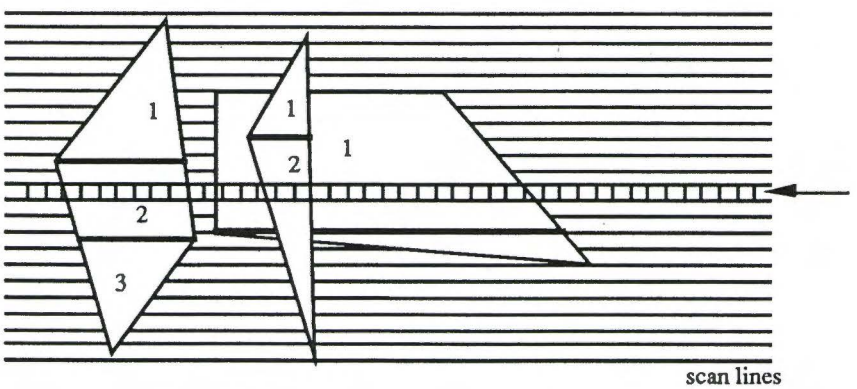
Fiume, Fournier, and Rudolph

Fiume, Fournier, and Rudolph [Fium83] simulated a version of a spanning scan line algorithm for an ultracomputer, which would be similar in design to the NYU Ultracomputer. The processors use the Fetch and Add instruction (called RepAdd in the paper) to assure atomic access to write operations in shared memory. They also propose an addition, the RepMin operation, which would write and replace the element in shared memory if the new one is less than it. This could be used to perform the hidden surface elimination. The polygons are broken down into span-areas which are related to vertices rather than scan lines. Each span-area is a trapezoidal or

triangular region, and each processing element (*PE*) processes the different areas in parallel. All PEs synchronize at the end of processing for each scan line. The authors claim that this is not necessary, and if sufficient memory is available, the technique could be generalized to k scan lines, $k \geq 1$. An example distribution of PEs is illustrated in figure 2.9.

One problem mentioned in the paper is that all PEs could be waiting for a single PE to finish calculation on a long span. The authors suggest subdividing a span if it is larger than some M maximum number of pixels. In order to incorporate anti-aliasing into the algorithm, a coverage mask (8 x 8) is used for the span-area covering a pixel. The weight (mask) of a particular span-area is the fraction of the area of the pixel covered corresponding to the number of one bits assigned to the span-area. Anti-aliasing is calculated after a PE has computed the hidden surface calculations for its pixels on the scan line. The authors' goals were to achieve performance which was better than a sequential algorithm, as well as a parallel method for computing anti-aliasing.

The Fiume *et al.* algorithm suffers from a few limitations. First, the fact that the processors need to synchronize at the end of a scan line forces the algorithm to slow down to the speed of the slowest processor, and this is done at every scan line. It is not clear whether one could take advantage of multiple scan line parallel processing, as



For marked scan line, a PE handles span-area 2 in first polygon, a PE handles span-area 1 in second polygon, and a PE handles span-area 2 in overlapping third polygon. Anti-aliasing and display is synchronized at end of scan line.

Figure 2.9: Trapezoidal span areas each processed by a separate PE per scan line

suggested by the authors in their paper, to solve this problem. The reason is that it may be difficult to synchronize processing of the same span-areas from scan line to scan line. The decomposition is a dynamic approach, but is limited by the synchronization problem. Scan line as well as pixel coherence is exploited. Since the ultracomputer architecture seems to have fast context switching, the utilization of each processor is very good.

Load balancing is a difficult issue, since some processors may be busy with large spans while others are processing short spans, even with span subdivision. Scalability can be solved, but only if there are more spans on the scan line to accommodate the additional processors. The algorithm was not implemented on an actual multiprocessor, so one cannot tell whether a large number of processors would produce a good speedup. It seems that since the parallelism is assigned to PEs by span areas (S) and if $S < P$, some processors will go underutilized. This seems to be one of the major limitations of the algorithm. Memory referencing is not as important an issue, especially with the fetch and add instruction. The constructs are somewhat different than in other multiprocessors, but are not hard to program.

2.2.3. Summary

Based on the algorithms analyzed here, it seems logical that the choice of an image space parallel algorithm based on rectangular areas of pixels holds the most promise for high performance. Note that the work decomposition strategy only amounts to a small portion of the total parallel algorithm. The setup overhead prior to rendering, in addition to the memory referencing strategy, represents an additional issue that affect the overall performance. The implementations analyzed in the remainder of this book are given next.

1. A scan line algorithm similar to the one introduced by Hu and Foley. The dynamic assignment method shown by Hu and Foley indicates a potential for good performance even with the loss of vertical coherence. Since this algorithm was simulated and not implemented, a multiprocessor implementation of this algorithm is necessary to refute or substantiate their claims. In addition, the database storage issue was not fully addressed in their research.

2. A rectangular region algorithm such as the one suggested by Kaplan and Greenberg as well as the one by Whelan. This

method seems to be the most logical choice for parallel implementation since the granularity of tasks can be varied. Although Kaplan and Greenberg determined that a finer granularity yielded better performance, they did not analyze this to any degree. Neither of these research efforts fully addressed the memory storage and access issue, and their results are based on simulations rather than real-world implementations, so a full analysis is necessary.

3. An algorithm which uses a type of task assignment similar to the static approach suggested by Whelan's Median Cut algorithm. Whelan showed that this method resulted in the best performance of all of his simulated algorithms. The problem was that the overhead necessary to determine the task decomposition prior to tiling degraded the overall performance. A simpler type of static approach which is not as accurate as the Median Cut algorithm might perform nearly as well, but without the substantial overhead.

4. A task decomposition scheme which is similar to the rectangular region algorithm involving task sizes determined at runtime. This is based on an idea by Rao and Kumar [Rao89] in which tasks are dynamically split during parallel execution. This approach to load balancing would seem to be a good extension to the rectangular region method.

While some algorithms listed here have been developed in the past, they have not been thoroughly analyzed in terms of task partitioning and memory referencing schemes. In particular, previous efforts by other researchers have not addressed a number of issues relating to computer graphics and parallel processing. These previous efforts have largely been attempts at obtaining parallel graphics display solutions without a thorough analysis of the problem. Some of the issues not addressed in full include:

- Mapping of algorithm to intended architecture
- Size and distribution of tasks
- Memory distribution and communication
- Coherence and parallelism
- Load balancing

The preceding items are fully analyzed in the designs presented in chapter 5. Since the algorithms are implemented on a particular