BLDSC no:- DX182016

LOUGHBOROUGH
UNIVERSITY OF TECHNOLOGY
LIBRARY

AUTHOR/FILING TITLE
HIGHFIELD, J.C.

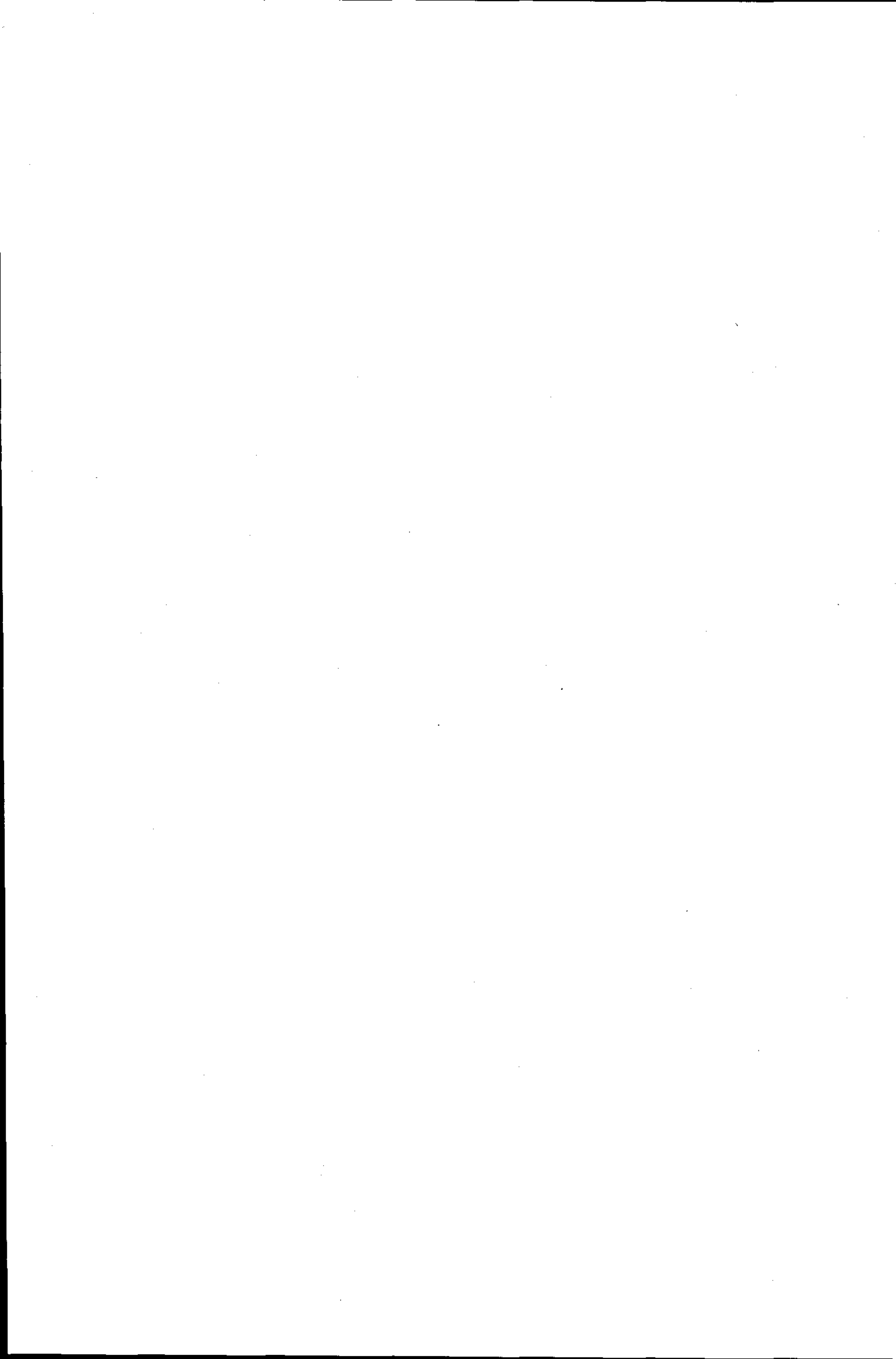
ACCESSION/COPY NO.
040091018

VOL. NO.
CLASS MARK

LOAN COPY

28 JUN 1996
22 MAR 1996
15 NOV [illegible]
28 JUN 1996
25 JUN 1999

BADMINTON PRESS
18 THE HALFCROFT
SYSTON
LEICESTER. LE7 8LD
ENGLAND



Polygon Based Hidden Surface Elimination Algorithms : Serial and Parallel

by

Julian Charles Highfield

A Doctoral Thesis
Submitted in partial fulfilment of the requirements for the award of
Doctor of Philosophy of the Loughborough University of Technology

March 1994

© by Julian Charles Highfield, 1994.

Loughborough University
of Technology Library
Date June 94
Class
Acc. No. 040091018
V8909721

Certificate of Originality

This is to certify that I am responsible for the work submitted in this thesis, that the original work is my own except as specified in acknowledgements or in footnotes, and that neither the thesis nor the original work contained therein has been submitted to this or any other institution for a higher degree.

JULIAN CHARLES HIGHFIELD

To my parents

Acknowledgements

I would like to thank my Supervisor, Dr. Helmut Bez, for his support and encouragement during my research. I would also like to thank my Director of Research, Professor E. A. Edmonds, for his advice. I am grateful to the SERC for their financial support and for use of the Edinburgh Concurrent Supercomputer. Finally, I would like to thank my parents for their support during this period.

Part of the work in this thesis has previously been published as:

J. C. Highfield, "Parallel Scan Line Algorithms for Hidden Surface Elimination", *Occam and the Transputer - Current Developments*, (Proceedings of the 14th World Occam and Transputer User Group Technical Meeting, 16th-18th September, 1991, Loughborough), pp. 217-224, IOS Press (1991).

J. C. Highfield and H. E. Bez, "Hidden Surface Elimination on Parallel Processors", *Computer Graphics Forum* **11**(5), North-Holland, pp. 293-307 (1992).

Abstract

Chapter 1 introduces the need for rapid solutions of hidden surface elimination (HSE) problems in the interactive display of objects and scenes, as used in many application areas such as flight and driving simulators and CAD systems. It reviews the existing approaches to high-performance computer graphics and to parallel computing. It then introduces the central tenet of this thesis - that general purpose parallel computers may be usefully applied to the solution of HSE problems. Finally it introduces a set of metrics for describing sets of scene data, and applies them to the test scenes used in this thesis.

Chapter 2 describes variants of several common image space hidden surface elimination algorithms, which solve the HSE problem for scenes described as collections of polygons. Implementations of these HSE algorithms on a traditional, serial, single microprocessor computer are introduced and theoretical estimates of their performance are derived. The algorithms are compared under identical conditions for various sets of test data. The results of this comparison are then placed in context with existing historical results.

Chapter 3 examines the application of MIMD style parallelism to accelerate the solution of HSE problems. MIMD parallel implementations of the previously considered HSE algorithms are introduced. Their behaviour under various system configurations and for various data sets is investigated and compared with theoretical estimates. The theoretical estimates are found to match closely the experimental findings.

Chapter 4 summarises the conclusions of this thesis, finding that HSE algorithms can be implemented to use an MIMD parallel computer effectively, and that of the HSE algorithms examined the z-buffer algorithm generally proves to be a good compromise solution.

Table of Contents

1 Introduction 1

1.1 Overview 1

1.2 Graphics 5

1.2.1 Graphic Displays 5

1.2.2 The Graphics Pipeline 6

1.2.3 Approaches to High Performance Graphics 8

1.2.4 Examples of Graphic System Architectures 11

1.2.4.1 Sun Microsystems GX Architecture 11

1.2.4.2 Silicon Graphics IRIS 12

1.2.4.3 Stellar GS1000 13

1.2.4.4 AT&T Pixel Machine 14

1.2.4.5 Pixel Planes 5 14

1.3 Parallel Processing 16

1.3.1 Types of Parallel Computer 16

1.3.1.1 SISD Computers 17

1.3.1.2 SIMD Computers 17

1.3.1.3 MISD Computers 18

1.3.1.4 MIMD Computers 18

1.3.1.5 Shared Memory versus Distributed Memory Parallel Computers 21

1.3.2 Programs for Parallel Computers 23

1.3.3 Programs for Distributed Memory Parallel Computers 24

1.3.4 Measuring Parallel Systems 25

1.4 Hidden Surface Elimination 28
1.4.1 Hidden Surface Elimination on Parallel Computers 29
1.5 This Thesis 29
1.5.1 The Parallel Computer Used 30
1.6 Test Data & Environment Statistics 32
1.6.1 Estimating the Cost of Algorithms 35
2 A Comparison of Five Hidden Surface Elimination Algorithms 38
2.1 Introduction 38
2.2 The HSE Algorithms 38
2.2.1 Recursive Subdivision Algorithm 39
2.2.1.1 Cost Estimate 45
2.2.2 Scan Line Algorithms 48
2.2.2.1 Cost Estimate for Scan Line Algorithm Without Edge Tables 55
2.2.2.2 Cost Estimate for Scan Line Algorithm With Edge Tables 57
2.2.3 Z-Buffer Algorithm 60
2.2.3.1 Cost Estimate 61
2.2.4 Painter's Algorithm 62
2.2.4.1 Cost Estimate 64
2.3 Timing Information 65
2.4 Results 66
2.4.1 Recursive Subdivision Algorithm 66
2.4.2 Scan Line Algorithm (Unoptimised) 68
2.4.3 Optimised Scan Line Algorithm 69

2.4.4 Z-Buffer Algorithm 71
2.4.5 Painter's Algorithm 73
2.5 Comparison with Sutherland et. al. 75
2.6 Conclusions 78
3 A Comparison of Five Parallel Hidden Surface Elimination Algorithms 80
3.1 Introduction 80
3.2 The Parallelisations of the Algorithms 81
3.2.1 Recursive Subdivision Algorithm 81
3.2.1.1 Cost Estimate 86
3.2.2 Scan Line Algorithms 88
3.2.2.1 Unoptimised Parallel Scan Line Algorithm 90
3.2.2.2 Cost Estimate for the Unoptimised Scan Line Algorithm 91
3.2.2.3 Optimised Parallel Scan Line Algorithm 93
3.2.2.4 Cost Estimates for the Parallel Optimised Scan Line Algorithm 94
3.2.3 Z-Buffer Algorithm 95
3.2.3.1 Cost Estimate 97
3.2.4 Painter's Algorithm 98
3.2.4.1 Cost Estimate 102
3.3 Timing Information 103
3.4 Results 104
3.4.1 Recursive Subdivision Algorithm 105
3.4.2 Scan Line Algorithms 110

3.4.3 Z-Buffer Algorithm 122
3.4.4 Painter's Algorithm 128
3.5 Comparison of the Algorithms 135
3.6 Conclusions 138
4 Conclusions 140
4.1 Serial HSE Algorithms 140
4.2 Parallel HSE Algorithms 141
4.3 Overall Conclusions 142
5 References 143
6 Appendix 149

Chapter 1

Introduction

1·1. Overview

In the past fifteen years there has been little work on the performance of hidden surface elimination (HSE) algorithms. This is probably due to the area having been extensively considered in the early days of computer graphics, (see for instance the classic survey paper of Sutherland et al [1]).

Reductions in the cost of computer technology has allowed the more general use of computer graphics. Graphic displays are particularly useful in computer aided design (CAD) systems, giving the user a better indication of the object under design. Flight simulators [2,3] are becoming more common and finding new applications for similar cost reasons. Thus the requirement for real-time and near-real-time displays is ever growing.

The HSE process has however remained a significant bottleneck in the performance of graphical systems. This is unlike other parts of the graphics pipeline, such as coordinate transformation which has largely succumbed to the vast performance increases of floating point numeric processing.

The hardware of computer graphics systems evolves in a cyclical fashion [4], changing from a general purpose computer with a frame buffer added, to specialised hardware for painting in the frame buffer and supporting graphics transformations, and back to

the frame-buffer-on-general-purpose-computer approach as the computers of the time themselves evolve.

Although the paper of Myer and Sutherland [4] was written more than twenty years ago, different points within this cyclical evolution may still be seen in current machine architectures. Many modern graphics workstation vendors supply VLSI graphics engines that support coordinate transformation and the painting into a z-buffer of simply shaded polygons [5-7]. There has also been much research into such systems that does not directly appear in product lines [8-11].

Other vendors have taken the approach of using comparatively simple polygon painting hardware and having the computer system's general purpose processor compute the necessary coordinate transformations [12-14].

Specialised graphics hardware has one particular disadvantage - it is utterly inflexible. It provides high levels of performance for the few tasks it supports, but once the type of workload changes this hardware becomes useless. For example, most current graphics workstations support flat and Gouraud [15] shading. Some also support Phong [16,17] shading. Unfortunately, should the user move to more complex shading schemes such as ray tracing [18,19] then all of the computation must be done by the general purpose system processor, leaving the hardware graphics engine as a rather expensive white elephant - a waste of resources that could otherwise be spent improving overall system performance. Of course, hardware support for ray tracing could be added [20] but there is always a limit to what can reasonably be given direct hardware support.

There is now another factor entering into the design of such hardware graphics engines: the bandwidth available to the frame buffer memory is becoming a practical limit. While the relatively recent invention of video RAM [21] has improved the available bandwidth considerably, the matter is still of serious concern [6]. It effectively caps the potential

performance of any graphics system, making it easier for general purpose microprocessors to equal the performance of dedicated hardware.

This limit is not absolute - there are ways to circumvent it, such as by breaking the frame buffer up into smaller chunks of memory, giving a separate data path into each chunk - effectively multiplying the bandwidth by a substantial factor. The disadvantage to such solutions is that they usually increase the physical size, power dissipation, and cost of the graphics system. An interesting variant of the approach is considered in the research of Fuchs et. al. [22,23]. Basically, the Pixel-Planes system integrates many simple microprocessors into the frame buffer. This is logically equivalent to breaking up the frame buffer and connecting each part to its own microprocessor, but the integration of many such frame buffer/processor pairs onto a single chip makes the system a more practical proposition.

The factors discussed above are now causing the appearance of systems that form compromises between direct hardware support for graphics and general purpose microprocessors. This can be seen when considering the Texas Instruments 34010 and 34020 graphics processors [24] which have many general purpose instructions, and the Intel 80860 [25,26] which is a general purpose microprocessor with some graphics support instructions.

General purpose parallel computers are now widely available, whose aggregate computing performance is more than enough to equal dedicated hardware graphics engines, *if* that aggregate performance can be usefully harnessed. Should such a system be an effective graphics engine, then it would have one critical advantage over special purpose hardware - flexibility. Once the program is changed, the entire computing resource may be directed against a totally different problem. This leads to a number of questions regarding the efficient execution of graphics algorithms on such platforms.

The study of parallel graphics algorithms is not new, although much of the existing literature concentrates on optimising algorithms for implementation as parallel functional units on VLSI chips [27,7]. The only area in which this emphasis on specialised hardware design has not occurred is that of ray tracing, which as discussed earlier is not particularly amenable to such implementation. A good overview of parallelism in graphics may be found in Crow [28], which discusses both parallel machines and parallel algorithms.

The solution of HSE problems on parallel machines is likewise not new. Franklin and Kankanhalli [29] considered parallel object space HSE, while most other researchers have concentrated on image space HSE [30-33]. Within the literature covering image space HSE there is a predominance of simulated results [31-33].

This thesis presents the results of comparative tests for HSE algorithms on polygonal models from two viewpoints. First, it compares the performance of several widely used algorithms implemented serially in the same hardware and software environments, and secondly it extends the comparison to parallel implementations of these algorithms.

The four common HSE algorithms considered are recursive (quadtree) subdivision, a scan line algorithm [34-36], the z-buffer algorithm and the painter's algorithm.

The relative performance figures of these algorithms given by Sutherland et al [1] for serial computers are based on order of magnitude estimates, but the quality of their work is underlined by the correlation that exists between their figures and those presented in this thesis for the single processor implementations.

The work covered in this thesis was carried out on a parallel processing system of the multiple-instruction multiple-data stream [37] (MIMD) distributed memory type. Such a system is basically a collection of independent computers, connected by communication

links. In particular the system used was a collection of transputers. The system was programmed in OCCAM [38], which being based upon Hoare's CSP [39] provides a firm foundation for the construction of parallel programs.

Transputers are particularly suited to use in parallel computers due to the inclusion of most of the required parts of a computing node, (i.e. CPU, RAM, and communications links) in a single package. Transputers have almost alone popularised the parallel processing concept by being being both cheap and readily available.

1·2. Graphics

1·2·1. Graphic Displays

Since their invention / introduction, graphic displays for computers have been used for more and more purposes, for a relentlessly growing audience. Initially highly expensive devices, they were first used for military flight simulators. Over the years these have grown in capability, producing ever more lifelike images. As an editable representation of a paper document they have come to be widely used in the creation of documents for publication, and in the design of complex systems (computer aided design).

A graphic display allows a computer to feed data to its user through its user's primary sense, that of sight. Perhaps it was only natural that the humans' need for play has produced games using this feature - video games. It is also the obvious channel to use to inform a computer user what the computer is currently doing, hence the use of windowing systems where a window is connected to each particular task.

Graphic displays normally show their pictures on cathode ray tubes (CRTs), although alternative display technologies are now increasingly available, particularly liquid crystal displays (LCDs). There have been two types of graphic display, vector and frame-buffer.

Vector displays repeatedly draw lines on the screen. Each line is described by start and finish coordinates and perhaps by a brightness value. Within the cathode ray tube (CRT), an electron beam is guided along the defined line. Where the beam strikes the screen, a phosphor coating is excited and glows visibly. A list of such lines or vectors represents the entire displayed information and is repeatedly drawn upon the screen. Vector displays are largely obsolete since their primary advantage over frame buffers - the small amount of memory used - is no longer relevant due to semiconductor technology advances.

Frame buffers store an array of values in memory. This array represents the brightnesses of a rectangular array of points on the screen. This array of points is repeatedly redrawn on the screen, line by line. Each point is known as a pixel.

1·2·2. The Graphics Pipeline

The most time consuming graphics display jobs involve the display of a three dimensional scene, as though seen from a particular viewpoint, with particular lighting and surface details. The three dimensional scene data must be processed to produce suitable colour information, transformed into screen coordinates, processed to avoid the display of objects which should be hidden behind other objects, and finally drawn into the frame buffer for display. This "pipe" through which all scene data must pass is known as the "graphics pipeline".

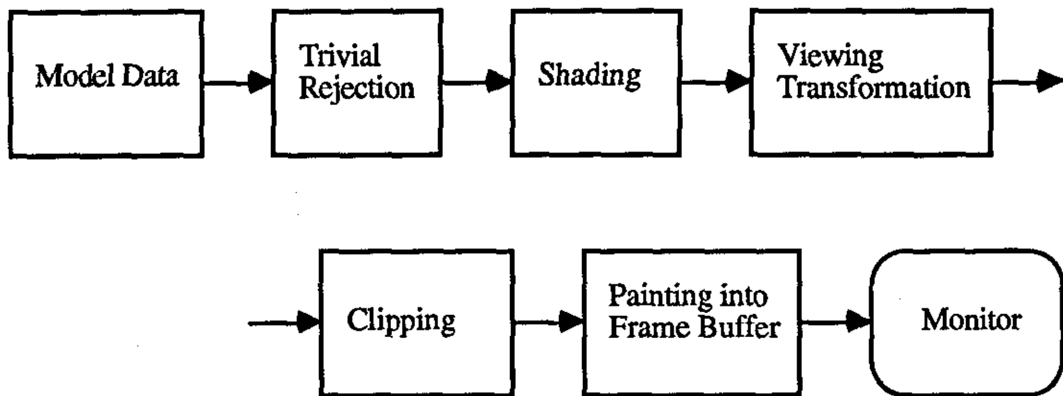


Figure 1·1: *The Graphics Pipeline.*

The pipeline is fed by a database of objects which describes the scene to be displayed. These objects may in turn be constructed from common component objects which are transformed into the correct positions - a transformation step not shown in Figure 1·1. A common version of the pipeline and the one used in this thesis operates purely on polygons, from which all objects must be built. Conventionally these polygons have only one visible side. This is basically a performance enhancement which allows a later step to easily identify and discard polygons on the far side of an object, on the assumption that polygons on the far side of an object - with their visible faces facing away from the viewer - are hidden from the viewer by polygons on the near side of the object. This discarding step is called "the backface cull".

The objects are then tested for clear irrelevance to the visible part of the scene. This step is not fundamentally necessary as it is an approximation of the later clipping step but can save a great amount of work by eliminating many irrelevant objects early in the pipeline.

Each object has its shading according to its surface properties and the lighting conditions, then transformed from the "world coordinates" used to describe the model and lighting to the viewer-centric "viewing coordinates" for display. This

transformation not only takes account of the viewer's position and the direction he is looking in but also of perspective effects. This step is known as the "perspective transformation".

Objects are then "clipped" against the limits of the screen, to avoid wasting time trying to draw objects or parts of objects which fall outside the limits of the screen. Finally, the remaining objects are painted into the frame buffer for display on a screen.

Hidden surface elimination normally occurs somewhere after the viewing transformation step, depending upon the precise HSE method employed. It may form part of the painting step or exist on its own as a distinct step. Its job is to identify the frontmost objects at each point of the screen, discarding hidden surfaces or parts of surfaces.

Each of the steps in the display pipeline may be done to various levels of quality. If a large scene is to be processed the sheer quantity of scene description data may easily tax the computer system. The shading of the scene may be done using different numbers of light sources, differing levels of description of the surfaces being lit. Atmospheric effects may be taken into account or ignored. This shading work may be done for few or many points within the image. The removal of hidden objects and parts of objects may be solved exactly, or to the resolution of the display device (which may leave the question of what colour a pixel containing several edges is unanswered or fudged).

1·2·3. Approaches to High Performance Graphics

The paper of Myer and Sutherland [4] illustrated early approaches to increasing the performance of computer display systems. Unhappy with existing display systems they

decided to design their own, and during this effort discovered that the design of display systems was evolving in a cyclical fashion.

The earliest display systems had been a simple frame buffer directly attached to a host computer. The problem with this method is that the host then spends much of its time doing simple operations on the frame buffer. Subsequently, the display hardware grew from a simple frame buffer to a hardware enhanced frame buffer which could handle simple operations with little intervention from the host. This then grew to be a frame buffer with its own, simple processor. Soon this display processor had more general features added so that more of the display related work could be off-loaded from the host. This choice resulted in a system of two processors, one with an attached frame buffer. Myer and Sutherland considered this point to be one complete turn of their wheel of display system evolution.

In such a system the display processor spends much of its time doing simple frame buffer operations, which is probably a waste of resources for such a general processor. Hence it seems reasonable to enhance the frame buffer to remove simple jobs from the display processor. This step moves the display partly into the second turn of the wheel of evolution.

Myer and Sutherland were able to place the designs of many contemporary display systems within their wheel of evolution analogy. Frustrated by this apparently endless cyclical evolution, they were eventually persuaded of the view that general purpose computing resources should be pooled into a single, central processor. This choice makes more efficient use of available resources and restricts display systems to less than one turn of the wheel of display system evolution. This choice has been reflected in most display systems designed since that time, though there is still some argument as to how complex and how flexible the display support hardware should be.

Major factors governing the design of display systems today are:

(i) Most of the advanced display systems are used to show two or pseudo-three dimensional pictures consisting of lines and/or polygons.

Considerable performance gains may be obtained by the addition of dedicated hardware to draw these lines and polygons. Both lines and polygons are drawn into frame buffers using simple algorithms which are amenable to hardware implementation. Such implementations typically calculate several partial results simultaneously, resulting in good performance compared with an unaided host processor. Better still, such an approach removes one of the most time consuming display-related jobs from the host processor, leaving it free for jobs of a complexity more suited to such a general purpose device.

This solution is amenable to being used many times over in the same system - adding multiple line drawers to a frame buffer can prove worthwhile for up to several tens of such devices.

(ii) Many display jobs require a lot of coordinate transformation calculations.

This work is basically the multiplication of a coordinate vector and a transformation matrix to produce a result coordinate vector. Such work may be profitably offloaded to a dedicated vector processor. Alternatively the host processor's numerical performance may be improved and then be given this work.

(iii) Currently, points (i) and (ii) have been exploited to their practical limits. Display systems have run into a limiting factor - the available bandwidth to the frame buffer memory. One way around this is to break the frame buffer up, resulting in several smaller frame buffers, each with the same available bandwidth the single large frame

buffer would have. Some display hardware, (e.g. a polygon painter) may then be attached to each small frame buffer. If this approach is taken to extremes, the part count becomes rather high and so the processor and display memory may then be fabricated on the same silicon chip. Such a device is typically referred to as a logic enhanced memory.

1·2·4. Examples of Graphic System Architectures

There have been many examples of using hardware to accelerate graphics. These tend to support a few basic types of graphics operations, as discussed previously, generally line and polygon drawing/filling with simple shading equations. Hardware support for more complicated graphics operations do not seem to have been economically viable to date, but several research efforts such as the Pixel-Planes architecture (discussed below) have shown how this may be implemented.

Hardware support for graphics usually implies the use of many, simple functional units to "execute" the graphics algorithm in parallel.

1·2·4·1. Sun Microsystems GX Architecture

The GX graphics accelerator architecture [6] is a good example of a mass-market display system. It was intended to become the "least common denominator" in Sun's graphics systems, and has largely done so. One of the GX's design goals was that it should survive for several years, requiring that the hardware support for graphics should not become the bottleneck of a system with a much faster processor. This implied that the GX should be able to saturate the frame buffer interface, which is the best an infinitely

fast processor could manage. [This argument assumes that frame buffer memory does not significantly increase in speed with time, which has so far proved true.]

The GX contains two major functional blocks, the Transformation Engine (TE) and the Frame Buffer Controller (FBC). The TE handles coordinate transformation work at a rate of up to 50 MFLOPs. Its output is fed to the FBC, which draws flat shaded quadrilaterals into the frame buffer, clipping them against a rectangular region. The FBC can also copy rectangular areas of the screen image to another place on the screen, and provides some support for drawing text. The GX is one of the most simplified graphics accelerators, in that it implements a very restricted set of basic operations in hardware.

1·2·4·2. Silicon Graphics IRIS

This system [5] is a well known landmark of computer graphics. It used internal parallelism to achieve its high performance. It included five geometry engines which executed coordinate transformations, shading calculations and clipping. The output from these is fed to a polygon processor which breaks the polygon into trapezoids and calculates the gradients of the edges and colours for each trapezoid. The trapezoids are then fed to the edge processor which breaks them into vertical stripes. These are fed to one of five span processor, depending upon their x-coordinate. Each span processor turns stripes into per-pixel information which is fed to one of four image engines, depending upon y-coordinate. The image engines, (of which there are twenty in total), are little more than memory controllers which can do z-buffer style pixel painting.

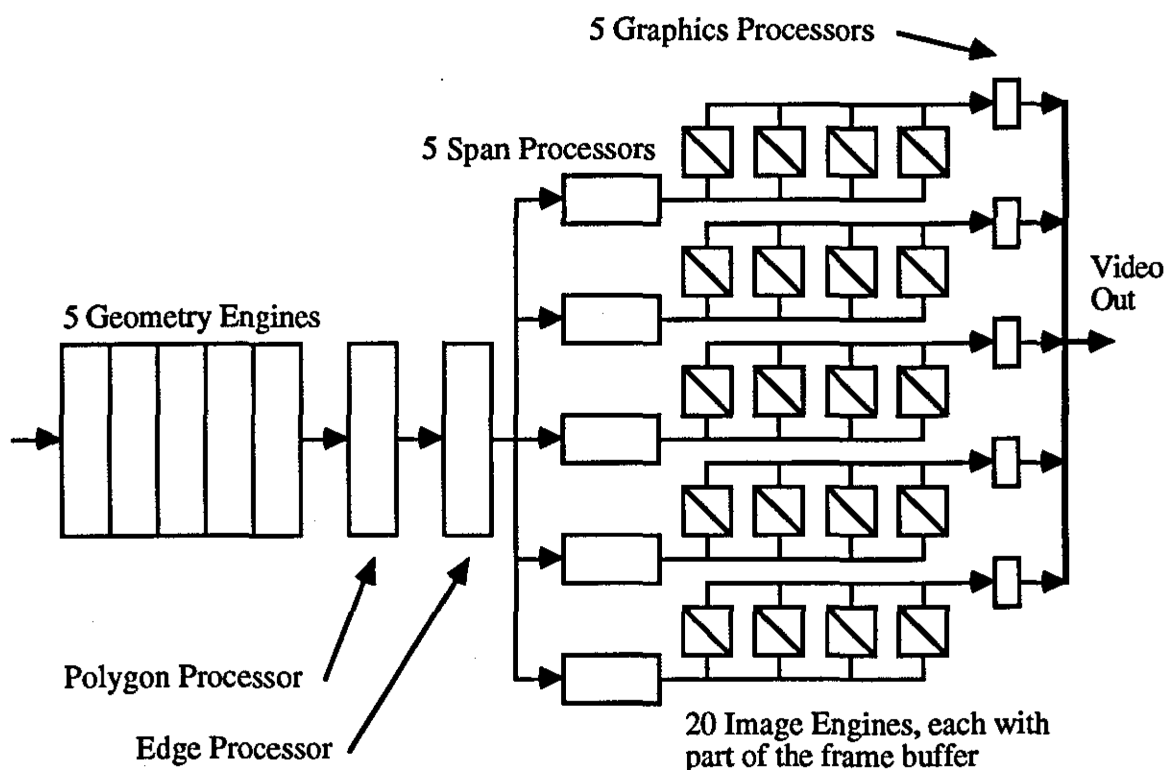


Figure 1·2: *The graphics architecture of the Silicon Graphics IRIS.*

This design is a very good example of the use of hardware parallelism to support graphics. It consists of a large number of functional units, with each section of the graphics pipeline balanced in capability to support an overall throughput goal with a minimum of wasted resources.

Later, low-end Silicon Graphics machines have sometimes omitted the geometry engines in favour of doing the coordinate transformation and shading work on the increasingly powerful, central microprocessor.

1·2·4·3. Stellar GS1000

The Stellar GS1000 [12] chose from the beginning to do its coordinate transformation and shading work using a maths unit shared with the rest of the system. This maths

unit was however significantly more powerful than most workstation maths units of that time, in keeping with the machine's title of Graphics Supercomputer. The rendering hardware consists of two major blocks, the set-up engine and the foot print engine. The set-up engine processes incoming primitives into equations and coefficients of a form suitable for the foot print engine.

The foot print engine consists of sixteen toe processors, arranged in a four by four grid. It simultaneously solves for sixteen pixels the equations it has been passed by the set-up engine. Apgar et. al. commented that while an eight by eight array of toe processors would provide a speedup of approximately three times over the four by four grid used, any larger a foot print would suffer from low efficiency and would most probably not be cost efficient. This comment is interesting in light of the contrast between this graphics architecture and that of the Pixel Planes series of machines discussed later.

1·2·4·4. AT&T Pixel Machine

The AT&T pixel machine [28] is a MIMD parallel computing machine. It is basically an array of 16 to 64 processors, each of which is connected to part of the screen memory. This array is fed work through a single pipeline of 18 processors, or dual 9 processor pipelines. These pipelines are generally used for transformation work.

Interestingly, these machines seem to have been used only rarely for interactive work, instead being used for more time consuming jobs such as ray tracing.

1·2·4·5. Pixel Planes 5

The Pixel Planes experimental graphics engines have explored the combination of a processor attached to each pixel. The combination resulting from this is called a logic

enhanced memory, which has the advantage of avoiding bottlenecks between the processors and screen memory. Each processor is one node of an SIMD processor array and solves quadratic equations. However, this approach unfortunately leads to a rather low efficiency of use of the processors, especially when rendering small primitives.

In Pixel Planes 5 there is no longer one processor per screen pixel. Instead, a variation on this approach is used to increase efficiency. Pixel Planes 5 [22,23] has three types of functional blocks - graphics processors (GPs), rasterisers, and the frame buffer. The GPs use general purpose microprocessors to handle coordinate transformations. They can locally store many primitives, which avoids having to reload the primitives for each new frame. Each rasteriser contains an SIMD array of processor / pixel memory nodes which render an area of 128 by 128 pixels. All of these functional blocks communicate over a ring network. The system may contain any number of rasterisers.

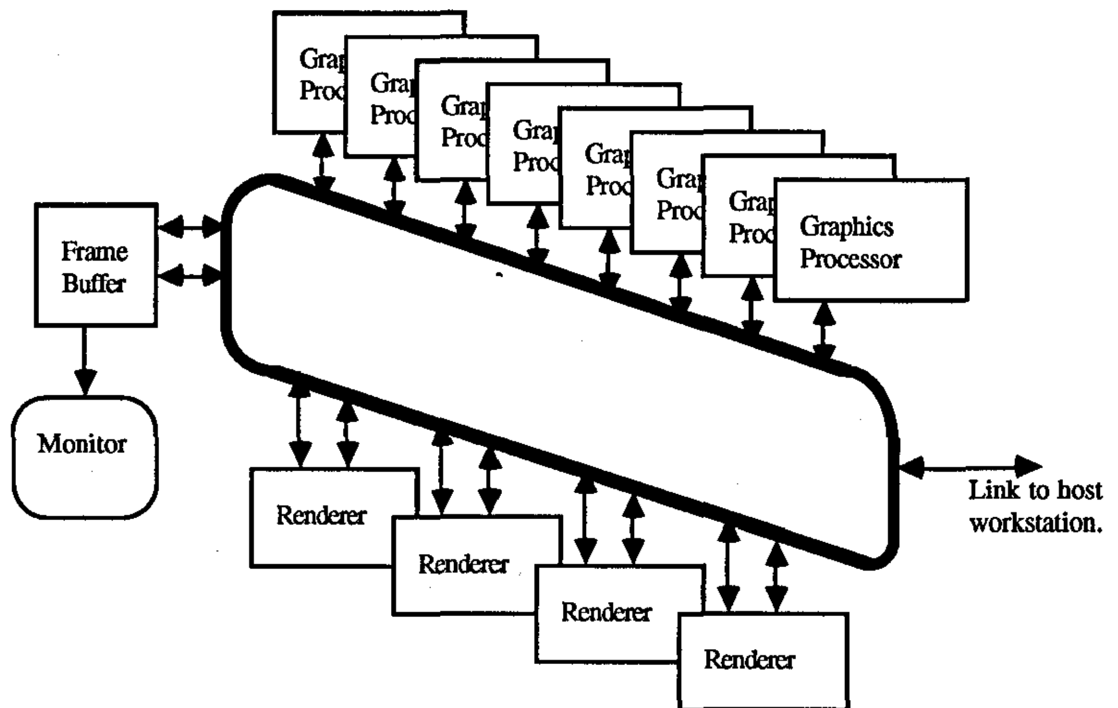


Figure 1·3: *The structure of Pixel Planes 5.*

1·3. Parallel Processing

Parallel processing is an obvious approach to high performance computing. Once the performance of a single processing element approaches its practical bounds, the combination of several such elements will possibly provide a greater performance boost than investing the same amount of resources in increasing the performance of a single element. If many such elements are combined then economies of scale may result, further pushing the balance toward the "multiple elements processing in parallel" approach.

Parallel processing is basically the art of avoiding performance bottlenecks in computer systems. There are two main facets of parallel programming:

(i) the programming of parallel computers.

(ii) the use of parallelism within the program structure to produce more straightforward or simply elegant programs.

To use a parallel processing system suitable algorithms are needed. The history of computing is however largely one of single processor systems, resulting in a library of existing algorithms which expect to be run upon a single processor. Some of these are trivial to extend to parallel processor systems, while others are utterly unsuited to such treatment. To discover whether an algorithm is suitable for parallel implementation the nature of available parallel systems must be considered.

1·3·1. Types of Parallel Computer

There have been several attempts at classifying parallel computers. Few of these are generally popular. The most enduring system is that of Flynn. He classified systems according to the number of instruction and data streams used. Systems are then referred

to as "nInD" where "n" is either "S" for "single" or "M" for multiple. There are four possible basic types, described below. There are many variations on each of these basic types, and several "in between" machines.

1·3·1·1. SISD Computers.

This is simply a typical single processor system, with a single instruction stream and a single data stream. Computation is done in the ALU (arithmetic logic unit). The ALU is driven by the control section which gets its instructions from the instruction stream. Data for the ALU is acquired through a single data stream.

Figure 1·4: *The simplest system in Flynn's taxonomy - SISD. Such a system has one control unit and one ALU, with a single instruction stream and a single data stream.*

1·3·1·2. SIMD Computers.

The first multiple processor systems sought to economise by having multiple ALUs to operate on data but only a single control unit. Such a system has only a single instruction stream, driving many ALUs. Each ALU has its own data stream, so there are multiple data streams. These machines are well suited to data parallel problems, where the same operations are executed on every piece of data. However, they suffer considerably when required to handle complex algorithms where the operations executed on a piece of data are a function of the data itself.

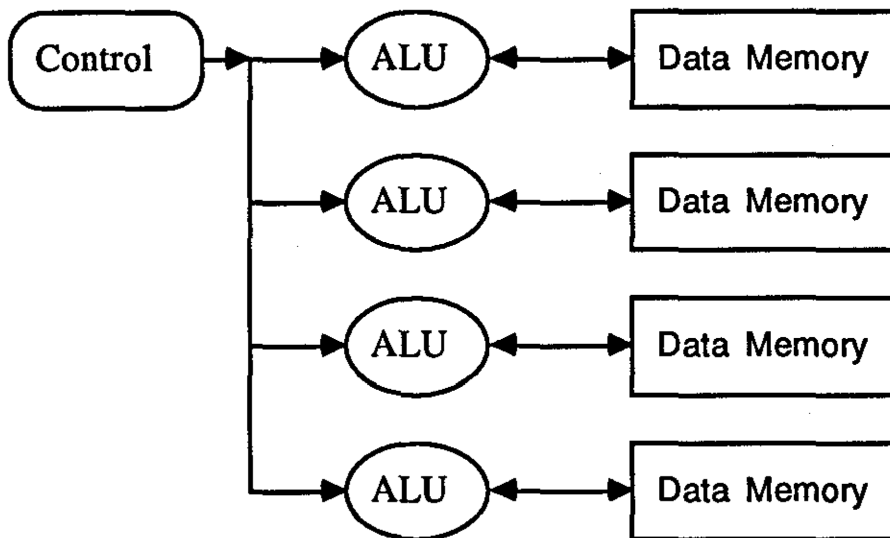


Figure 1·5: *An SIMD computing system. It has one control unit driving many ALUs and so has a single instruction stream, but many data streams.*

1·3·1·3. MISD Computers.

These are rare. They do many things at once to only one set of data. This classification is normally considered to refer to code breaking machines, trying many ways to decrypt an encoded message (the data stream).

1·3·1·4. MIMD Computers.

These are fully parallel. They have multiple control units, each controlling its own ALU. There are multiple instruction streams and multiple data streams. There are two notable variations on this theme - shared memory and distributed memory machines. In a shared memory MIMD computer, each control unit / ALU pair can access every memory location in the entire machine. In a distributed memory MIMD computer every control unit / ALU pair has its own memory which none of the other pairs can access.

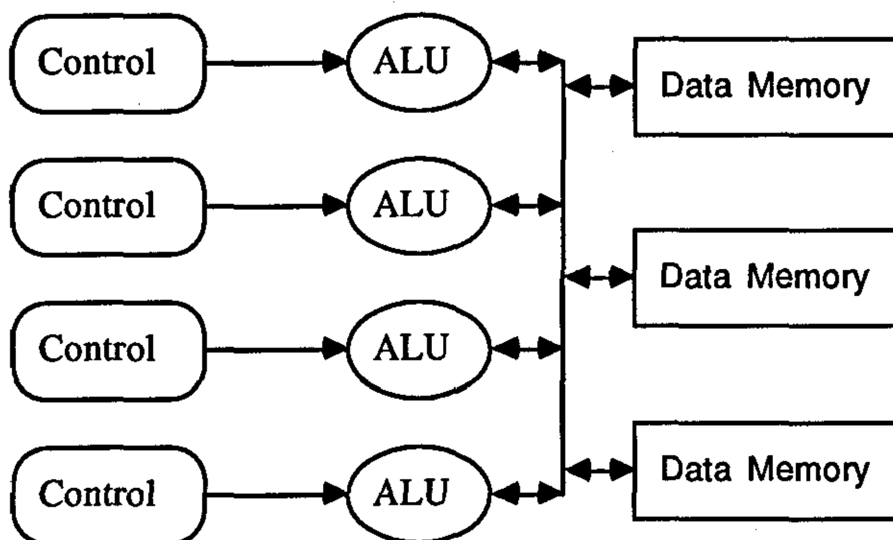


Figure 1·6: *A shared memory MIMD computing system. It has many control units each driving one ALU and so has many instruction streams and many data streams. Every ALU can access every memory.*

Distributed memory MIMD computer systems are very similar to collections of SISD computers. The difference is that the SISD elements of the MIMD system must be able to communicate with each other. This is normally done by providing either a communication bus or point-to-point communication links. In the case of a bus, when one element transmits information it is visible, but not necessarily of interest to every other element.

There are many ways of connecting point-to-point links. Ideally each element would have one link direct to each other element. Unfortunately this results in an impractically large number of links for a large system. Instead, the elements are usually connected in a 2- or 3-dimensional grid or as an n-dimensional (hyper)cube. Typically a single bus is inadequate and point-to-point links are easier and cheaper to implement than multiple buses.

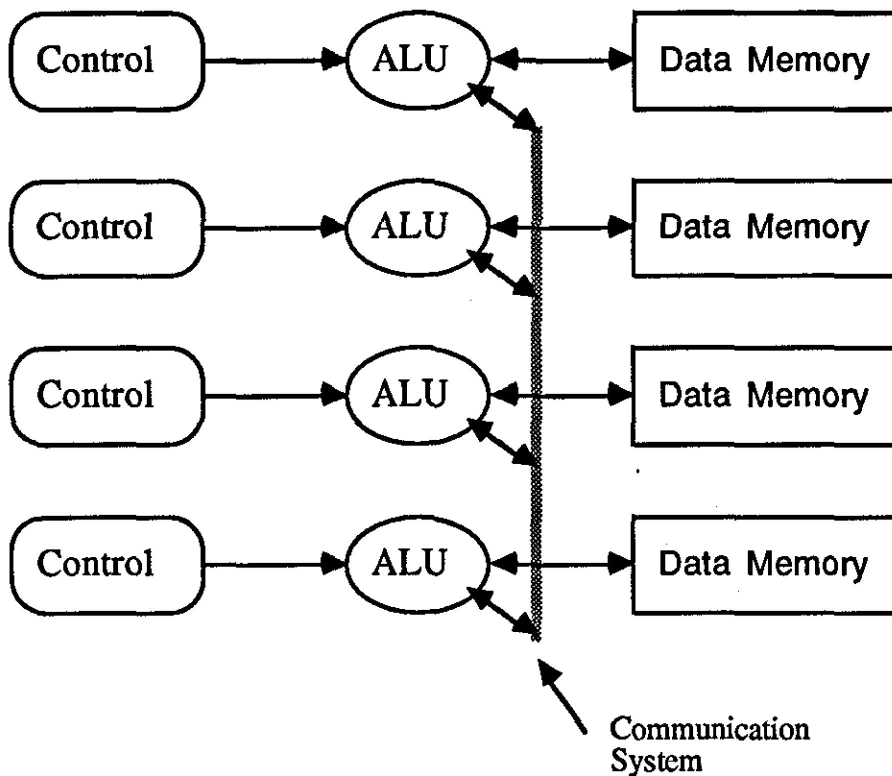


Figure 1·7: *A distributed memory MIMD computing system. It has many control units each driving one ALU. Each control unit / ALU pair has its own memory. Each ALU can only access its own memory. To be useful there must be some form of communication between each control unit / ALU / memory section.*

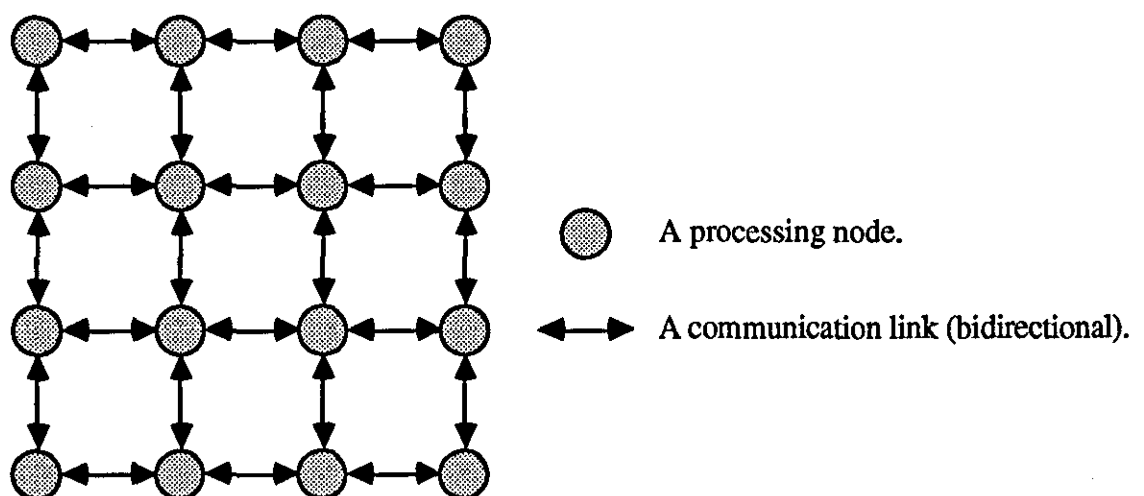


Figure 1·8: *A two dimensional connection network for a distributed memory MIMD computing system. This particular example is a four node by four node network.*

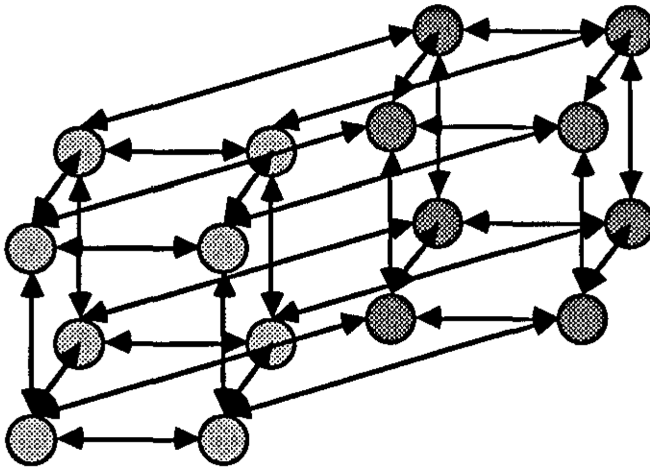


Figure 1·9: *A four dimensional hypercube connection network for a distributed memory MIMD computing system. A four dimensional hypercube may be constructed by connecting two three dimensional (hyper)cubes. The two component cubes are shown by different node shadings.*

1·3·1·5. Shared Memory versus Distributed Memory Parallel Computers

Shared memory systems have one particular advantage over distributed memory systems - all of the processors share the same view of memory. Because of this it is relatively simple to modify existing single processor programs to use a shared memory multiprocessor if there is any possible parallelism in the program.

Shared memory MIMD systems typically do not exist with more than about thirty processors. This is due to the fact that in a shared memory system all of the processors share the same bus and memories, causing these resources to be saturated at about this point.

The use of hybrid shared / distributed memory systems, where each processor in a shared memory system also has a purely local memory of its own, can help reduce traffic to the shared memories. However, for this to be really useful the local memories

must be used almost exclusively and the shared memories must be largely ignored, effectively reducing the system to a distributed memory system.

The use of program and data caches with each processor can also reduce traffic to the shared memories, by storing the recent traffic and answering the requests themselves if a request is repeated. Unfortunately modern processors are so much faster than the available memories that caches are necessary for even single processor systems and hence cannot help much in a shared memory multiprocessor.

There have been a few exceptions to the "thirty processor" rule. Usually these systems have used a switching network between the processors and shared memories in place of the simple bus normally used. Unfortunately these systems have seemingly proved unsuccessful and disappeared from the parallel computer market, perhaps because the switching network has usually proved to be a major component of the cost of the system.

Distributed memory systems have several helpful qualities, particularly in that the overall cpu to memory bandwidth increases in direct proportion to the number of processors, (since adding a new processor implies adding a memory with it). This means that the practical limit on the numbers of processors that can be reasonably be incorporated into a distributed memory parallel computer is much higher than for shared memory parallel computers.

Instead of processors communicating through the shared memory of a shared memory system, in a distributed memory system communication normally takes place over point to point links. When adding further processors, further links are also added to connect the new processors into the system. As a result of this, adding processors also implies increasing the processor to processor bandwidth, again avoiding the bottlenecks which stop the growth of shared memory systems.

The major disadvantage of distributed memory computers is that they are potentially difficult to write programs for, as will be explained in the next section. Also, they are rather hard to modify existing single processor programs for, since each processor sees a different memory and any communication between parts of the program executing on different processors must be explicitly programmed.

1·3·2. Programs for Parallel Computers

Since a parallel computer consists of a number of processors communicating with each other to coordinate their actions and pass partial results, a significant amount of interprocessor communication may result. Whether such traffic actually does occur or not will depend upon the algorithm being used. Efficient use of a multiprocessor system requires that most of the processors' time is spent computing, not communicating.

For an n-processor system to outperform a single processor system merely requires that each of the n processors computes for at least 1/n of its time. If a system has more than a few processors but each does not compute for significantly more than 1/n of its time it is probably a waste of resources, since it is not much faster than a single processor. So for a multiprocessor system to be useful it must spend most of its time computing, and as little time as possible communicating.

Using a parallel processing system requires that the algorithm can be parallelised. In some cases every step of the algorithm requires that all of the earlier steps be complete and have delivered their results. In this case it is impossible to parallelise the algorithm.

For algorithms where there is no way to avoid high levels of interprocessor communication or where there is no way to parallelise the algorithm at all, an alternative algorithm must be found which is amenable to parallelisation. Otherwise a parallel

processor is of no help and the fastest solution of the algorithm will be found using the fastest single processor machine.

1·3·3. Programs for Distributed Memory Parallel Computers

In a typical shared memory parallel computer, the bus between the processors and shared memories is a bottleneck. In a distributed memory parallel computer there is no obvious bottleneck. This does not mean that there are actually no bottlenecks, but should one exist it is less obvious. For example, if some particular data structure should be central to all stages of a computation, then all parts of that computation must clearly be able to access the data structure. In a parallel computer these various parts of the computation will be executing on separate processors. The question thus arises, "where should the data structure be placed?" Unfortunately, no matter where it is placed some processor will face a considerable delay accessing it.

There are two aspects of interprocessor communication in a distributed memory parallel computer which may slow the access of remote data. These are the bandwidth of the connections and the latency of reply. The bandwidth is the rate at which data may pass along a communication link and is of concern only when passing large amounts of data. The latency is the delay before the reply to a request starts arriving.

The bandwidth of interprocessor connections may be increased to alleviate congestion, though such remote connections never equal a local memory access for bandwidth. Problems with latency are however becoming unavoidable - as computation rates increase the effect of the "speed of light" physical limit for electronic signals is becoming more significant, where a remote memory takes longer to access than a local one simply by virtue of the fact that it is further away. In a distributed memory parallel computer there are few obvious bottlenecks and so these systems may be built with

very many processors and may consequently be physically very large, making the latency problem severe.

1·3·4. Measuring Parallel Systems

Note: This section aims to create a few simple metrics of parallel computer system and program performance. However, no single, simple to measure figures are going to offer more than an approximate guide to such complex systems.

An algorithm that can make effective use of a parallel computer must not make more than light use of non-local information. To take a more quantitive view of this factor, "light use" must be calculated with reference to the particular parallel computer under consideration. A good starting point for such a system of metrics is the ratio of interprocessor bandwidth to processor instruction rate. This is based on the assumption that a program will on average send/receive so much data for every so much program executed. Since processor instruction rate is a rather difficult quantity to measure in a machine independent way, an alternative measure must be used instead. For purely numerical computations, the possible rate of floating point calculation offers a reasonable approximation. For non-numeric calculations, the choice of metric is much harder since the instruction rate of a machine has long since been discredited as a measure of machine performance. For such jobs, the processor-to-memory bandwidth is suggested. This gives a quality factor for the machine, $Q_{machine}$ or Q_m:

$$Q_m = \frac{\text{Processor to Processor Bandwidth}}{\text{Processor to Memory Bandwidth}}$$

Q_m is thus basically the cost of non-local data access divided by the cost of local data access. The processor-to-memory bandwidth figure can still be hard to measure, for it

does not state how systems with caches are to be treated. The numerical alternative is thus $Q_{machine,numeric}$ or Q_{mn}:

$$Q_{mn} = \frac{\text{Processor to Processor Bandwidth}}{\text{Floating Point Operation Rate}}$$

There is then an obvious similar metric for a program, the ratio of computation to communication within the program. The amount of communication may usually be estimated fairly simply by examining the program, but again the question of how to fairly measure computational costs over a range of machines arises. For primarily numerical computations the number of floating point operations may be used, leading to a quality factor for the program, $Q_{program,numeric}$ or Q_{pn}:

$$Q_{pn} = \frac{\text{Floating Point Operations}}{\text{Amount of Communication}}$$

For non-numerical computations, there is no clear alternative to instruction rate. Experience suggests that a scheme of costs per instruction type, yielding a weighted instruction count, gives a reasonable approximation to the real program cost. Such a metric should not be taken too seriously since a single figure is never going to provide an accurate description of computing costs. This gives:

$$Q_p = \frac{\text{Weighted Instruction Count}}{\text{Amount of Communication}}$$

As a simple example of the use of these metrics, consider the performance of a highly numerical program on a network of Inmos T414 microprocessors and on a similar network of Inmos T800 microprocessors [40]. For both transputer types, at 20MHz clock rate, with 4-cycle external memory access, the (external) memory bandwidth is 20Mbytes per second. The link speed may be 20Mbits/s for each of four links. This

gives (naively) 10Mbytes/s interprocessor bandwidth. The floating point operation rates are approximately 1MFLOP for the T800 and 0.15MFLOP for the T414. Hence

$Q_{mn,T800} = 10$ MB/MFLOP

$Q_{mn,T414} = 70$ MB/MFLOP

$Q_{m,T800} = 0.5$

$Q_{m,T414} = 0.5$

This suggests that a T414 network will be easier to use efficiently for numerical work than a T800 network, but that for non-numerical work they will be almost identical. It does not mean that the T414 network is necessarily preferable to the T800 network for numerical calculation, since it would take seven T414s to equal the numerical calculation rate of the T800. Extending this to the Inmos T9000 [41] and Texas Instruments 320C40 microprocessors [42] gives:

$Q_{mn,T9000}$ = 50MB/s / 25MFLOPs = 3.3 MB/MFLOP

$Q_{mn,320C40}$ = 120MB/s / 50MFLOPS = 2.4 MB/MFLOP

$Q_{m,T9000} = 50 / 50 = 1.0$

$Q_{m,320C40} = 120 / 100 = 1.2$

These figures suggest that it is getting harder to efficiently use a parallel computer for numerical work, but slightly easier for non-numerical work. However these figures should not be taken too seriously for several reasons:

1. They are all approximations anyway.
2. They are based on manufacturer's data, which can be misleading.
3. They ignore other aspects of the microprocessors considered, such as the T9000's elegant message routing system which should eliminate the software routing of messages, a time consuming job.

Two much more objective measures of the advantages of a parallel program are "speedup" and "linearity of speedup". Speedup is simply how many times faster the algorithm executes compared with the one processor case. Linearity of speedup could also be called the efficiency of parallelisation in that it measures the fraction of the maximum possible speedup obtained in practice. So:

$$\text{Linearity of Speedup} = \frac{\text{Speedup}}{n} \qquad \text{where } n = \text{no. of processors}$$

and

$$\text{Speedup} = \frac{\text{Execution Time for 1 Processor}}{\text{Execution Time for } n \text{ processors}}$$

1·4. Hidden Surface Elimination

Hidden surface elimination (HSE) is one of the earliest computer graphics problems. Given a collection of objects in three dimensional space and the point and direction from which they should be viewed, the problem is to decide which parts of which object are visible to the viewer. Alternatively, the problem may be seen as that of discarding or eliminating those parts of objects which cannot be seen by the viewer because they are hidden by other objects. The earliest HSE work was sometimes called "hidden line elimination", which sought to solve the same problem for scenes displayed as line drawings.

Much of the early HSE work done considered only objects constructed from polygons. While other representations have become more popular over the years, the simplicity of polygon based descriptions has ensured that they are still in wide use today. Variations of four of the major polygon based HSE algorithms are extensively described in chapter two. The HSE algorithms considered are recursive (quadtree) subdivision, two variants of a scan line algorithm [34-36], the z-buffer algorithm and the painter's algorithm.

1·4·1. Hidden Surface Elimination on Parallel Computers

The majority of the work on parallel HSE has covered ray tracing. As mentioned in section 1·4, most of the work on the parallel implementation of polygon based hidden surface elimination algorithms has taken the form of simulations, or of optimisations for implementation as parallel functional units on VLSI chips.

Of the few papers on parallel computer, polygon based HSE, Franklin and Kankanhalli [29] considered a parallel object space HSE algorithm while most other researchers have concentrated on image space HSE. Of these, Parke [33] simulated the performance of three types of multiprocessor z-buffer and Hu and Foley [31] also simulated a number of varieties of z-buffer. Fiume et. al. [30] experimented with a parallel scan conversion algorithm on the experimental shared memory Ultracomputer. Strothotte and Funt [32] designed and simulated a parallel computer and display algorithm solely for the display of rotating objects. Unfortunately none of these papers investigated whether some HSE algorithms are more suitable for parallel implementation than others.

1·5. This Thesis

This thesis investigates the application of a general purpose distributed memory MIMD computing resource to the graphics problem of hidden surface elimination. With increasing numbers of such machines becoming available the possibility of using them as flexible, quick interactive graphics resources has become worth investigating.

The method discussed in this thesis is the use of a collection of general purpose processors each with a small attached display memory. While not as fast for line drawing (or whatever is given hardware support) as a hardware accelerated graphics system such as those discussed in section 1·2·3, it still provides a reasonable way of

accelerating graphics work while simultaneously remaining a much more general purpose device. If the display algorithm is changed, a hardware graphics accelerator becomes unusable and must be redesigned, but the more general system described here could simply be loaded with a new program.

The problem of limited bandwidth to the frame buffer is also altering the balance between dedicated hardware and general purpose microprocessors. Since modern microprocessors are almost able to saturate the interface to the frame buffer, dedicated hardware can no longer make better usage of the frame buffer. Part of the historical advantage of dedicated hardware has been that it could draw many pixels into a frame buffer in the time it took a general purpose microprocessor to draw one pixel. The frame buffer bottleneck has largely negated this advantage.

This thesis presents the results of comparative tests for HSE algorithms on polygonal models from two viewpoints. First, it compares the performance of several widely used algorithms implemented serially in the same hardware and software environments, and secondly it extends the comparison to parallel implementations of these algorithms.

1·5·1. The Parallel Computer Used.

This work was carried out on a network of Transputers [40]. Each Transputer is a processing element incorporating a CPU with integrated floating point unit, four serial communications links, and some on-chip RAM. Each Transputer also had at least a further 1Mbyte of off-chip RAM connected to it.

Since each Transputer has only four links to its neighbours, the maximum size of a fully connected network, (where every element has a direct connection to every other element) is five Transputers. For relevance to larger networks a fully connected

network was not used. Instead the structure diagrammed in Figure 1·10 was adopted, with one master element acting as file store, and a number of worker elements chained to it. This "processor farm" structure may be seen throughout the literature, for example in Packer [43] and Bez [44].

This arrangement has a potential problem in that all data must pass down the "chain" of workers instead of direct to its destination, possibly causing a communications bottleneck. In the algorithms in this thesis all of the processors require local copies of the polygons. With each polygon averaging 60 bytes and a single transputer link having a bandwidth of approximately 1.5 Mbytes/s, at most 25000 polygons per second may be transmitted. So for the cases considered in this thesis, transmission time would take at best between 0.01 and 0.1 seconds, (for the largest and smallest sets of polygons respectively).

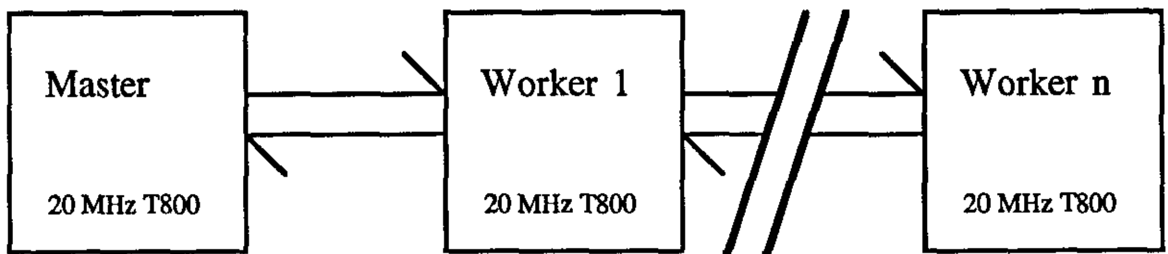


Figure 1·10: *The connection structure of the parallel computer.*

Since the design of the transputer is such that it may simultaneously receive and transmit data, one transputer may be passing polygon k to its downstream neighbour while receiving polygon k+1 from its upstream neighbour, allowing the polygons to be passed along at full link speed. The cost of adding a processor to a pipeline is therefore a one polygon delay, (roughly 0.04 milliseconds).

Should this system be found inadequate, (perhaps for larger data sets), then a second pipeline could be added using the remaining two links per processor. Even greater data

rates could be achieved by using a memory bus and writing the same data to all of the processors at once. Such busses easily operate one or two orders of magnitude faster than transputer links, but require additional hardware.

1·6. Test Data & Environment Statistics

The test data consisted of two sets of scenes. The basic designs of the scenes were chosen for their familiarity within the computer graphics community. All of the scenes in the first set consisted of a row of six teapots. The viewpoint and viewing direction were set so that the row was seen almost exactly end on. Five versions of this scene were used, the difference between them being the number of polygons used to describe the scene. The nominal numbers of polygons in the data sets were 200, 500, 1000, 2000, and 2500 polygons. These numbers are for the backface-culled scene, (i.e. there are no backfacing polygons in the data sets). The upper limit on the number of polygons used in a data set was a consequence of the available memory on each processor of the development system. The lower limit was set by the practicality of describing six teapots with a small number of polygons.

The second set of scenes were the "tetra" scene from Haines [45], a recursively generated tetrahedral scene. Allowing the generating program an extra level of recursion causes each tetrahedron in the scene to be replaced by four smaller ones, at the vertices of the original. Three versions of this scene were used, with varying numbers of tetrahedra (and hence polygons).

All perspective projection, back face culling, and (flat) shading was done in a preprocessing stage since these operations are common to systems involving any of the HSE algorithms considered. This decision is compatible with Sutherland et al [1].

Key to Environment Statistics		
Statistic	Description	Rule of Thumb
n	Vertical screen resolution (in pixels)	Given
m	Horizontal screen resolution (in pixels)	Given
F_T	Number of forward facing faces	Given
F_C	Average number of faces per cluster	Given
D_C	Depth complexity	Given
F_t	Total number of faces	$2 F_T$
C_t	Number of clusters	F_t / F_C
E_t	Total number of edges	$4 F_t$
E_T	Number of edges on forward facing faces	$E_t / 2$
E_C	Number of contour edges	$\frac{E_T}{\sqrt[2]{\frac{1}{2} F_C}}$
E_S	Number of edges on forward faces if sharing is allowed	$\frac{1}{2}(E_T - E_C) + E_C$
X_T	Total number of edge crossings in the viewing plane	$(D_C - 1) \frac{E_T}{4}$
X_V	Number of intersections of visible edges	X_T / D_C
H_f	Average face height (in pixels)	$\sqrt[2]{\frac{n\ m\ D_C}{F_T}}$
S_l	Average number of segments per screen line	$\sqrt[2]{\frac{D_C F_T m}{n}}$
S_V	Average number of visible segments per screen line	S_l / D_C
L_V	Total length of visible edges (in pixels)	$2 n S_V$

Table 1·1. *A key to the various scene measurement statistics. After Tables I and II of Sutherland et al* [1].

A summary of the properties of the data sets is shown in Tables 1·2 and 1·3. Table 1·1 is a key to the various statistics and the rules of thumb used for calculating many of them. The rules of thumb are those of Sutherland et al [1]. There are five basic statistics which must be measured for each data set - n, m, F_T, F_C and D_C. The first two are simply measures of the vertical and horizontal display resolution respectively. F_T is the number of polygons facing toward the viewpoint, (polygons are assumed to have one visible side and one invisible side). F_C is the average number of faces per polygon cluster, (where a cluster is a group of polygons clearly separate from all other polygons). D_C is the average depth complexity of the scene, which is defined as the number of overlapping, forward-facing polygons at a given point.

All of the other statistics are calculated from these five basic quantities using Sutherland et. al.'s rules of thumb. These statistics are F_t, C_t, E_t, E_T, E_C, E_S, X_T, X_V, H_f, S_l, S_V, and L_V. F_t is the total number of faces in the scene, including the invisible ones facing away from the viewer. C_t is the number of clusters in the scene. E_t is the total number of edges in the scene, while E_T is the number of edges on forward-facing polygons. E_C is the number of edges per contour. E_S is the number of distinct edges on forward-facing polygons, i.e. edges shared by two polygons are counted only once. H_f is the average face height. X_T is the number of edge crossings for edges on forward-facing polygons projected into the viewing plane, and X_V is the number of visible edge crossings. S_l is the average number of segments per screen line, i.e. the average number of forward-facing polygons per screen line, while S_V is the average number of visible segments per screen line. L_V is the total length of visible edges.

The properties of the data sets used in this thesis are very similar to those of the scenes described in Sutherland et al [1], except for those measures that depend upon the clustering of the polygons, (F_C, C_t, E_C, E_S). This difference should be irrelevant as none of the algorithms described in this thesis make any use of the clustering of

polygons. Several of the statistics in Table 1·3 are clearly incorrect, such as the property X_V which is negative for all of the tetra scenes. This anomaly is due to the depth complexity of the scenes being less than one on average, while the rule of thumb used to calculate X_V assumes a depth complexity greater than one.

1·6·1. Estimating the Cost of Algorithms

When an algorithm's cost is estimated in this work, the cost is reduced to a number of independent terms. Each of these terms is stated in terms of those environment properties referred to as "given" in Table 1·1 - n, m, F_T, F_C and D_C. Actually, no cost is expressed in terms of F_C because this statistic is a measure of the clustering of polygons within a scene, but none of the HSE algorithms uses clustering in any way. With clustering being irrelevant, F_C may effectively be eliminated in favour of F_T. Little use is made of calculations of costs calculated in terms of n or m due to their being fixed for all of the work described here. Also, there is little interest in the growth of algorithm costs as a function of screen resolution (i.e. n and m) because this has changed little over many years.

Environment Statistics for the Teapot Scenes					
	Model Size (nominal)				
Statistic	200	500	1000	2000	2500
n	512	512	512	512	512
m	512	512	512	512	512
F_T	205	499	1027	2035	2575
F_C	68.33	166.33	342.33	678.33	858.33
D_C	3.029	3.005	3.005	3.019	3.015
F_t	410	998	2054	4070	5150
C_t	6	6	6	6	6
E_t	1640	3992	8216	16280	20600
E_T	820	1996	4108	8140	10300
E_C	140.29	218.87	313.99	442.00	497.19
E_S	480.14	1107.43	2211.00	4291.00	5398.60
X_T	415.945	100.495	2059.135	4108.665	5188.625
X_V	137.32	332.94	685.24	1360.94	1720.94
H_f	62.24	39.73	27.70	19.72	17.52
S_l	24.92	38.72	55.55	78.38	88.11
S_V	8.23	12.89	18.49	25.96	29.22
L_V	8424.17	13195.57	18930.54	26585.85	29925.74

Table 1·2. *A summary of the properties of the five teapot scene descriptions used, given in the format of Table II of Sutherland et al* [1].

Environment Statistics for the Tetra Scenes			
	Model (Size)		
Statistic	Tetra 4 (156)	Tetra 5 (624)	Tetra 6 (2496)
n	512	512	512
m	512	512	512
F_T	156	624	2496
F_C	312	1248	4992
D_C	0.351	0.391	0.469
F_t	312	1248	4992
C_t	1	1	1
E_t	1248	4992	19968
E_T	624	2496	9984
E_C	49.96	99.92	199.84
E_S	336.98	1297.96	5091.92
X_T	-101.2	-380.0	-1325.4
X_V	-288.4	-971.9	-2825.96
H_f	24.29	12.82	7.02
S_1	7.40	15.62	34.21
S_V	21.08	39.95	72.95
L_V	21587.8	40907.6	74702.6

Table 1·3. *A summary of the properties of the three tetra scene descriptions used, given in the format of Table II of Sutherland et al* [1]. *The tetra scenes have only one cluster each and a low overall depth complexity, which causes some of the rules of thumb to result in ridiculous values, e.g.* X_T *and* X_V.

Chapter 2

A Comparison of Five Hidden Surface Elimination Algorithms

2·1. Introduction

This chapter considers serial versions of four common image space hidden surface elimination algorithms. The algorithms are described in detail, their execution costs are roughly estimated, and their performances measured and compared. These results are also compared with those of Sutherland et. al.. The algorithms considered were :

(i) the recursive subdivision algorithm.

(ii) two versions of the scan line algorithm, with and without the edge-table optimisation.

(iii) the z-buffer algorithm.

(iv) the painter's algorithm, (which is actually partly an object space HSE method).

2·2. The HSE Algorithms.

Descriptions of each of the five HSE algorithms studied are given here. Also, their costs are estimated in the style of Sutherland et al [1], though with greater refinement.

2·2·1. Recursive Subdivision Algorithm

This algorithm tries to find a simple solution to the hidden surface problem for a particular area of the screen. Should it fail to do so, it breaks the area up into a number of sub-areas and then applies itself recursively to each of the sub-areas in turn.

First an area of the screen, (initially the entire screen) is considered. Those polygons wholly or partially within this area are identified. Then, if the area has an easily identified shading scheme, (i.e. a simple solution) of one of the four following types, the shading is done immediately. The simple solutions the algorithm recognises, and the actions taken for each one are:

1. There are no relevant polygons.
   - Colour the area with the background colour.
2. There is only one relevant polygon, which is partly or completely enclosed by the area.
   - Colour the area with the background colour, overlaid with the polygon or part of polygon.
3. There is only one relevant polygon, which completely surrounds (encloses) the area.
   - Colour the area with the polygon colour.
4. There is at least one polygon which surrounds the area, and which is in front of all other relevant polygons within this area.
   - Colour the area with the polygon colour.

In the discussion of this algorithm, a *surrounding* polygon is one which completely surround the area of interest, a *surrounded* polygon is one which is completely enclosed by the area and a *crossing* polygon is one which partly overlaps the area. Also, a *relevant* polygon is one which partly or completely encloses the area of interest,

and may extend into neighbouring areas. An *irrelevant* polygon is one which has no overlap with the area.

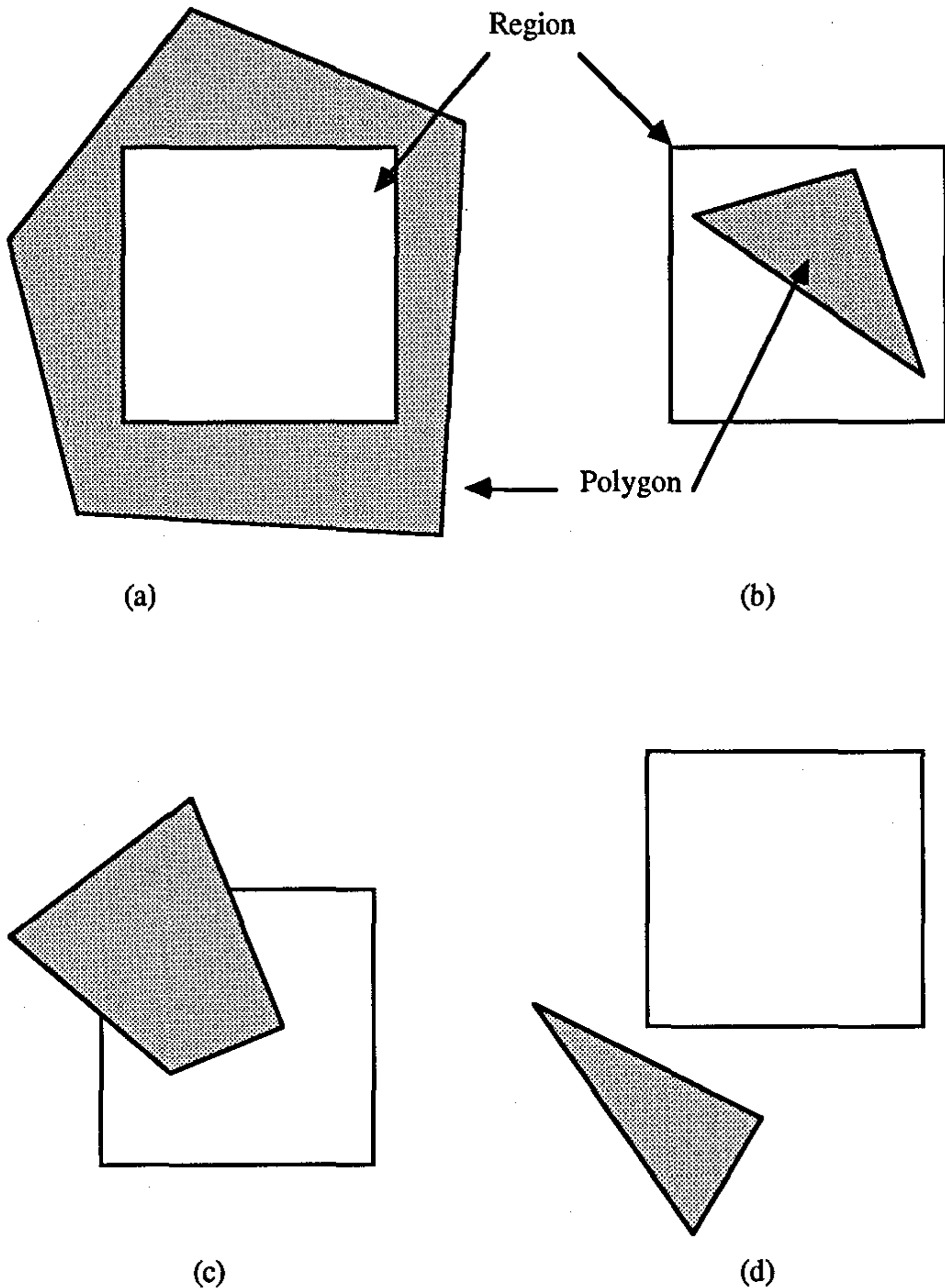


Figure 2·1: *(a) A surrounding polygon, (b) a surrounded polygon, (c) an intersecting polygon, and (d) a disjoint polygon. (a), (b) and (c) are relevant polygons, while (d) is irrelevant.*

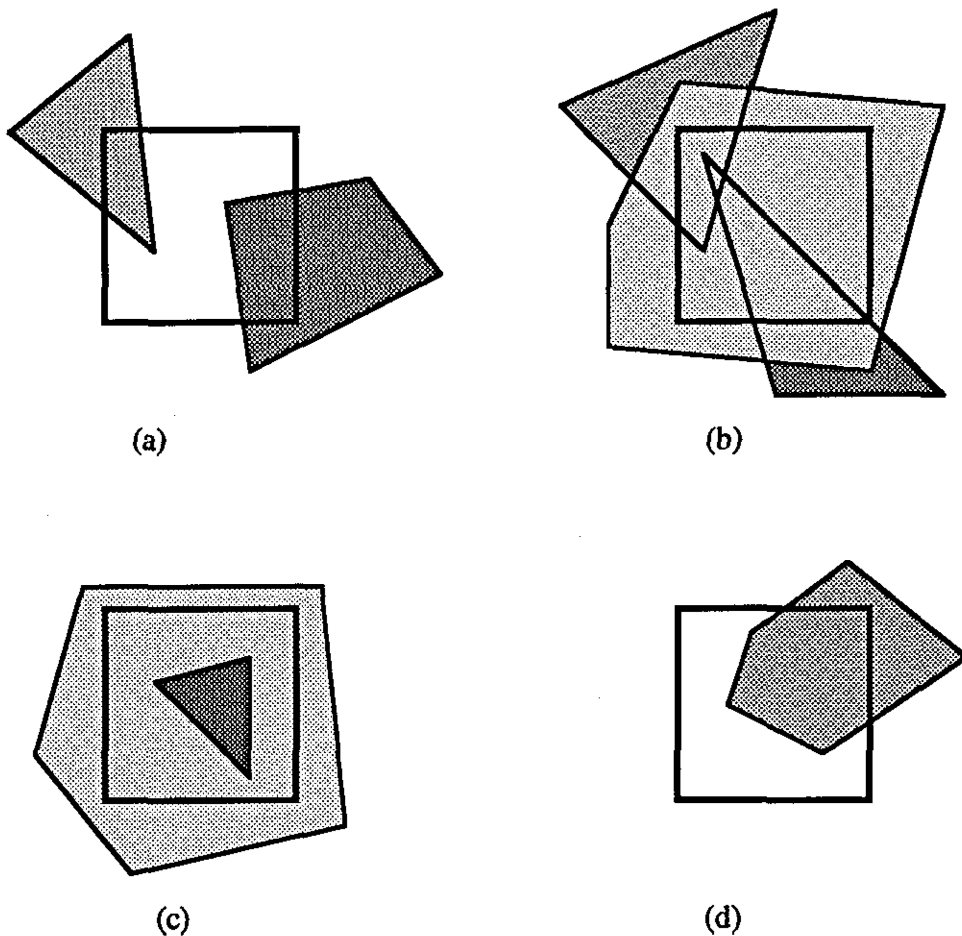


Figure 2·2: *(a) and (c) are examples of situations which cannot be directly handled by the algorithm and must be broken up. (b) and (d) are examples that the algorithm can handle.*

If none of these simple shading solutions is found to apply to the area, then the area is subdivided. In the implementation described in this thesis, the area is subdivided into equal quarters by splitting along the vertical and horizontal halfway points. If the area under consideration is only one pixel in size, then it is not subdivided. Instead a compromise solution, based on the average of the colours of the foremost polygons at the four corners of the pixel, is used.

This technique is applied recursively until the original area has been completely shaded.

The recursive division of the problem may be compared to a "tree" of decisions. The area initially considered, (the outermost area) is the root of the tree. If this is broken into four sub-areas, they may be referred to as intermediate or branch nodes. A sub-area which is shaded rather than being further broken down is considered to be a terminal or leaf node.

This recursive subdivision method is sometimes referred to as a quadtree subdivision method due to the recursive four-branching of its decision tree.

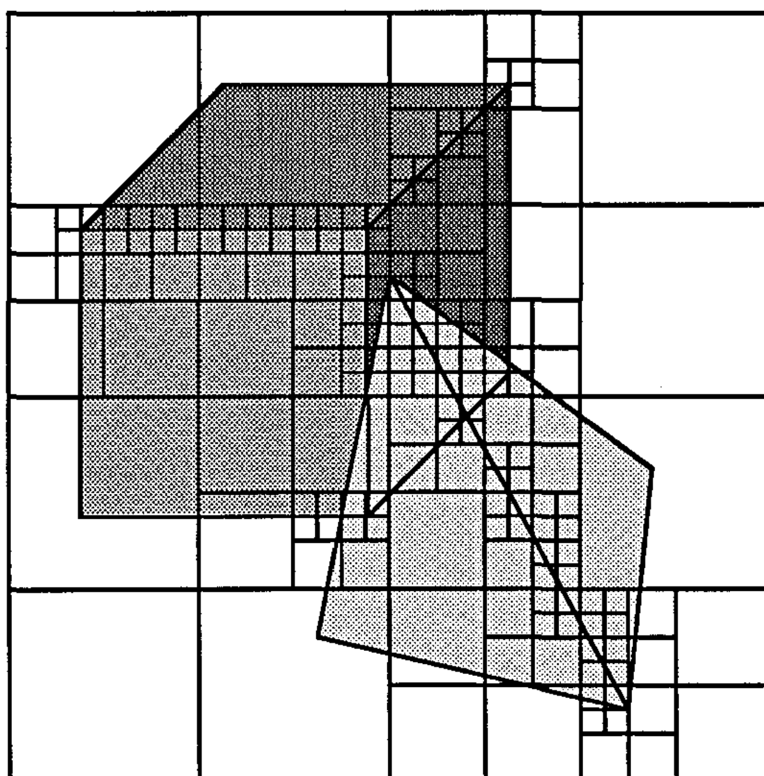


Figure 2·3: *An example of the recursive break up of the problem, to five levels of recursion.*

Of the four HSE algorithms considered here, this was by far the most complicated algorithm to implement. It involves a large number of floating point mathematical

operations and has an extensive control flow structure. This structure is described by the following pseudo-code.

```
PROC recsub (region, list_of_polygons)
  SEQ
    SEQ polygon = 0 FOR all_polygons
      SEQ
        IF
          was_found_to_surround_parent_region (polygon)
            accept_as_surrounding (polygon)
          was_found_disjoint_from_parent_region (polygon)
            reject_as_irrelevant (polygon)
          TRUE
            SEQ
              IF
                -- trivial rejection test to increase performance --
                totally_left_right_above_or_below_region (polygon)
                  reject_as_irrelevant (polygon)
                all_polygon_vertices_within_region (polygon)
                  accept_as_surrounded (polygon)
                any_polygon_edge_crosses_edge_of_region (polygon)
                  accept_as_crossing (polygon)
                odd_num_poly_edges_from_region_to_infinity (polygon)
                  accept_as_surrounding (polygon)
                TRUE
                  reject_as_irrelevant (polygon)
    num_others = num_crossing + num_surrounded
    IF
      (num_surrounding = 0) AND (num_others = 0)
```

```
  paint_region_background_colour ()
(num_surrounding = 0) AND (num_others = 1)
  SEQ
    paint_region_background_colour ()
    paint_relevant_part_of_polygon ()
(num_surrounding = 1) AND (num_others = 0)
  paint_region_polygon_colour ()
(num_surrounding >= 1) AND (one_of_these_hides_all_others ())
  paint_region_polygon_colour ()
TRUE
  SEQ
    IF
      region_is_one_pixel_in_size ()
        paint_pixel_average_colour_of_corners ()
      TRUE
        SEQ
          recsub (top_left_of_region, region's_relevant_polys)
          recsub (top_right_of_region, region's_relevant_polys)
          recsub (btm_left_of_region, region's_relevant_polys)
          recsub (btm_right_of_region, region's_relevant_polys)
```

This algorithm could be altered in many ways. For instance, the subdivision step might be changed to divide an area in half, splitting the area vertically for even levels of subdivision and splitting it horizontally for odd levels of subdivision. Another possible alteration would be to no longer require the resulting parts of a subdivision step to be equal in area, with the division being chosen after considering the relevant polygons. More "solutions" could be recognised in order to avoid unnecessary subdivisions. All

of these changes would offer the possibility of reducing the eventual number of solved areas, but would also introduce extra costs.

The version used here was chosen for its simplicity and familiarity within the computer graphics community. The variations discussed may improve the algorithm's performance somewhat but are unlikely to significantly alter its character.

2·2·1·1. Cost Estimate

To make a useful estimate of the cost of this algorithm, whose actions clearly depend rather heavily upon the exact scene data, several assumptions about reasonable workloads must be made. Even the form of the "worst case" n-polygon scene is not immediately obvious. For instance, one possible "worst case" involves every region being subdivided as far as possible, with the compromise solution being used in all cases. Such a scene implies a fairly well distributed set of polygons. An alternative possible "worst case" scene would involve all polygons clustering into a small region, which involves fewer subdivisions, but increases the cost of testing for each subdivision.

Outermost Level

On first entering the program, all polygons are tested for relevance to the outermost area (which is normally the screen) and are classified as surrounding, surrounded, crossing or irrelevant polygons. Assuming a reasonably well framed object, no rejections will occur during the first pass. Also, there will be few surrounding or crossing polygons. For a scene of P polygons:

- P trivial rejection tests are made. (No polygons are rejected).
- P tests for surrounded polygons are made. (All P polygons are accepted).

• 0 tests are made for crossing polygons.

• 0 tests are made for surrounding polygons

This stage thus will have a cost of O (F_T). This cost is only incurred once for any execution of the algorithm, and so will be negligible compared to the costs of the later stages.

Intermediate Levels

Similarly, for some particular sub-area, whose parent area had P relevant polygons:

• P trivial rejection tests are made. Assuming the polygons were evenly distributed throughout the parent area approximately three quarters of the polygons will be rejected, since the area being considered is a factor of four smaller than its parent area. Very few rejectable polygons will escape this trivial rejection test and need rejecting after testing for acceptance as a surrounded, crossing or surrounding polygon. Such cases are thus ignored here. (3P/4 polygons are rejected). This is O (F_T).

• P/4 tests are made for surrounded polygons. This costs O (F_T).

• A small number of tests are made for crossing polygons.

• A small number tests are made for surrounding polygons.

Although there are likely to be many such branch nodes of the decision tree, they will still be significantly outnumbered by the leaf nodes. Hence the cost of the branch nodes will be swamped by the cost of the leaf nodes.

Terminal Levels

A terminal node will either be of single pixel size and therefore indivisible, or have only one relevant polygon, or have a frontmost surrounding polygon. For the single pixel

case, there will be D_C relevant polygons on average. There will be few occurrences of the single relevant polygon case, since these consume so much more area than single pixel terminal nodes. In the case of a frontmost surrounding polygon the number of relevant polygons will be proportional to D_C, but there will again be few such nodes.

• The number of polygons intersecting an area is on average, approximately D_C for terminal nodes. The culling of irrelevant polygons will thus cost approximately O (D_C).
• The solution for a terminal node that is of single pixel size will cost O (D_C). The solution for a terminal node of greater than single pixel size will also cost O (D_C).

Overall Cost

To draw useful conclusions from this analysis, two extreme cases will be considered. These are (i) a scene consisting of very large polygons, and (ii) a scene consisting of very small polygons.

Only "terminal" areas are considered here since they will significantly outnumber subdivided areas. Of these, single pixel size terminal nodes will also outnumber all other types of terminal nodes, so only these terminal nodes will be considered. Each terminal node costs O (D_C) for both culling and finding a solution.

Case (i):

For this case, single pixel terminal nodes will tend to occur only along visible shared edges in the polygonal scenes. Hence the number of such nodes will be approximately equal to the total length of visible shared edges in the scene.
Total cost is therefore O (D_C * length of visible shared edges). The total number of shared edges will be approximately E_T / 2 since most polygons in the test scenes have immediate neighbours on all sides. The number of visible, shared edges is

therefore E_T / (2 D_C). The average edge length is approximately H_F. Therefore length of visible, shared edges is (E_T H_F) / (2 D_C).

$$\text{Total cost is } O\left(\sqrt{\frac{n\ m\ F_T}{D_C}}\right)$$

Case (ii):

When the scene consists of small polygons, the number of terminal nodes will be approximately O (F_T / D_C) and each such node costs O (D_C).

Thus the total cost is O (F_T).

Scenes consisting of polygons of intermediate size or of different sizes should show costs somewhere between the two extremes considered.

2·2·2. Scan Line Algorithms

Here, the screen is considered as each horizontal line in turn. First the algorithm calculates line segments for each polygon which crosses the current screen line. These segments are essentially horizontal stripes of colour whose descriptions consist of starting coordinates (x1, z1), finishing coordinates (x2, z2), and the polygon's colour. The y-coordinates of the ends of the line segments are implied by the y-coordinate of the current screen line. All of the line segments for the current screen line are placed into a list. The algorithm then resolves any cases of overlapping or intersecting segments within this list. Finally the resulting list of visible segments are sent to the screen processor for display.

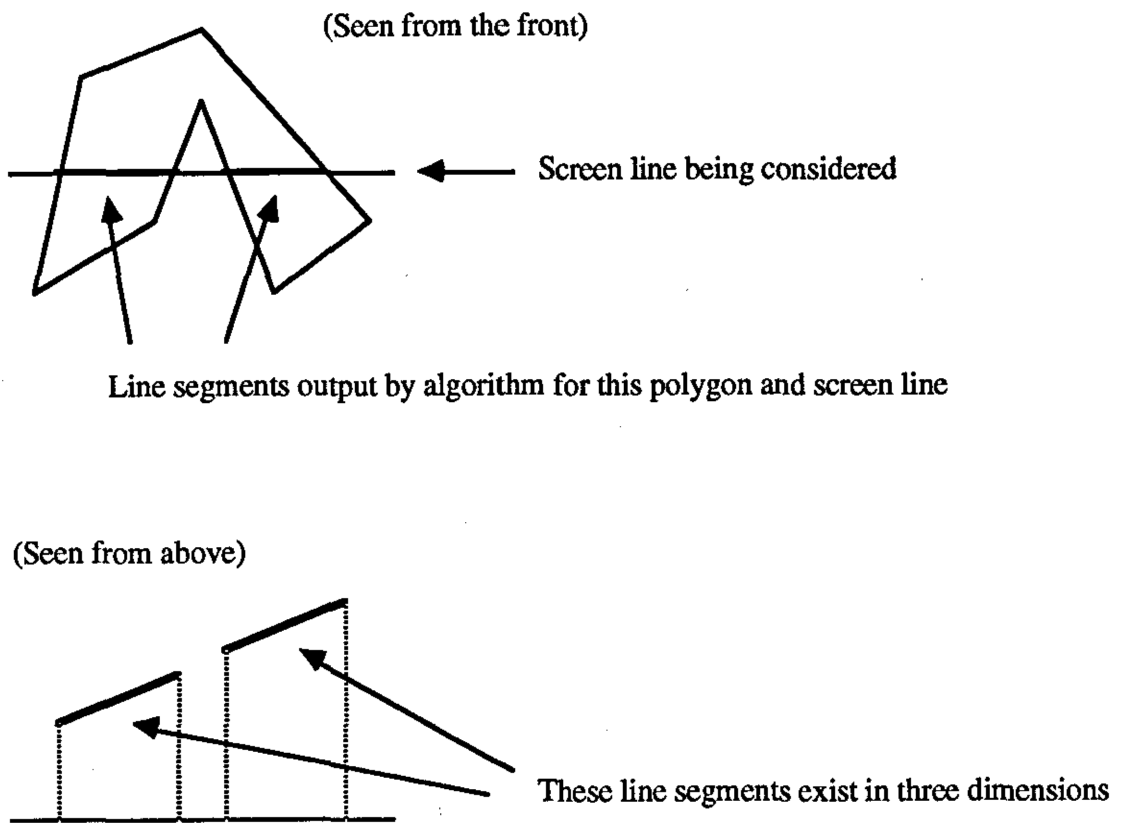


Figure 2·4: *A polygon and its resulting line segments after scan conversion for one particular screen line.*

Two versions of this algorithm were tested. One used edge tables to take advantage of coherence between screen lines within the scene, while the other did not. The edge table optimisation is discussed later. The version without edge tables is described by the first piece of pseudo-code:

```
SEQ
  read_in_polygons ()
  SEQ y = min_y FOR num_screen_lines
    SEQ
      reset (store)
```

```
SEQ polygon = 0 FOR all_polygons
  SEQ
    find_resultant_scan_lines (polygon, y)
    append_scan_lines_to_store (store)
resolve_z_overlaps (store)
output_scan_lines (store)
```

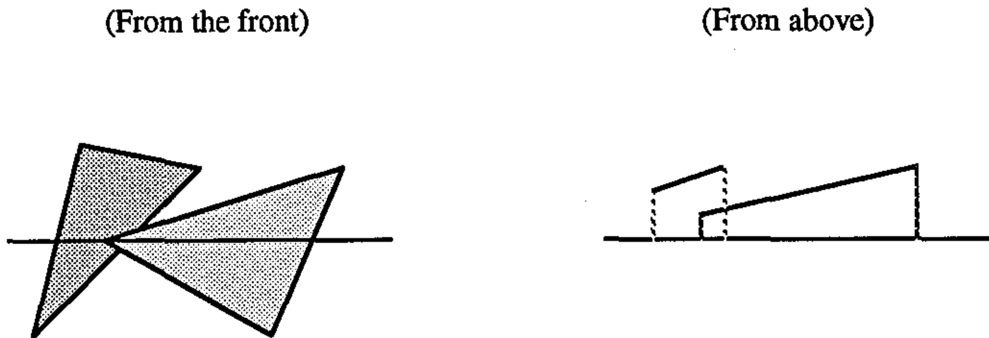


Figure 2·5: *An example of one polygon obscuring another, and the 'overlapping' line segments produced by scan conversion. Correcting these line segments so that only their visible portions are output is the primary job of the scan line algorithm.*

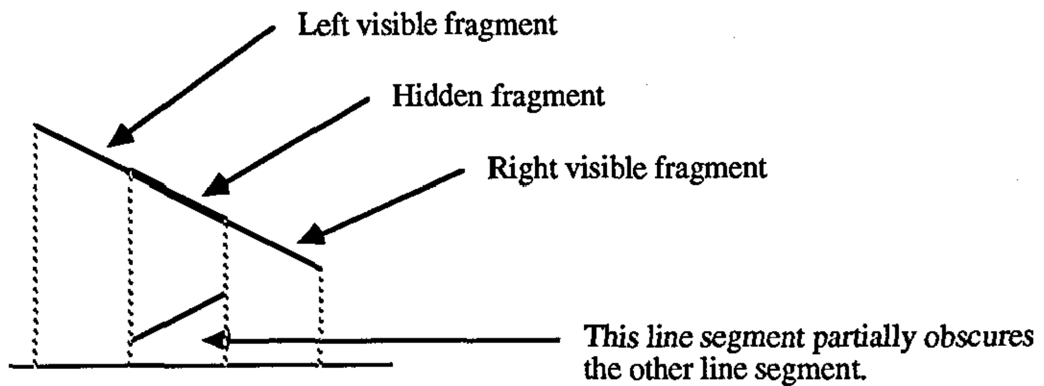


Figure 2·6: *When one line segment hides part of another from view, up to three fragments may result.*

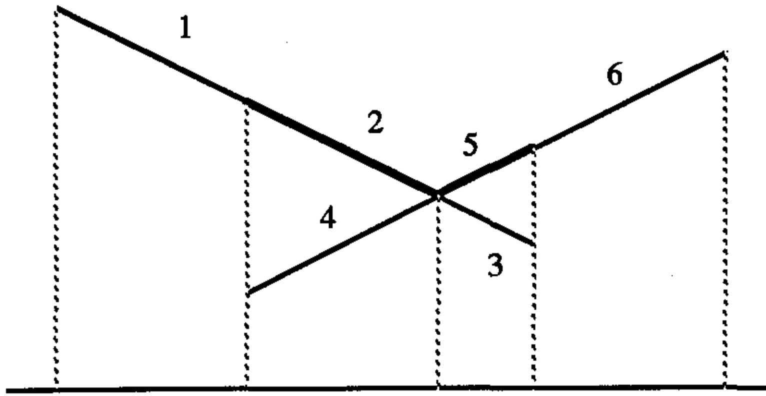


Figure 2·7: *When two line segments intersect, due to their parent polygons intersecting, up to six fragments may result, of which two are always hidden and are therefore discarded. Fragments 2 and 5 are the hidden ones, being obscured by fragments 4 and 3 respectively.*

In the following pseudo-code procedure, the term x-extent appears quite regularly. The x-extent of a three dimensional line segment is the range of possible x values of a point on that line segment. When a partly hidden line segment is broken up by the algorithm, it may consist of up to three pieces. There is the hidden fragment, possibly a visible left end fragment, and possibly a visible right end fragment. The hidden fragment is discarded since it cannot be seen.

```
PROC resolve_x_overlaps (store)
SEQ
  for all possible pairs of scan_lines s1 and s2
    SEQ
      IF
        x-extents_overlap (s1, s2)
          IF
            s1_in_front_of_s2 ()
```

```
SEQ
  IF
    (s2 extends left of s1) AND (s2 extend right of s1)
      SEQ
        replace s2 by left fragment of s2
        add right fragment of s2 to end of segment list
    (s2 extends left of s1)
        replace s2 by left fragment of s2
    (s2 extends right of s1)
        replace s2 by right fragment of s2
    TRUE
      delete s2
s2_in_front_of_s1 ()
  ** similar to previous case **
TRUE
  -- comment: z-extents of s1 and s2 overlap
  IF
    s1 does not intersect s2
      SEQ
        ** Essentially a repeat of previous section.    **
        ** but with more exact in_front_of test.        **
        ** This was a small performance enhancement.    **
    TRUE
      -- comment: s1 intersects s2
      IF
        left_of_s1 in_front_of left_of_s2
          IF
            (s2 extends left of s1) AND
                                  (s1 extends right of s2)
```

```
                SEQ
                  replace s1 by left fragment of s1
                  replace s2 by left fragment of s2
                  append right s1 fragment to segment list
                  append right s2 fragment to segment list
              (s2 extends left of s1)
                SEQ
                  replace s1 by left fragment of s1
                  replace s2 by left fragment of s2
                  append right s2 fragment to segment list
              (s1 extends right of s2)
                SEQ
                  replace s1 by left fragment of s1
                  replace s2 by right fragment of s2
                  append right s1 fragment to segment list
              TRUE
                SEQ
                  replace s1 by left fragment of s1
                  replace s2 by left fragment of s2
          TRUE
            ** similar to above **
  TRUE
    SKIP
```

The next piece of pseudo-code describes the version of the scan line algorithm that uses edge tables to take advantage of coherence within the scene. The rest of this version of the scan line algorithm is identical to the unoptimised version already discussed.

This algorithm begins by constructing two tables, one which records which polygons start on each screen line, and one which records which polygons end on each screen line. The algorithm then keeps a list of currently relevant polygons, which it updates from the two polygon tables as it moves through the screen line by line. This technique was originally applied to tables of polygon edges and so is known as the edge table optimisation. The variety of this technique used here works on tables of polygons due to this being more suited to the particular scan conversion method used. It would thus perhaps be clearer to refer to it as a polygon table optimisation, but the edge table name is far more widely recognised. The difference between polygon tables and edge tables is not significant in terms of algorithm cost.

This modification allows the algorithm to only consider those polygons which are known to be relevant to the current screen line, (i.e. those polygons that will result in scan lines). This requires some pre-processing costs but largely eliminates the costs of examining non-relevant polygons to find whether they are relevant.

```
SEQ
  read_in_polygons ()
  SEQ polygon = FOR all_polygons
    SEQ
      obtain_bottom_and_top_of_each_polygons (bottom, top)
      store_polygon_id (add_polys[bottom])
      store_polygon_id (remove_polys[top])

  find_polygons_intersecting_first_screen_line (current_polygons)

  SEQ y = min_y FOR num_screen_lines
    SEQ
      reset (store)
```

```
SEQ polygon = 0 FOR all_polygons
  SEQ
    find_resultant_scan_lines (polygon, y)
    append_scan_lines_to_store (store)
resolve_z_overlaps (store)
output_scan_lines (store)
remove_expired_polygons (current_polygons, remove_polys[y])
add_newly_relevant_polygons (current_polygons, add_polys[y])
```

2·2·2·1. Cost Estimate for Scan Line Algorithm Without Edge Tables

Part 1 - Initialisation Steps

• The edges are adjusted so that where a polygon's edges touch (at the ends) they do not have identical coordinates, so only one edge occupies a vertex. This is necessary to allow the algorithm to operate correctly. This step considers each edge in turn, and so the cost is O (no. of edges).

• For each edge, a number of properties are calculated and stored. These properties include the minimum and maximum y values, the y height, x and z gradients. For this step the cost is again O (no. of edges).

Part 2 - The Scan Conversion Step

For each screen line, line segments are produced. This involves finding intersections with relevant edges.

• Every edge is checked for relevance at a cost of O (edges).

• Intersections are then found for the relevant ones at a cost of O (relevant edges).

• These intersections are "paired up" using an x-sort step to produce line segments. For each polygon, edge intersections are bubble sorted on their x-values. Assuming the polygons are convex and hence have only two edges crossing a particular screen line, this has a cost of O (polygons). All of the test scenes contain only convex polygons, although the scan line algorithms can handle non-convex polygons.

Part 3 - The HSE Step

As in part 2, this is repeated for each screen line. The line segments created in part 2 are then corrected for overlapping one other (hidden parts are removed). This is similar to a bubble sort on the line segments, except that as overlapping line segments become fragmented the extra fragments are added to the end of the list of items being sorted.

• If no line segments overlap, then each test for overlapping edges is a simple comparison of x values, and there are 0.5 * $segments^2$ tests. This gives a cost of O ($segments^2$). This represents the cost for a scene with a depth complexity of 1 or less at every point of the screen. • In the worst case line segments could be broken up at every comparison, resulting in squaring the number of line segments. Assuming these extra line segments are created at the very beginning of this step, this has a cost of approximately O ($segments^2$ + $segments^4$). Such a scene would be highly unlikely, since every initial segment would be intersecting every other initial segment! • In the more general case, for a scene with a depth complexity D_C, there will be (segments / D_C) sets of D_C overlapping segments at each point of the screen. In most scenes intersecting polygons and therefore intersecting segments are unusual, so most of these cases of overlapping segments will not produce the maximum number of extra segments. A more likely case is the production of one extra segment for every obscured segment. This would result in (segments / D_C) * (D_C - 1) extra segments, or approximately (2 * initial segments) overall, causing approximately (4 * 0.5 * $segments^2$) operations, with a cost of O ($segments^2$).

Part 4 - Painting

The visible segments are then painted with a cost approximately proportional to the number of pixels painted, i.e. O (nm). This step is much less complex than the previous ones, and should contribute negligibly to the overall cost.

Total Cost

• Part 1 cost O (edges) + O (edges) = O (E_t) + O (E_t) = O (E_t)

Reducing this to the basic environment variables using the rules of thumb gives a cost of O (F_T).

• Part 2 cost O (lines * edges) + O (lines * relevant edges) + O (lines * polygons) = O (n E_t) + O $\left(\sqrt{n\ m\ F_T\ D_C}\right)$ + O (n F_t).

Reducing to basic variables gives O (n F_T) + O $\left(\sqrt{n\ m\ F_T\ D_C}\right)$.

• Part 3 cost O (lines * segments^2) = O (n S_l S_l)

Reducing this to basic variables gives O (D_C F_T m).

• Part 4 cost O (nm).

Overall cost is thus O (F_T) + O (n F_T) + O $\left(\sqrt{n\ m\ F_T\ D_C}\right)$ + O (D_C F_T m) + O (nm). All but one of these terms are dependent upon the model size (F_T) and most are directly proportional to it. For many polygons, this algorithm's cost will grow as O (F_T).

2·2·2·2. Cost Estimate for Scan Line Algorithm With Edge Tables (Optimised Scan Line Algorithm)

Part 1 - Initialisation Steps

• The edges are adjusted so that where a polygon's edges touch (at the ends) they do not have identical coordinates, so only one edge occupies a vertex. This is necessary to

allow the algorithm to operate correctly. This step considers each edge in turn, and so the cost is O (no. of edges).

- For each edge, a number of properties are calculated and stored. These properties include the minimum and maximum y values, the y height, x and z gradients. For this step the cost is again O (no. of edges).
- The edge tables (actually polygon tables) are prepared. This costs O (polygons).

Part 2 - The Scan Conversion Step

For each screen line, line segments are produced. This involves finding intersections with relevant edges.

- The list of relevant polygons is updated using the edge tables, at a cost of O (change in relevant polygon set).
- Every edge is checked for relevance at a cost of O (relevant polygons * no. of sides per polygon).
- Intersections are then found for the relevant ones at a cost of O (relevant edges).
- These intersections are 'paired up' using an x-sort step to produce line segments. For each polygon, edge intersections are bubble sorted on their x-values. Assuming the polygons are convex and hence have only two edges crossing a particular screen line, this has a cost of O (polygons * ($0.5 * 2^2$))

Part 3 - The HSE Step

As in part 2, this is repeated for each screen line. The line segments created in part 2 are then corrected for overlapping one other (hidden parts are removed). This is similar to a bubble sort on the line segments, except that as overlapping line segments become fragmented the extra fragments are added to the end of the list of items being sorted.

• If no line segments overlap, then each test for overlapping edges is a simple comparison of x values, and there are 0.5 * $segments^2$ tests. This gives a cost of O ($segments^2$). This represents the cost for a scene with a depth complexity of 1 or less at every point of the screen. • In the worst case line segments could be broken up at every comparison, resulting in squaring the number of line segments. Assuming these extra line segments are created at the very beginning of this step, this has a cost of approximately O ($segments^2$ + $segments^4$). Such a scene would be highly unlikely, since every initial segment would be intersecting every other initial segment! • In the more general case, for a scene with a depth complexity D_C, there will be (segments / D_C) sets of D_C overlapping segments at each point of the screen. In most scenes intersecting polygons and therefore intersecting segments are unusual, so most of these cases of overlapping segments will not produce the maximum number of extra segments. A more likely case is the production of one extra segment for every obscured segment. This would result in (segments / D_C) * (D_C - 1) extra segments, or approximately (2 * initial segments) overall, causing approximately (4 * 0.5 * $segments^2$) operations, with a cost of O ($segments^2$).

Part 4 - Painting

The visible segments are then painted with a cost approximately proportional to the number of pixels painted, i.e. O (nm). This step is much less complex than the previous ones, and should contribute negligibly to the overall cost.

Total Cost

• Part 1 cost O (edges) + O (edges) + O (polygons) = O (E_t) + O (E_t) + O (F_T)
Reducing this to basic variables gives a cost of O (F_T).

• Part 2 cost O (lines * change in relevant polygon set) + O (lines * relevant polygons * 4) + O (lines * relevant edges) + O (lines * polygons)

$= O(F_T) + O\left(\sqrt{F_T\, m\, n\, D_C}\right) + O(n\, F_T)$

• Part 3 cost O (lines * segments^2) = $O(n\, S_l\, S_l)$

Reducing this to basic variables gives $O(D_C\, F_T\, m)$.

• Part 4 cost O (nm).

Overall cost is thus $O(F_T) + O\left(\sqrt{F_T\, m\, n\, D_C}\right) + O(n\, F_T) + O(D_C F_T\, m) + O$ (nm).

These component costs mostly vary with model size (F_T) with powers of 0.5 to 1.0. For large numbers of polygons, the overall cost is $O(F_T)$.

2·2·3. Z-Buffer Algorithm

This is the simplest hidden surface algorithm tested. Basically, every point of every polygon is plotted into a z-buffer. Every pixel in a z-buffer consists of a storage location for that pixel's displayed colour and also a storage location for the z-value of the pixel. When plotting into a z-buffer the algorithm must check to see if the point being plotted is behind the one already in the z-buffer, in which case nothing is done, or is in front of the pixel already in the z-buffer in which case the old z and colour values are overwritten. Z-buffers are frequently supported in hardware due to their simplicity, (and hence low cost). The implementation described here used a simple scan conversion algorithm to calculate the points to be plotted.

The pseudo-code routine `z_plot` describes the method of plotting into a z-buffer. The colour values for screen points are stored in the array `screen` and the corresponding z-values are stored in the array `z_value`. These two arrays together form the z-buffer. The three dimensional, coloured point described by `x`, `y`, `z`, and `colour` is plotted into this z-buffer.

```
PROC z_plot (x, y, z, colour)
SEQ
  IF
    z < z_value[x][y]
      SEQ
        -- comment: The point is in front of whatever is already in
        -- comment: the z-buffer at these x and y coordinates.
        z_value[x][y] = z
        colour[x][y] = colour
    TRUE
      -- comment: The point is hidden, so do not draw it.
      SKIP
:
```

The program processes one polygon at a time. It simultaneously works its way up the left and right sides of the polygon, interpolating coordinates between vertices. This scan conversion algorithm is slightly different to that used in the scan line algorithms, but its costs are almost identical for convex polygons (which are the only sort present in the test data). The resulting scan line segments are then drawn into the z-buffer one pixel at a time, using the method described previously.

2·2·3·1. Cost Estimate

Each polygon is converted to line segments in turn.

• A preprocessing step builds a list of edges on the left side of the polygon and a list of right side edges. This costs O (E_T).

• The scan conversion steps through the y-range of each polygon one screen line at a time. This costs $O(H_f)$ for each polygon.

• Each pixel in a line segment is tested against the z-buffer at a cost of O (segment length) per segment. Visible pixels are then painted into the z-buffer at a cost of O (visible pixels).

Total Cost

The cost is $O(E_T) + O(F_T H_f) + O(\text{segment length} * \text{segments}) + O(\text{visible pixels})$

$= O(E_T) + O(F_T H_f) + O(n\, m\, D_C) + O(n\, m)$

$= O(F_T) + O\left(\sqrt{n\, m\, D_C\, F_T}\right) + O(n\, m\, D_C) + O(n\, m)$

This has a fixed cost of $O(n\, m)$. The term $O(n\, m\, D_C)$ is also effectively a fixed cost for each set of test data used in this work because D_C is approximately the same for each member of a particular set of test scenes.

The first two cost terms grow with the size of the test scene, so for large numbers of polygons the cost of this algorithm would be $O(F_T)$. However, if the polygons are of multiple pixel area then these two costs are swamped by the per-pixel cost since they occur only once per polygon (for the first term) or once per segment (for the second term).

The overall cost for large or medium size polygons will therefore tend to be $O(n\, m)$. For large numbers of very small polygons, the overall cost would be controlled by the $O(F_T)$ term.

2·2·4. Painter's Algorithm

Unlike the previous three algorithms, the implementation of this algorithm does not create correct hidden surface images for scenes containing penetrating or interleaving

polygons. This algorithm first sorts the polygons by z order and then scan converts them in (back to front) order onto the screen. The sorting technique used is a bucket sort, with 2000 buckets. This sorting technique was chosen because it offers well controlled costs for sorting large numbers of items [46]. For example, a simple bubble sort costs O (n^2) for n items, a quicksort costs O (n log n) on average and O (n^2) in the worst case, a heapsort costs O (n log n) and a bucket sort (sometimes known as bin sort) costs O (n + m) where m is the number of buckets.

Sorting Algorithm	Cost	Cost for 2500 item sort vs. cost for 100 item sort
Bubble sort	O (n^2)	625
Quicksort (average)	O (n log n)	42
Quicksort (worst case)	O (n^2)	625
Heapsort	O (n log n)	42
Bucket sort, 100 buckets	O (n + buckets)	13
Bucket sort,2000 buckets	O (n + buckets)	2.1

Table 2·1. *Sorting cost variation with number of items to be sorted, for several common sorting algorithms.*

As may be seen in table 2·1, the cost of a bucket sort rises far more slowly with the number of items to be sorted than the costs of the other sort algorithms. The bucket sort is a relatively simple algorithm, with each of its steps being of low absolute cost. The choice of the number of buckets used (2000) was made to limit the sorting cost for large numbers of polygons in preference to limiting the cost of sorting small numbers of polygons. The table also includes a 100-bucket sort to illustrate this point. The 100-bucket sort limits the absolute cost of a 100 item sort in preference to limiting the

growth of sorting cost for the range of quantities of items sorted, (which was 200 to 2500 polygons for this work).

Since the polygon sort is carried out in object space this algorithm is partly an object space HSE algorithm. The remainder of the algorithm however operate in image space. The structure of this algorithm is illustrated by the pseudo-code:

```
SEQ
  read_in_polygons ()
  find_and_store_average_z_of_polygon's_vertices ()
  bucket_sort_polygons ()
  scan_convert_polygons_from_back_to_front ()
```

This algorithm has one potentially adjustable factor - the number of buckets used for the bucket sort. This could possibly be changed to improve the performance of the sort in terms of execution time for a particular scene, or in terms of the sorting time's dependence upon model size. Two thousand buckets were used for the work described here since the aim was to limit the growth of cost with model size, and this number is close to the largest number of polygons handled. The sort costs O (buckets + polygons) and so for small scenes the cost is O (buckets), and for large scenes the cost is approximately O (buckets + polygons). This avoids the cost of the sort growing by more than a factor of two or so.

2·2·4·1. Cost Estimate

• In a preprocessing step, the average z value of each polygon is found.

This costs O (polygon * 4).

• The polygons are then sorted on their average z values using a bucket sort.

This costs O (buckets + polygons).

• Each polygon is then scan converted. This costs O (H_f) for each polygon.

• The pixels are painted. This costs O (nmD_C).

Total Cost

The total cost is O (edges) + O (buckets + polygons) + O (H_f * polygons) + O (nmD_C)

= O (E_T) + O (buckets + F_T) + O ($H_F F_T$) + O (nmD_C).

This reduces to O (F_T) + O (buckets + F_T) + O $\left(\sqrt{n\ m\ D_C\ F_T}\right)$ + O (nmD_C).

Thus this algorithm's cost for large numbers of polygons is O (F_T).

2·3. Timing Information

The fast, on-chip RAM was not used. This decision was made because the on-chip RAM is limited in size to 4K bytes, and hence has an effect upon the execution speeds of programs of different sizes, since differing proportions of such programs fit in this high speed RAM. Instead, all program and data were stored instead in the slower, expandable, external RAM.

In all cases, the clock was started after the polygons had been loaded into memory from disk. This was done to avoid attributing a cost to the algorithms for which they are not responsible. The timing of the optimised scan line algorithm includes the creation of the edge tables. All timings were taken using the transputer's low priority clock, which ticks 15625 times per second.

2·4. Results

For each of the five HSE algorithms previously described, timings were taken for the solution of the hidden surface problem for each of the five teapot scenes and three tetra scenes discussed.

2·4·1. Recursive Subdivision Algorithm

Figures 2·8 and 2·9 show an interesting compound behaviour. For small numbers of large polygons, the execution time grows as some fractional power of the model size. As the model size increases and the polygons decrease in size, (since the depth complexity is held constant for the test scenes), the behaviour alters to a linear growth of execution time with model size. This strongly supports the cost analyses for this algorithm which suggested a square root growth of execution time for small numbers of polygons and a linear growth for larger numbers of smaller polygons. The execution times are tabulated in Table 2·2.

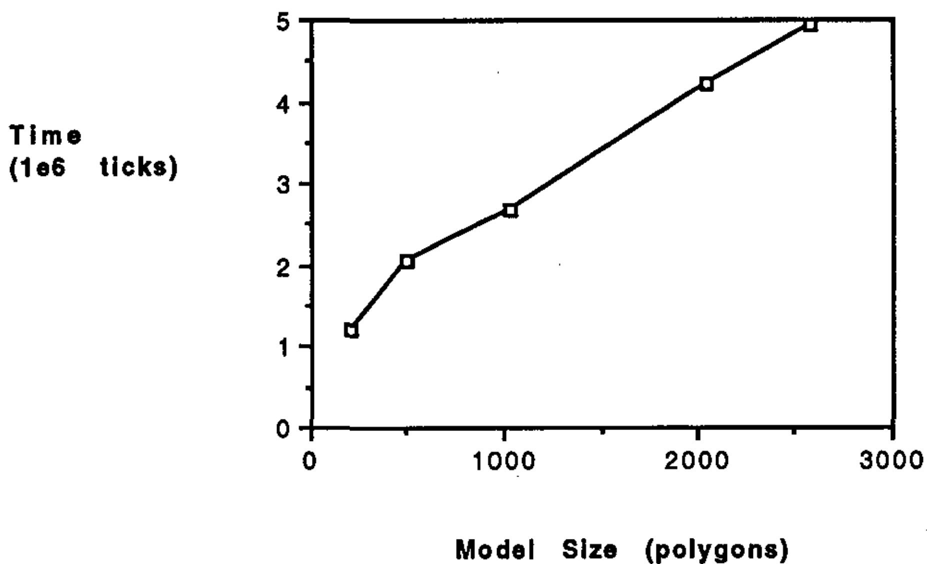


Figure 2·8: *Execution time versus model size for the recursive subdivision algorithm and teapot models.*

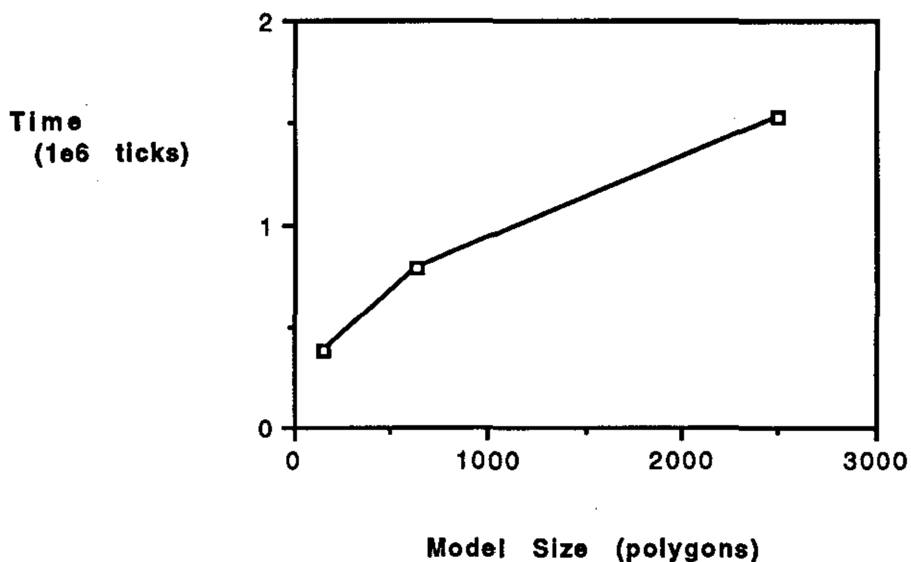


Figure 2·9: *Execution time versus model size for the recursive subdivision algorithm and tetra models.*

Model Type and Size	Execution Time (in ticks)
Teapot 205	1222352
Teapot 499	2055422
Teapot 1027	2672128
Teapot 2035	4210430
Teapot 2575	4947562
Tetra 156	385115
Tetra 624	784343
Tetra 2496	1528651

Table 2·2: *Execution times for the recursive subdivision algorithm.*

2·4·2. Scan Line Algorithm (Unoptimised)

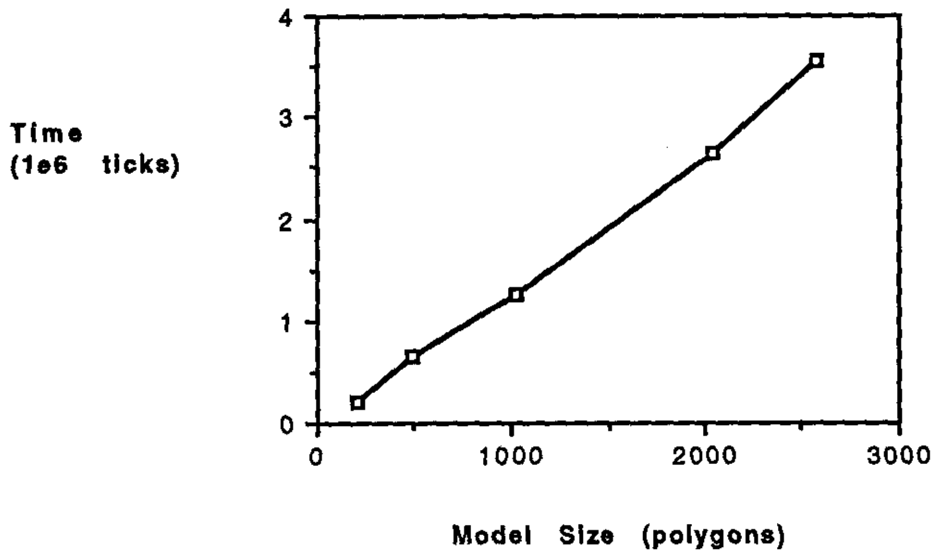


Figure 2·10: *Execution time versus model size for the unoptimised scan line algorithm and teapot models.*

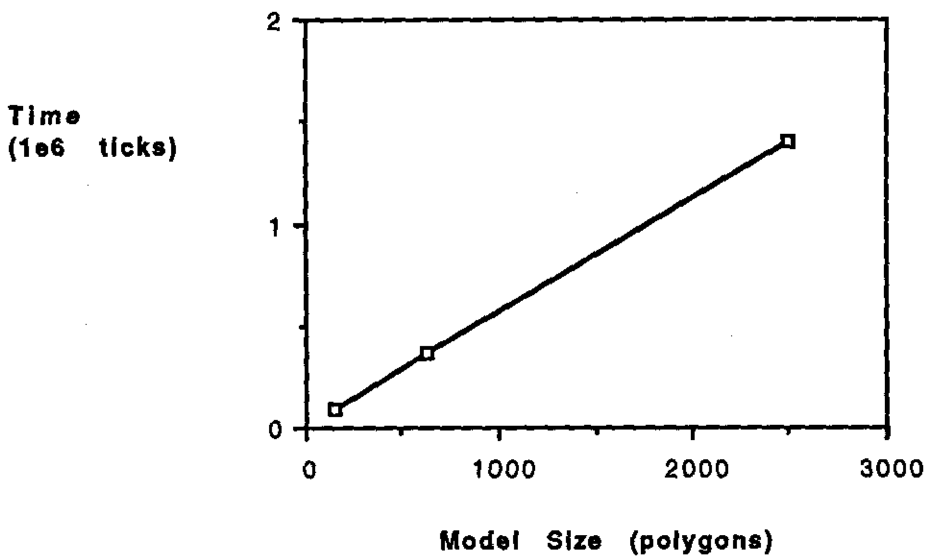


Figure 2·11: *Execution time versus model size for the unoptimised scan line algorithm and tetra models.*

Figures 2·10 and 2·11 show that the execution time of the algorithm is roughly proportional to the model size. This is supported by the cost analysis which concluded that the cost of the algorithm was proportional to the model size, (with small fixed costs etc.).

Model Type and Size	Execution Time (in ticks)
Teapot 205	208212
Teapot 499	652933
Teapot 1027	1255844
Teapot 2035	2651283
Teapot 2575	3559407
Tetra 156	96931
Tetra 624	363791
Tetra 2496	1403996

Table 2·3: *Execution times for the unoptimised scan line algorithm.*

2·4·3. Optimised Scan Line Algorithm

For this algorithm, Figures 2·12 and 2·13 suggest that execution time is approximately proportional to model size. This is supported by the cost analysis which noted many steps of linear cost. Comparing the two figures for small models shows some difference in execution costs between the two models, presumably due to a dependence upon depth complexity - the major difference between the two set of models. This implies that the $O(D_C F_T m)$ term is a major component of the total cost, which is reasonable since this is the HSE step.

Comparing the unoptimised scan line algorithm results (figures 2·10 and 2·11) shows a much smaller dependence on depth complexity, and implies that the HSE step is a

smaller part of the total cost than for the optimised scan line algorithm. The optimised version may thus be considered as being more "focussed" on the HSE job rather than on the "book-keeping" jobs which support it.

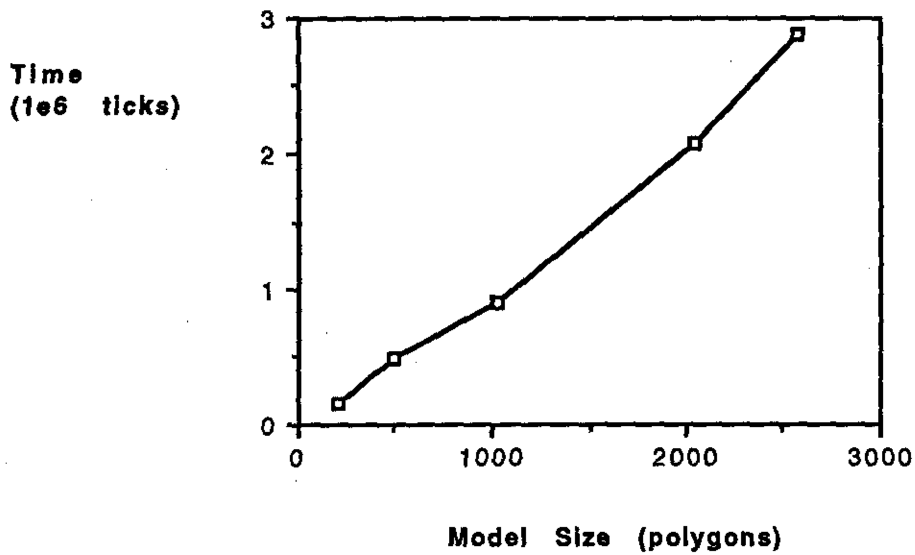


Figure 2·12: *Execution time versus model size for the optimised scan line algorithm and teapot models.*

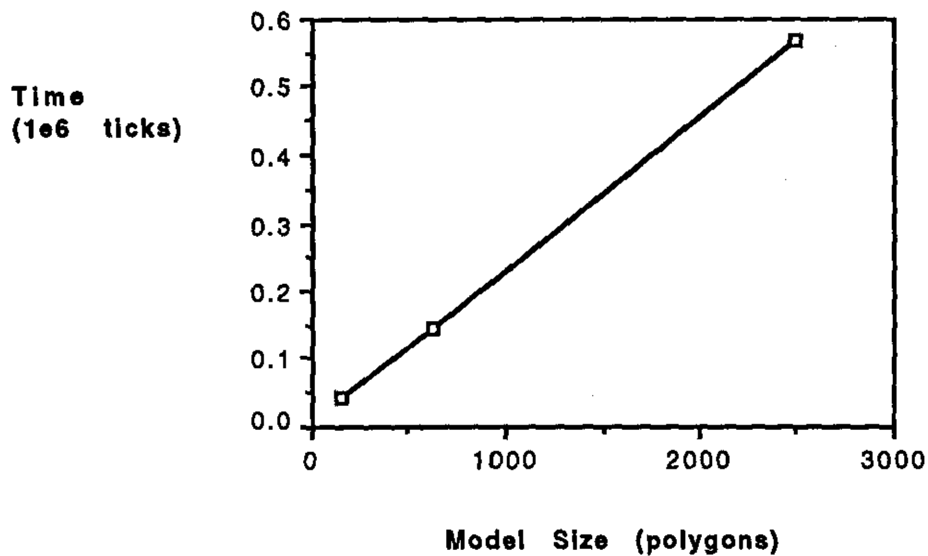


Figure 2·13: *Execution time versus model size for the optimised scan line algorithm and tetra models.*

Model Type and Size	Execution Time (in ticks)
Teapot 205	150692
Teapot 499	483806
Teapot 1027	897526
Teapot 2035	2069488
Teapot 2575	2880572
Tetra 156	44162
Tetra 624	146107
Tetra 2496	568946

Table 2·4: *Execution times for the optimised scan line algorithm.*

2·4·4. Z-Buffer Algorithm

Figures 2·14 and 2·15 show that this algorithm has an execution time which shows a small growth in proportion to the model size, with a large fixed cost. This is supported by the cost analysis which found both a linear dependence on the model size and steps whose costs depend on the total number of pixels in the scene but not on the model size. (The number of pixels in the scene is almost independent of model size for each set of scenes).

Extrapolating the results graphs toward low numbers of polygons gives an estimate of the fixed (i.e. non number-of-polygon dependent) costs. The teapot results extrapolate to a y-axis intersection of about 500000 ticks and the tetra results to about 60000 ticks. This factor of ten difference shows the relative importance of the two fixed cost terms of the cost analysis. These fixed costs were the O (nmD_C) term due to testing points against the z-buffer and the O (nm) term due to actually painting a point into the z-

buffer. Since the difference between the teapot and tetra data sets was mostly a factor of ten difference in D_C, it can be seen that the O (nmD_C) term dominates the fixed costs. As already discussed, the fixed costs in turn dominate the cost of the algorithm.

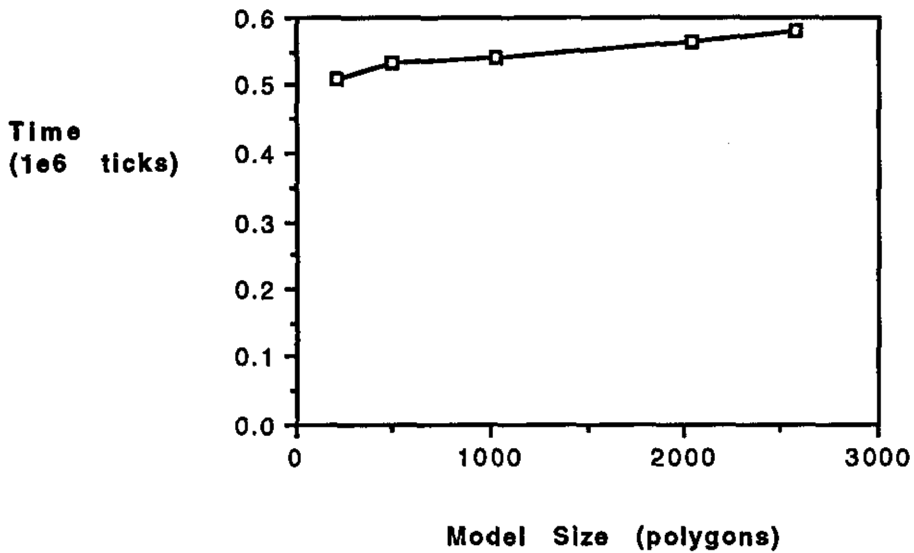


Figure 2·14: *Execution time versus model size for the z-buffer algorithm and teapot models.*

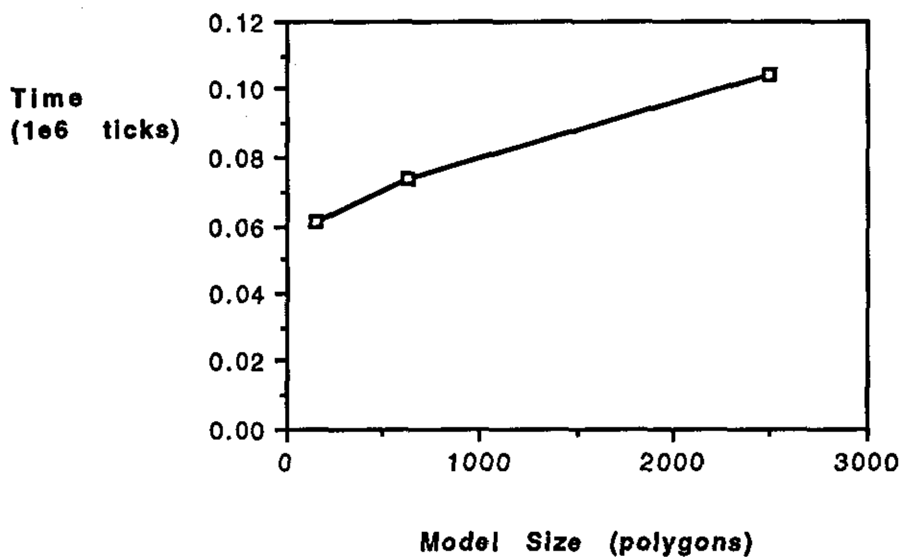


Figure 2·15: *Execution time versus model size for the z-buffer algorithm and tetra models.*

Model Type and Size	Execution Time (in ticks)
Teapot 205	509952
Teapot 499	531721
Teapot 1027	539235
Teapot 2035	565572
Teapot 2575	581151
Tetra 156	60846
Tetra 624	73994
Tetra 2496	104037

Table 2·5: *Execution times for the z-buffer algorithm.*

2·4·5. Painter's Algorithm

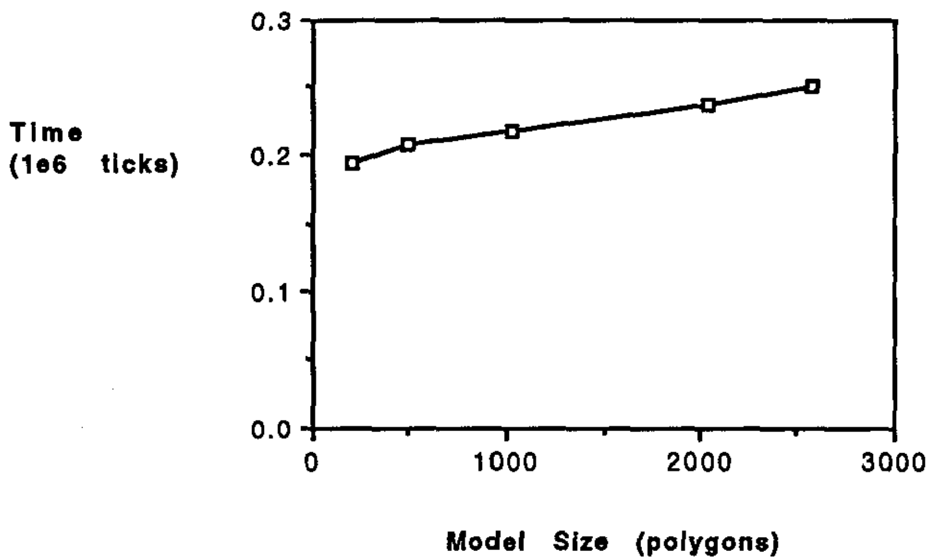


Figure 2·16: *Execution time versus model size for the painter's algorithm and teapot models.*

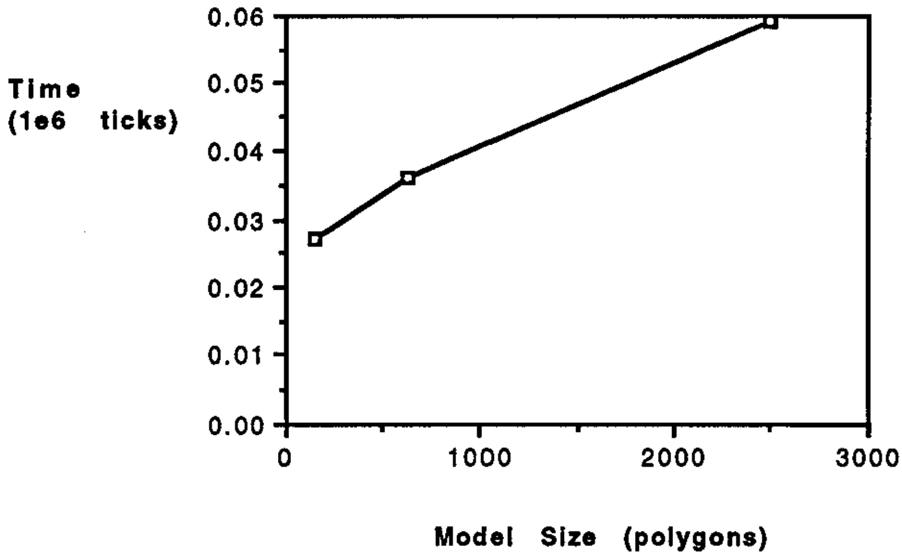


Figure 2·17: *Execution time versus model size for the painter's algorithm and tetra models.*

In figures 2·16 and 2·17, this algorithm shows a linear dependence upon model size, with some apparently fixed overhead costs. This is supported by the cost analysis which found several separate costs for this algorithm including a linear dependence upon model size and a linear dependence upon the number of pixels in the scene. With the number of pixels being constant for each scene in a given set, the dependence on the number of pixels forms the fixed cost component.

Extrapolating the graphs to the y-axis to give the fixed costs shows an approximately factor of ten difference in the size of the fixed costs between the teapot and tetra data sets. Since the difference between these data sets is mostly a factor of ten difference in D_C, it can be seen that the O (nmD_C) term dominates the costs of the painter's algorithm.

Model Type and Size	Execution Time (in ticks)
Teapot 205	194052
Teapot 499	207781
Teapot 1027	216999
Teapot 2035	238060
Teapot 2575	250232
Tetra 156	27166
Tetra 624	36167
Tetra 2496	59161

Table 2·6: *Execution times for the painter's algorithm.*

2·5. Comparison with Sutherland et. al.

A landmark in the discussion of polygon processing algorithms was the survey paper of Sutherland, Sproull, and Schumacker [1]. They produced a table showing the relative cost of several algorithms for various numbers of polygons. A reduced version, taken from Foley and van Dam [47] is shown below, (Table 2·7). This table was based upon the estimated costs of executing the various algorithms. The author's results are also shown for purposes of comparison, (Table 2·8).

Relative Cost		
	Model Size (polygons)	
Algorithm	100	2500
Painter's	1	10
Z-Buffer	54	54
Scan Line (with Edge Tables)	5	21
Recursive Subdivision	11	64

Table 2·7: *Estimated relative costs of the algorithms, (relative to the 100 polygon, painter's algorithm case).*

Measured Relative Cost		
	Model Size (polygons)	
Algorithm	200	2500
Painter's	1.0	1.3
Z-Buffer	2.6	3.0
Scan Line (with Edge Tables)	0.8	14.8
Scan Line	1.1	18.3
Recursive Subdivision	6.3	25.5

Table 2·8: *Measured relative performance of the algorithms, (relative to the 200 polygon, painter's algorithm case).*

The algorithms' relative performance varies significantly with the number of polygons in the scene description. While the z-buffer algorithm is largely independent of model size and the painter's algorithm is only slightly more dependent on model size, the

growth of the recursive subdivision algorithm's cost is somewhere between square root and linear growth, depending upon the sizes of the polygons in the model. The unoptimised scan conversion algorithm has a linear increase in cost as the model size increases, as does the edge-table optimised version.

These differences mean that while for 205 polygons the scan line algorithm using edge tables is faster than the z-buffer, it actually becomes far worse for 2575 polygons. By comparison with the other scan line algorithm, the edge table version suffers as the model size rises.

Extrapolating the relative performance trends toward larger model sizes suggests that the recursive subdivision algorithm could become cheaper, or at least no more expensive than either of the scan line algorithms. Also, the z-buffer will probably cost less than the painter's algorithm for large model sizes. The latter two algorithms are much faster than either of the former.

Table 2·7 was created only as an order of magnitude guide [1]. The differences between that table and the author's, (Table 2·8), are interesting. The relative performances for small numbers of polygons appear comparable except for the large cost of the z-buffer in Table 2·8. This appears to be largely due to Sutherland et al [1] giving a high estimate for the costs of the operations involved in the z-buffer compared to those of the other algorithms.

The changes due to increased model size also appear comparable except for the painter's algorithm case where the sorting algorithm considered for this case in Sutherland et al [1] was more expensive than the bucket sort used for this case in this work.

The absolute performance of the implemented algorithms is good for a software graphics system, but compared to a hardware graphics system it is not as impressive. This is not greatly surprising since the implementations discussed have not been optimised significantly. There is still much room for improvement, particularly in the routines which paint into the frame buffer.

2·6 Conclusions

The algorithms' dependence on model size may be largely attributed to a combination of the sorting techniques used and some overhead costs. The sorting in the z-buffer algorithm is entirely buried in the painting process; the sorting in the painter's algorithm is a low rate of growth bucket sort giving a small dependence upon linear growth; the sorting in the recursive subdivision algorithm is both area and polygon size dependent and hence a hybrid of square-root and linear behaviour; and finally the scan line algorithms depend upon collections of linear growth techniques. This conclusion is much the same as that of Sutherland et al [1,48].

Also of note is that several of the HSE algorithms showed significant dependence upon depth complexity. This dependence corresponded for each algorithm to the cost term derived from the major HSE step. The dependence is also more noticable for the faster algorithms. This suggests a correlation between an algorithm's performance and its focus on a single, major HSE step.

For almost any HSE job where the output is to appear on a pixel type display, the z-buffer algorithm gives a very good compromise solution with little dependence upon model size. It is also almost as fast as and is more exact than the fastest HSE algorithm tested, the painter's algorithm.

Although all of the algorithms discussed here only operate upon scenes consisting of flat shaded polygons, extending them to handle some simple smooth shading scheme (such as Gouraud shading) would be a minor modification. This would not significantly alter the costs of the algorithms and so these conclusions should also apply to systems which handle simply shaded polygon scenes.

Chapter 3

A Comparison of Five Parallel Hidden Surface Elimination Algorithms

3·1. Introduction

This chapter considers parallel implementation of five hidden surface elimination algorithms. The algorithms are those discussed in their classical, serial versions in the previous chapter. They are:

i) recursive subdivision algorithm
ii) two hidden scan line algorithms (with and without the edge table optimisation)
iii) z-buffer algorithm
iv) painter's algorithm

Each of these algorithms has been modified to allow its use on a parallel computer of the distributed memory multiprocessor type. The cost of the modified algorithms has been estimated. These parallel implementations have also been tested on a moderately large multiprocessor to test how well they actually perform for various sizes of the scene description and numbers of processors used. Limiting factors are discussed. The estimated and actual costs have been compared.

3·2. The Parallelisations of the Algorithms

In this section, each of the modified algorithms is discussed in turn. The costs of these algorithms are estimated as for the serial cases, in the style of Sutherland et al [1], but with greater refinement.

3·2·1. Recursive Subdivision Algorithm

The original, serial recursive subdivision algorithm has a structure which lends itself to being distributed across a number of processors. Since the algorithm ordinarily breaks the screen area up into smaller parts for solution, the algorithm is easily parallelised by giving each processor a subdivision of the screen to work on.

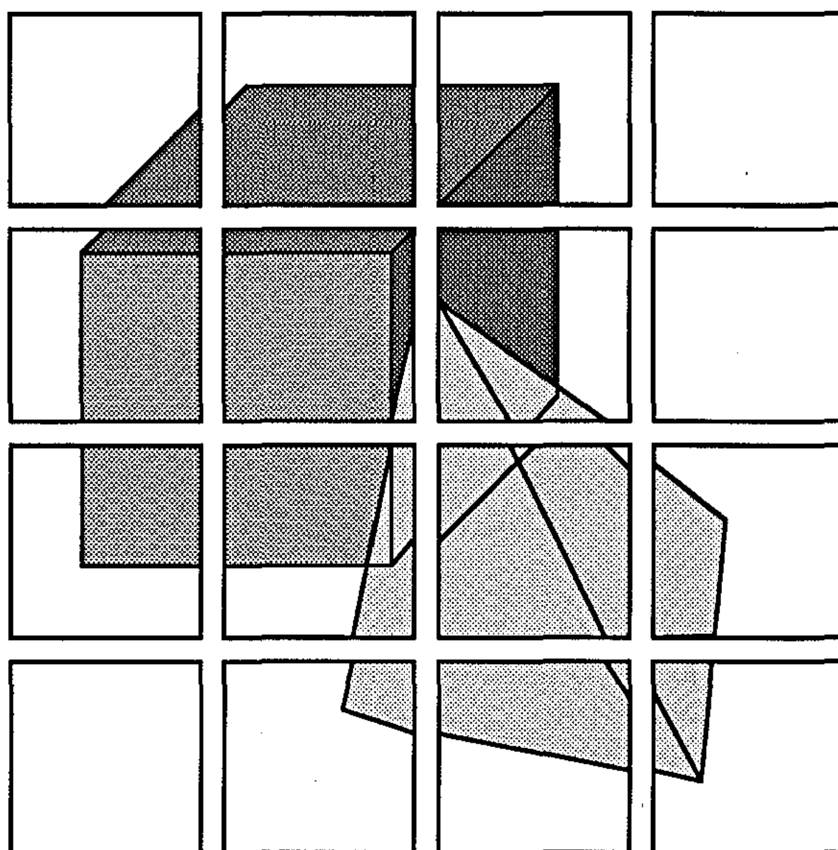


Figure 3·1: *For the sixteen processor case the screen is broken into sixteen parts, as shown, and one part is assigned to each processor for solution.*

Due to the subdivision step of this implementation dividing an area into four parts, this implementation has only been tested for (a) one worker processor, which considers the entire screen area, (b) four worker processors, each of which considers a quarter of the screen area, (c) sixteen worker processors, each of which considers a sixteenth of the screen area and (d) sixty-four worker processors, each of which considers a sixty-fourth of the screen area. In the general case the algorithm can handle 4^n processors, n = 0, 1, 2, 3, etc..

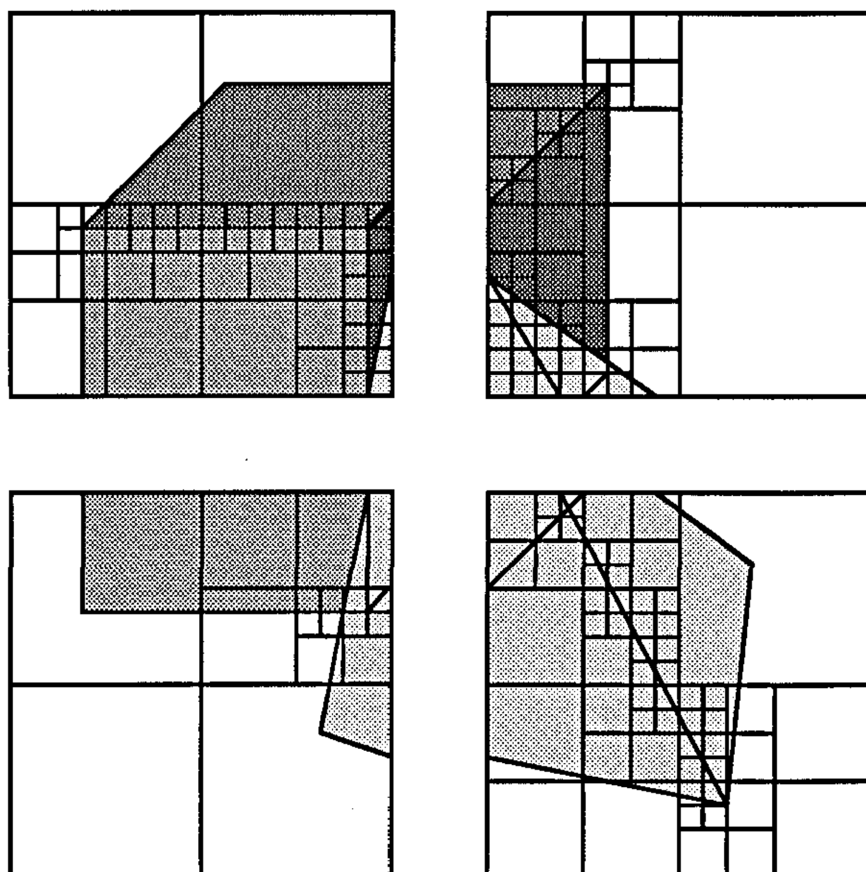


Figure 3·2: *The way each area is subdivided, for the four processor case. Some parts are subdivided far more often than others. This can lead to bad load balancing.*

Using this method of breaking the problem up into enough pieces for a large number of processors, the resulting parallelised recursive subdivision algorithm is almost identical

to the original serial version. On each processor a copy of the serial recursive subdivision program is run, but instead of always starting by considering the full screen area the program instead considers a fraction of the full screen area. This is equivalent to starting after a number of levels of subdivision.

There is one major potential problem with this simple approach to parallelising the algorithm - that of load balancing. As can be seen in figure 3·1, the different areas of the screen will contain images of varying complexity. In the worst case some areas will hold no polygons at all, while others have many polygons forming a complicated HSE problem. This property of the image becomes a problem because each area of the screen is assigned to one particular processor, so some processors may quickly solve their simple areas while others have only just begun solving their complicated areas.

Bad load balancing may be avoided to some extent by allowing processors that have completed their areas to take over the solution of parts of the more complicated areas. Such a scheme of redistributing work imposes certain extra costs. The processor that releases part of its work has to know when other processors are free. There are communication costs in passing descriptions of unsolved areas between processors. The receiving processor must already contain all polygons relevant to the transferred area, which it may not necessarily do so if the previous stages in the graphics pipeline only passed along those polygons relevant to each processor's initial area. The receiving processor must also either carry out an expensive initial cull from all the polygons it "knows", or a list of polygons in the transferred area's parent area must be communicated.

This redistribution technique was not implemented for this work because it would have taken considerable extra work to implement, while if the load balancing problem was considered, the character of the algorithm could still be ascertained without it.

The basic recursive subdivision algorithm is "wrapped" in an extra layer of program to handle communication with other processors. This wrapper program receives the original scene description and information describing the fraction of the screen area to be considered. The multiprocessor used for this work consisted of a master processor with an attached chain of up to one hundred and twenty eight processors, as discussed elsewhere. The structure of the parallel program naturally reflects this machine structure.

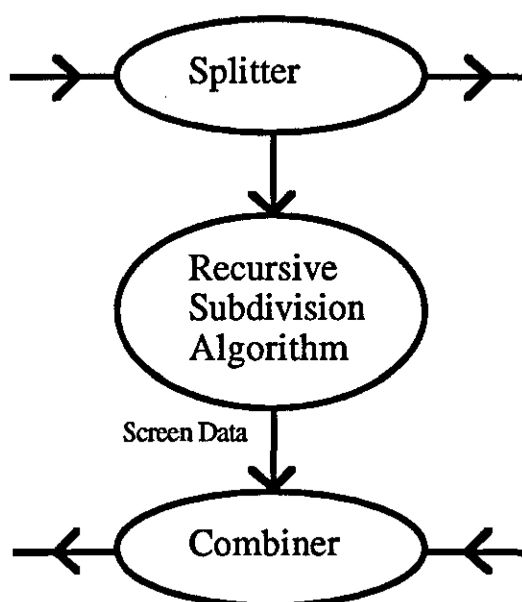


Figure 3·3: *The processes running on each worker node for the recursive subdivision algorithm.*

In operation, the program on the master processor passes the polygon data to the worker processors, and also passes any returned screen painting information to the screen processor. Each worker processor therefore runs three processes in pseudo-parallel - as shown in Figure 3·3 - one for handling and passing along data from the master processor, one that actually runs the implemented recursive subdivision

algorithm, and finally one that handles the transmission and passing along of control signals back to the master.

Since the recursive subdivision algorithm used in the parallel implementation is essentially identical to that used in the serial implementation described in the previous chapter, it is not redescribed here. The structure of the program on each worker node is described by the following pseudo code:

```
PAR
  splitter()
  recursive_subdivision()
  combiner()
:
```

The `splitter` process is described in the next piece of pseudo code. It simply forwards data to the relevant destination. The polygons of the scene description are passed to both the local HSE process and the next processor in the chain of worker processors. Messages telling a particular processor which area of the image space it is to solve the HSE problem for are either:

(a) forwarded along the processor chain for messages which have not yet reached their destination processors, or

(b) passed to the HSE process if this processor is the message's destination.

```
PROC splitter ()
  SEQ
    receive (no_of_polygons)
    send (no_of_polygons)
    SEQ poly = 0 FOR no_of_polygons
      SEQ
```

```
      receive (polygon)
      send_to_recursive_subdivision_process (polygon)
      send_to_next_processor (polygon)

  WHILE (not finished)
    SEQ
      receive (processor, region)
      IF
        (processor <> this_processor)
          send_to_next_processor (processor, region)
        TRUE
          send_to_recursive_subdivision_process (region)
:
```

The `recursive_subdivision` process is simply the serial recursive subdivision algorithm with some extra lines to receive the scene data and send out the resulting screen information to be painted. The `combiner` process combines incoming screen information from further along the processor chain with similar, locally generated screen information and passes it all back along the chain towards the master processor. It is similar to the splitter process.

3·2·1·1. Cost Estimate

Since the algorithm executing on each processor is essentially identical to the serial case, the costs are very similar. The only major difference is that initially all of the processors must cull the full scene description against their particular regions. Subsequent levels of subdivision will be identical to those of the serial case, (except that several of them are now being executed simultaneously). The duplication of start-

up costs may be expected to reduce the speedup possible for parallel execution of this algorithm, and will limit the maximum possible speedup. However, other limits of the maximum reasonably attainable speedup, such as bad load balancing, will most likely limit the performance before this becomes a major factor.

As for the serial case, two cases will be considered, scenes with large polygons and scenes with small polygons. Considering only terminal nodes as before, the costs are as follows.

Large polygons:

For this case, single pixel terminal nodes will tend to occur only along visible shared edges in the polygonal scenes. Hence the number of such nodes will be approximately equal to the total length of visible shared edges in the scene.

Total cost is $O\left(\sqrt{\frac{n\ m\ F_T}{D_C}}\right)$.

Small polygons:

When the scene consists of small polygons, the number of terminal nodes will be approximately $O\ (F_T / D_C)$ and each such node costs $O\ (D_C)$.

Thus the total cost is $O\ (F_T)$.

Unlike the serial case, the parallel case now has other potentially significant costs, the initial cull and other intermediate stages. One initial cull will take place for each processor used. The cost of culling full scene description against a window of an area (screen area / no. of processors) is $O\ (F_T)$. For large numbers of processors the cost of these root nodes could approach or exceed that of the terminal nodes.

For scenes with small polygons, the cost will be $O(F_T) + O(F_T) = O(F_T)$. So although the extra cost will reduce the speedup for many processors, the cost of the algorithm will grow as $O(F_T)$ as for the serial case.

For scenes with large polygons, the cost will be $O(F_T) + O\left(\sqrt{\frac{n\ m\ F_T}{D_C}}\right)$. Not only will the extra cost reduce the speedup for large numbers of processors but the cost will now grow more as $O(F_T)$ than $O\left(\sqrt{F_T}\right)$, resulting in an even worse performance than for the small polygon, many processor case.

3·2·2. Scan Line Algorithms

Since the two variations of this algorithm consider each screen line in turn, they were parallelised by giving each processor every nth screen line to consider, (while using n worker processors). This division of work results in good load balancing for small to medium numbers of processors because areas of unusually high detail will extend across several screen lines, and hence will be handled by several processors. As the number of processors approaches the number of screen lines, load balancing becomes more problematic. While areas of exceptional detail are still likely to spread across several screen lines, this will only occupy a limited number of processors. Also, the processors corresponding to the top and bottom of the screen may be very lightly loaded if the image is conventionally framed, with most of the image in the centre of the screen.

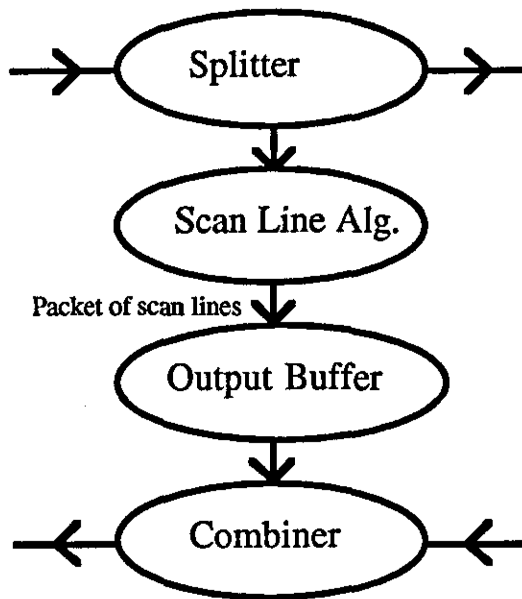


Figure 3·5: *The processes running on each worker node for the scan line algorithm.*

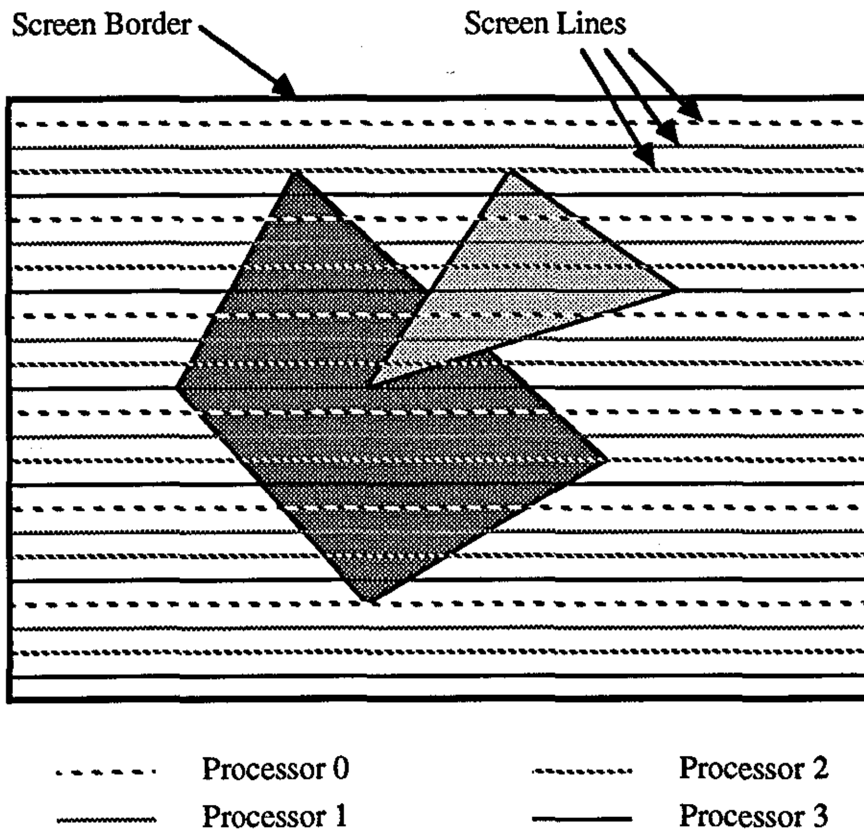


Figure 3·4: *The division of work for four processors. Every fourth screen line is given to each processor.*

As for the recursive subdivision case, the master processor passes the polygon data to the worker processors, and also passes any returned screen painting information to the screen processor. Each worker processor runs four processes in pseudo-parallel, (Figure 3·5) - one for handling and passing along data from the master processor, one that actually runs the implemented scan line algorithm, one that buffers one screen lines worth of output data, and finally one that handles the transmission and passing along of data back to the master processor.

3·2·2·1. Unoptimised Parallel Scan Line Algorithm

The parallel scan line algorithm (without the edge table optimisation) is described by the following pseudo code. As previously discussed, each processor handles every 'n'th screen line. The scan conversion is done a screen line at a time, starting at the lowest relevant screen line. The lowest relevant screen line is calculated from the lowest screen line (min_y) and the processor identification number (this_processor).

```
SEQ
  read_in_polygons ()
  y = lowest_relevant_screen_line (this_processor, min_y)
  WHILE y <= max_y
    SEQ
      reset (store)
      SEQ polygon = 0 FOR all_polygons
        SEQ
          find_resultant_scan_lines (polygon, y)
          append_scan_lines_to_store (store)
      resolve_z_overlaps (store)
      output_scan_lines (store)
```

```
y = y + no_of_processors
```

Apart from stepping up the screen in steps equal to the number of processors being used, this is very similar to the serial version. None of the major program blocks are significantly altered. The splitter and combiner processes are conceptually very similar to those described for the parallel recursive subdivision algorithm, and so are not described here. The buffer process simply paints resulting line segments into the screen buffer.

3·2·2·2. Cost Estimate for the Unoptimised Scan Line Algorithm

In the serial version, the HSE problem was solved for each screen line almost totally independently of the solution of the other screen lines. The only cost shared by the screen line solutions is a small initial step which calculates some items which describe the polygon edges, (such as their slopes). This step was used in the serial program to avoid repeating the calculation of frequently used variables and was inexpensive. Due to the similarities between the parallel and serial versions of this HSE algorithm, the cost estimates for the serial version are relevant here.

Initialisation steps:

• The edges ends are adjusted. This step costs O (no. of edges).

• For each edge, a number of properties are calculated and stored. O (no. of edges).

For each screen line handled by a particular processor:

• Every edge is checked for relevance at a cost of O (edges).

• Intersections are then found for the relevant ones at a cost of O (relevant edges).

• These intersections are "paired up" using a bubble sort. Assuming the polygons are convex, this has a cost of O (polygons)

The HSE Step. For each screen line handled by a particular processor:

• If no line segments overlap, a cost of O($segments^2$). • In the worst case, O($segments^2 + segments^4$). • For a scene with a depth complexity D_C, a cost of O ($segments^2$).

The painting step. The visible pixels for each processor's screen space are painted:

• A simple step costing O (pixels).

Total Cost, using N as the number of processors:

• Part 1 cost O (edges) + O (edges) = O (F_T).

• Part 2 cost O ((lines/processors) * edges) + O ((lines/processors) * relevant edges) + O ((lines/processors) * polygons) = $O\,(n\,E_t / N) + O\left(\frac{1}{N}\sqrt{nm\,F_T\,D_C}\right) + O\,(n\,F_t / N)$.

• Part 3 cost O ((lines/processors) * $segments^2$) = $O\,(D_C\,F_T\,m / N)$.

• Part 4 cost O (nm / N).

Overall cost is thus $O\,(F_T) + O\,(n\,F_T / N) + O\left(\frac{1}{N}\sqrt{n\,m\,F_T\,D_C}\right) + O\,(D_C\,F_T\,m / N) + O\,(nm / N)$. All but one of these terms are dependent upon the model size (F_T) and most are directly proportional to it. The other term should be negligible due to its relative simplicity. As in the serial case, for large numbers of polygons this algorithm's cost will grow as O (F_T). The cost of the initialisation steps now forms a greater part of the overall cost since it does not depend upon the number of processors in use while the other costs decrease with more processors. However, these initialisation steps are inexpensive by comparison with the other costs and will not greatly affect the available speedup. They will eventually limit the available speedup for some very large number of processors, but this number is likely to be large enough not to be a practical problem.

3·2·2·3. Optimised Parallel Scan Line Algorithm

This version uses edge tables to avoid recalculating which polygons are relevant to the screen line currently being considered. It is described by the pseudo code:

```
SEQ
  read_in_polygons ()
  SEQ polygon = FOR all_polygons
    SEQ
      obtain_bottom_and_top_of_each_polygons (bottom, top)
      store_polygon_id (add_polys[bottom])
      store_polygon_id (remove_polys[top])

  find_polygons_intersecting_first_screen_line (current_polygons)
  y = lowest_relevant_screen_line (this_processor, min_y)
  WHILE y <= max_y
    SEQ
      reset (store)
      SEQ polygon = 0 FOR all_polygons
        SEQ
          find_resultant_scan_lines (polygon, y)
          append_scan_lines_to_store (store)
      resolve_z_overlaps (store)
      output_scan_lines (store)
      SEQ i = 0 FOR no_of_processors
        SEQ
          remove_expired_polygons (current_polygons, remove_polys[y])
          add_new_relevant_polygons (current_polygons, add_polys[y])
```

```
y = y + 1
```

This is almost identical to the serial case, except that every nth screen line is considered instead of every screen line being considered. Unlike the unoptimised scan line case, there is a considerable cost shared between the solutions of the screen lines - that of the creation of the edge tables. The splitter, combiner and buffer processes are identical to those of the unoptimised scan line case.

3·2·2·4. Cost Estimates for the Parallel Optimised Scan Line Algorithm

Due to the similarities with the serial version, most of the cost estimates are identical.

Initialisation Steps:

• The ends of the edges are adjusted. The cost is O (no. of edges).

• For each edge, a number of properties are calculated and stored. O (no. of edges).

• The edge tables are prepared. This costs O (polygons).

The Scan Conversion Step. For each screen line considered by a processor:

• The list of relevant polygons is updated, cost of O (change in relevant polygon set).

• The edges are checked for relevance at a cost of O (relevant polygons * 4).

• Intersections are then found at a cost of O (relevant edges).

• These intersections are bubble sorted. Assuming the polygons are convex this has a cost of O (polygons)

The HSE Step. This is repeated for each screen line considered by a processor:

• If no line segments overlap, this costs O ($segments^2$). • In the worst case, this costs approximately O ($segments^2$ + $segments^4$). • For a scene with a depth complexity D_C, a cost of O ($segments^2$).

The painting step. The visible pixels for each processor's screen space are painted:

• A simple step costing O (pixels).

Total Cost, using N as the number of processors:

• Part 1 cost O (edges) + O (edges) + O (polygons)

$= O(E_t) + O(E_t) + O(F_T) = O(F_T)$.

• Part 2 cost O (lines * change in relevant polygon set) + O ((lines/processors) * relevant polygons * 4) + O ((lines/processors) * relevant edges) + O ((lines/processors) * polygons)

$= O(F_T) + O\left(\frac{1}{N}\sqrt{F_T\ m\ n\ D_C}\right) + O(n\,F_T / N)$.

• Part 3 cost O ((lines/processors) * segments^2) = $O(D_C\,F_T\,m / N)$.

• Part 4 cost O (nm / N).

The overall cost is therefore $O(F_T) + O\left(\frac{1}{N}\sqrt{F_T\ m\ n\ D_C}\right) + O(n\,F_T / N) +$

$O(D_C\,F_T\,m / N) + O(nm / N)$.

As for the serial case, most of these component costs vary with model size (F_T) with powers of 0.5 to 1.0. For large numbers of polygons the overall cost is $O(F_T)$.

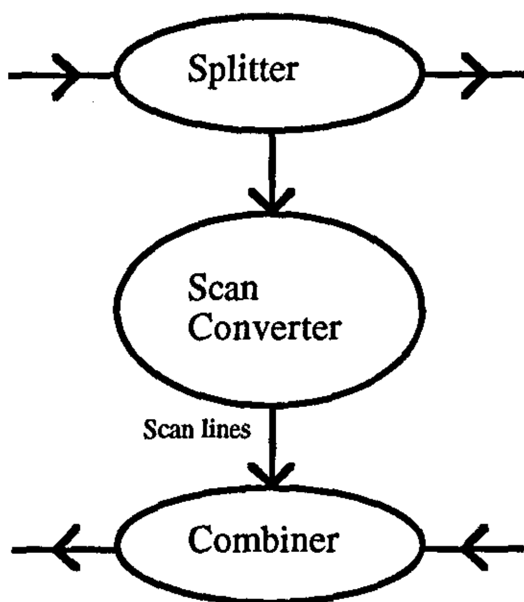
For increasing numbers of processors, the initialisation costs grow in comparison with the other costs. Since they are already a significant cost they will considerably reduce the available speedup and will limit the maximum reasonably attainable speedup.

3·2·3. Z-Buffer Algorithm

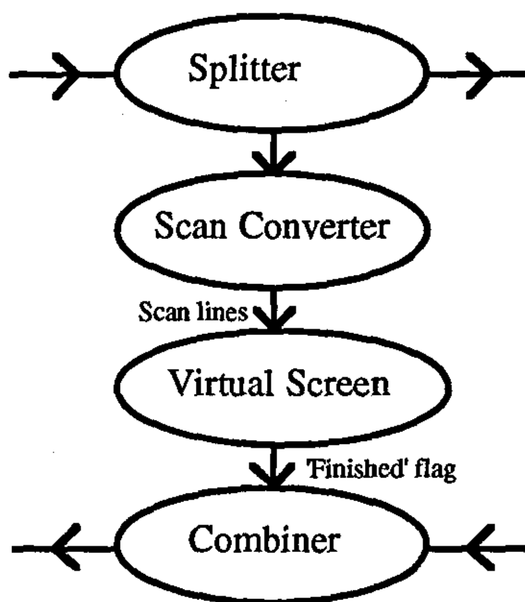
Initially this algorithm was parallelised by using multiple processors to scan convert polygons, (each processor converts a precalculated portion of the polygon database), all feeding one z-buffer running in software on the screen processor. However, the screen processor formed a bottleneck so subsequently an alternative parallelisation was devised.

This alternative scheme used n worker processors calculating scan lines for all polygons but for every nth screen line, (as for the hidden scan line algorithms). Also residing on each worker processor is a z-buffer process which handles the screen for every nth screen line. In this case the screen is treated as being distributed, i.e. part of its hardware is considered to be placed on each of the worker processors. Since such a distributed screen was not actually present, it was emulated by storing the images in normal memory.

This unusual distributed "virtual" screen is not as unrealistic as it may sound - several "real" versions have been constructed by researchers, though the author knows of none that are available on a large scale commercial basis. (For the benefit of the user, and to verify output, the sub-images are subsequently sent to a real screen and combined for display).



(a)



(b)

Figure 3·6: *The processes running on each worker node for the z-buffer algorithm, (a) without and (b) with the "virtual" screen, respectively.*

Like the serial version, this program processes one polygon at a time. It simultaneously works its way up the left and right sides of the polygon, interpolating coordinates between vertices. The resulting scan line segments are then drawn into the z-buffer one pixel at a time, using the method described previously. The only difference from the serial version is that the algorithm works its way up the polygon n lines at a time, for a system with n processors.

3·2·3·1. Cost Estimate

Each polygon is converted to line segments in turn:

• A initialisation step builds lists of edges, this costs O (E_T) for each polygon.

• The scan conversion steps through the y-range of each polygon at a cost of O (H_f / processors) for each polygon.

• Each pixel in a line segment is tested against the z-buffer at a cost of O (segment length) per segment. Visible pixels are then painted into the z-buffer at a cost of O (visible pixels / processors).

Total Cost per processor, using N as the number of processors:

• The cost is O (E_T) + O (H_f / N) + O (screen area * D_C / N) + O (screen area / N)

$$= O(F_T) + O\left(\frac{1}{N}\sqrt{n\ m\ D_C\ F_T}\right) + O(n\ m\ D_C / N) + O(n\ m / N).$$

As for the serial version, the first two cost terms grow with the size of the test scene, so for large numbers of polygons the cost of this algorithm will be O (F_T). However, if the polygons are of multiple pixel area then these two costs are swamped by the per-pixel cost since they occur only once per polygon (for the first term) or once per

segment (for the second term). The overall cost for large or medium size polygons will therefore tend to be O (n m). For large numbers of very small polygons, the overall cost would be controlled by the O (F_T) term.

For large numbers of processors, the first term remains fixed while the other terms decrease. This will result in the parallelised algorithm behaving more as O (F_T) for many processors than the serial case did. Since the initialisation step must be done by all processors it reduces the available speedup, particularly for scenes with many small polygons. This factor will also limit the maximum available speedup.

3·2·4. Painter's Algorithm

The initial parallel implementation used n worker processors to each sort an nth of the polygon database, merged these sorted portions of the database (in the process of passing these portions back to the master), retransmitted the now sorted database from the master to the workers, and scan converted these polygons in order, sending the scan lines to the screen processor for display. As for the z-buffer case, the screen processor became a limiting factor and so the screen was again treated as being distributed.

A potentially limiting factor for this algorithm is the bottleneck caused by having to return the sorted portions of the database in order to merge them.

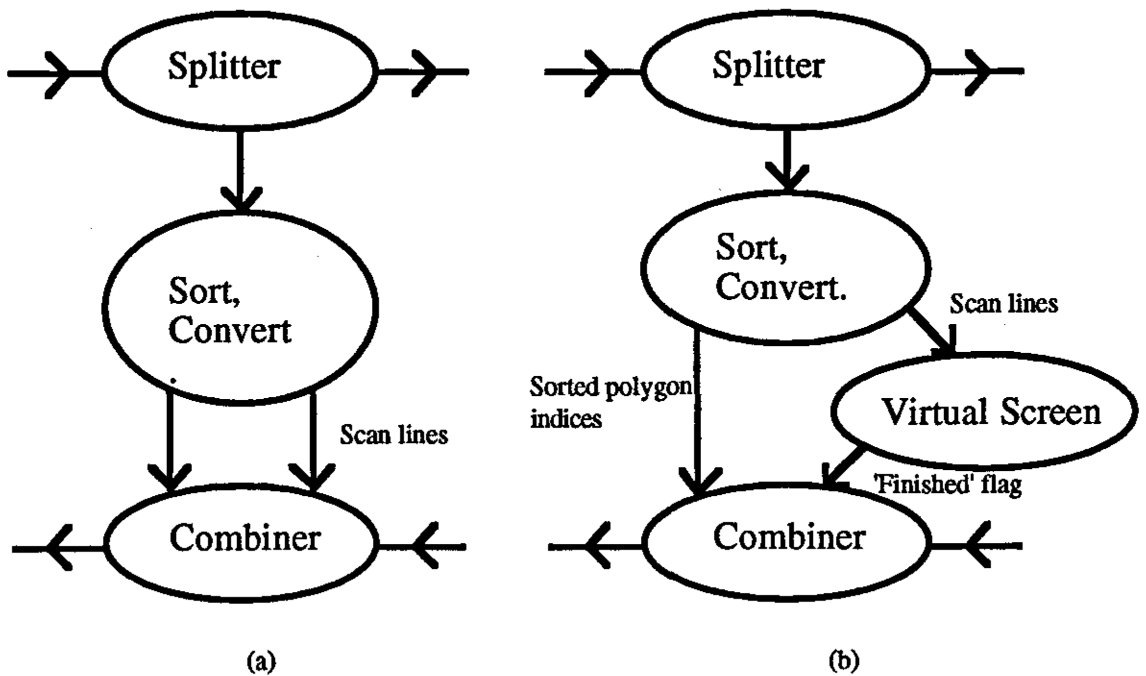


Figure 3·7: *The processes running on each worker node for the painter's (depth sort) algorithm, (a) without and (b) with the 'virtual' screen, respectively.*

The following pseudo code describes the main HSE process:

```
[running on processor 'j' of 'n'; 0 <= j <= (n-1)]
SEQ
  Get polygons
  Wait for start flag
  Pass start flag upstream (to the next processor in the chain)
  Bucket sort (by depth) the subset of polygon indices for the 'j'th
            part of the polygon store
  Send the sorted subset of indices (with polygon z-values) in order,
            to the combiner process
```

```
  WHILE not finished
    SEQ
      Get a polygon index
      Scan convert that polygon for every 'n'th line
      Output resulting scan lines to screen process
:
```

The combiner process merges the sorted parts of the polygon database into a sorted full database in the process of passing them back to the master. The merge is illustrated by figure 3·8. The combiner process itself is described by the next piece of pseudo code.

```
SEQ
  Get Polygon_index1:Z_value1 pair from local main process
  Get Polygon_index2:Z_value2 pair from upstream (next processor in
                chain)
  WHILE not finished
    SEQ
      IF
        Z_value1 > Z_value2
          SEQ
            Send Polygon_index1:Z_value1 downstream (to previous
                                        processor)
            Get Polygon_index1:Z_value1 pair from local main process
        Z_value1 <= Z_value2
          SEQ
            Send Polygon_index2:Z_value2 downstream
            Get Polygon_index2:Z_value2 pair from upstream
```

```
Wait for END flag from local main process

Wait for END flag from upstream

Send END flag downstream


Pass screen data from upstream to downstream

Pass locally generated screen data downstream

:
```

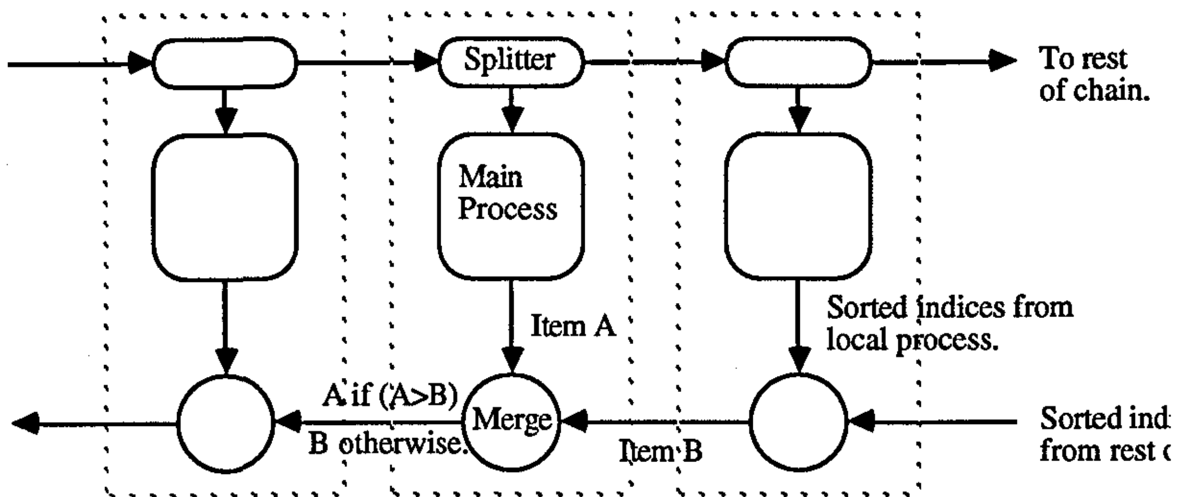


Figure 3·8: *An illustration of the "merge" part of the merge-sort of polygons. When a "combiner" process is working on the "merge" of the pre-sorted pieces of the polygon database it will see two incoming indices to polygons, one from the local sort process, and one from the merged results of the upstream processors. It simply passes along the index to the polygon with the greatest z-value. This is repeated until the merge is complete. [An index into a list of polygons is used rather than the actual polygon to reduce the amount of information to be transferred.]*

3·2·4·1. Cost Estimate

• In an initialisation step, the average z value of each polygon is found.

This costs O (polygons * 4).

• The polygons are then sorted on their average z values using a bucket sort. This costs O (buckets + (polygons / processors)) for each part of the sort, plus O (polygons) for the data transfer part of the merge and O (processors) for filling and emptying the pipeline, (as discussed in section 1·5·1). These last two terms are dependent upon the communication architecture used for this particular implementation of the algorithm.

• Each polygon is then scan converted. This costs O (H_f / processors) for each polygon.

• The pixels are painted. This costs approximately $O\left(\frac{nmD_C}{N}\right)$ for each processor.

Total Cost, using N as the number of processors is :

$$O(F_T) + O(\text{buckets} + (F_T / N)) + O\left(\frac{1}{N}\sqrt{n\ m\ D_C\ F_T}\right) + O(N) + O\left(\frac{nmD_C}{N}\right).$$

As for the serial case, the algorithm still costs O (F_T) for large numbers of polygons. For increasing numbers of processors, the initialisation and merge terms remain fixed. Also the sort term decreases in cost very slowly unless the number of buckets is adjusted to suit the number of processors in use. Thus much of the cost of this algorithm cannot be spread over multiple processors, greatly reducing the speedup and placing a rather low limit on the maximum possible speedup. Also, for large numbers of processors arranged in the pipeline architecture used for this implementation of the algorithm, the O (N) term which is due to pipeline filling and emptying delay may become significant and actually increase the time to complete the work compared with smaller numbers of processors.

3·3. Timing Information

In all cases, the screen store(s) were initialised before the clock was started. The fast, on-chip RAM was not used. This decision was made because the on-chip RAM is limited in size to 4K bytes and hence affects the execution speeds of programs of different sizes, since differing proportions of such programs fit in this high speed RAM. Instead, all program and data were stored instead in the slower, expandable, external RAM.

There are many possible ways to connect a number of transputers together. In particular, the way data is fed to graphical stages such as those tested in this work can depend heavily on the architecture of the previous stage of the system. To give the results presented here relevance to connection schemes other than the one used for this work, machine dependencies have been avoided where possible.

Since the machine architecture (of a MIMD machine) primarily affects only the communication rates within a parallel system, the ignored costs are simply a function of the amount of data being moved around the system and the available communications bandwidth. Note that in each case the clock is started after transferring the polygon data to the worker processors, in order to avoid these machine dependencies.

For the recursive subdivision algorithm the polygon data were first sent to the worker processors, which precalculated and stored the plane coefficients before the clock was started. The clock was stopped upon reception of completion flags from all of the workers.

For the two scan line algorithms the polygon data were first sent to the worker processors before the clock was started. The clock was stopped upon reception of

completion flags from all of the workers. For the version using edge tables, the time taken to create the edge tables is included in the execution time.

For the z-buffer and painter's algorithms the polygons were sent to the worker processors, then the clock was started. The clock was stopped upon reception of completion flags from all of the workers.

All timings were made by the master processor using its low priority clock, which ticks 15625 times per second.

3·4. Results

For each of the five parallel HSE algorithms previously described, for fourteen different numbers of processors (except for the recursive subdivision algorithm), timings were taken for the solution of the hidden surface problem for each of the five teapot scenes and three tetra scenes discussed.

As mentioned in section 1·3·4, two particularly useful measures of the advantages of a parallel program are "speedup" and "linearity of speedup", (the latter is simply referred to as linearity in the results tables). Speedup is simply how many times faster the algorithm executes compared with the one processor case. Linearity of speedup could also be called the efficiency of parallelisation in that it measures the fraction of the maximum possible speedup obtained in practice. i.e. if a program runs three times faster on four processors than on one, then the linearity is 3/4 or 75 percent. These two measures of how well the algorithms performed when parallelised are used often in the following consideration of the results.

In order to improve the correlation between the actual results and the theoretically derived cost estimates, the cost estimates were simplified and suitable coefficients deduced to show the similarity between the theory and practice. This is similar to the method used by Dixon et. al.[49], although their analysis was unfortunately not directly applicable to this work.

3·4·1. Recursive Subdivision Algorithm

The recursive subdivision algorithm showed falling linearity for increasing numbers of processors, (Figures 3·9 and 3·10). A speedup of twelve times by using sixty-four processors is useful but far from the ideal. These losses were almost certainly a consequence of bad load balancing. Watching the program run, it becomes obvious that some of the processors finish their parts of the screen well before the others, due to the images being of uneven complexity. Some of this lost performance could be regained by allowing those processors which have finished their parts of the screen to take over some of the unfinished screen areas, though this would not be a perfect solution.

The differences in speedup between the models are not consistent between the teapot and tetra test scenes, probably due to the significant difference in depth complexity between these sets of test scenes. No strong correlations may be drawn between the cost estimates made for this algorithm and the results, due to the load balancing problems swamping other effects.

Another point of interest is the way in which the execution time increases with the number of polygons in the image, as shown by Figures 3·11 and 3·12. These figures are for the parallel algorithm using sixty-four processors, and are very similar to those for the serial case. They show execution time increasing as some fractional power of

the number of polygons. This gives an indication as to how long the algorithm would take to execute images with far larger numbers of polygons than any tested here.

The similarity between Figures 3·11 and 3·12 and their serial algorithm equivalents suggests that the initial cull overhead has not yet become a significant factor in the total cost. If larger numbers of processors were used, this factor may still become important. In any case, the recursive subdivision algorithm visibly suffers too greatly from bad load balancing to show a good correlation with the cost estimates considered earlier.

Teapot model size	200	2500
Run time, 1 processor	1222352	4947562
Run time, 16 processors	211406	827771
Run time, 64 processors	69794	413791
Speedup	17.5	12.0
Linearity	0.27	0.19

Table 3·1. *Sample execution times and performance statistics for the recursive subdivision algorithm.*

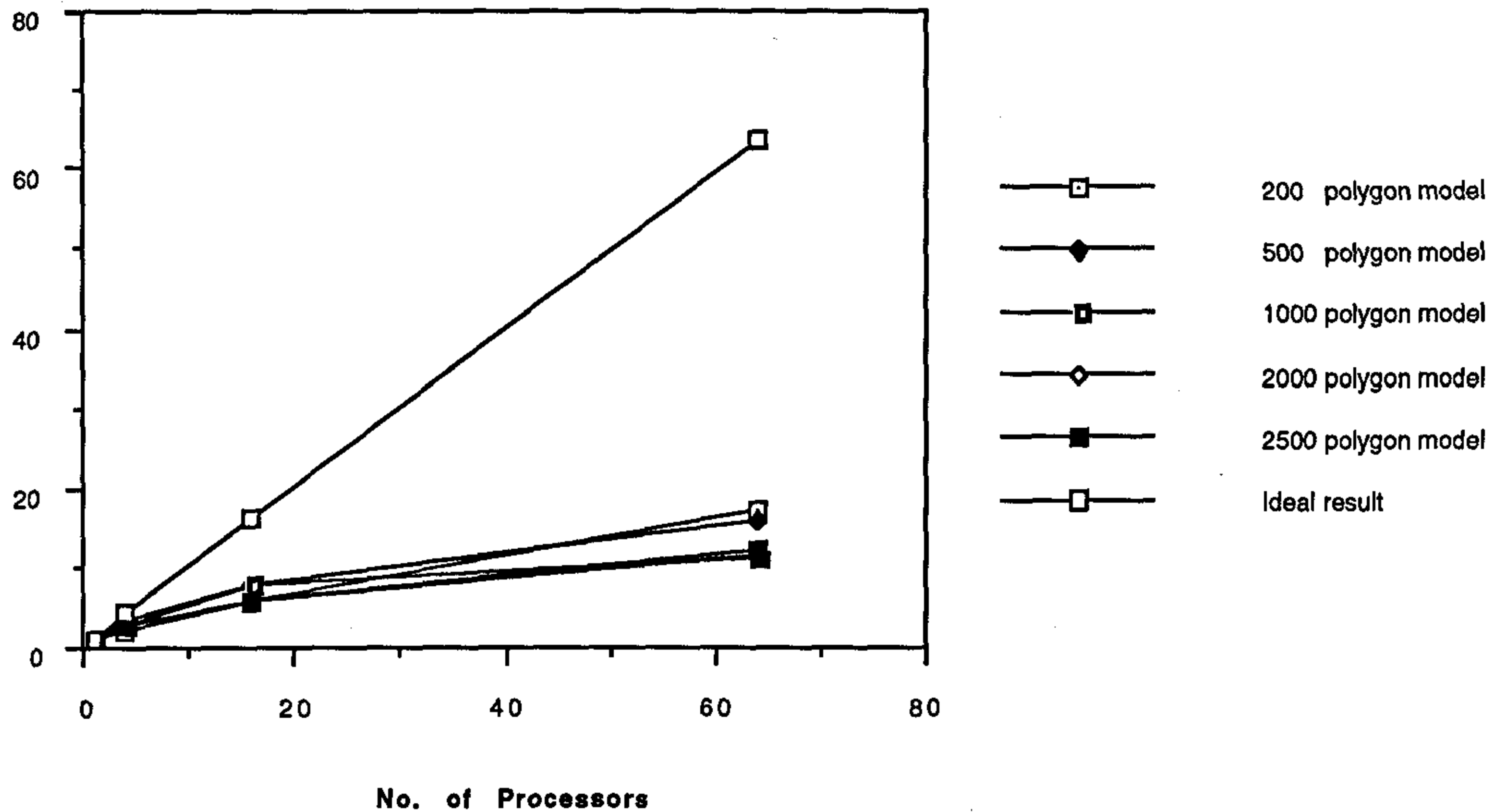


Figure 3·9: Speedup versus no. of processors for the recursive subdivision algorithm and teapot models.

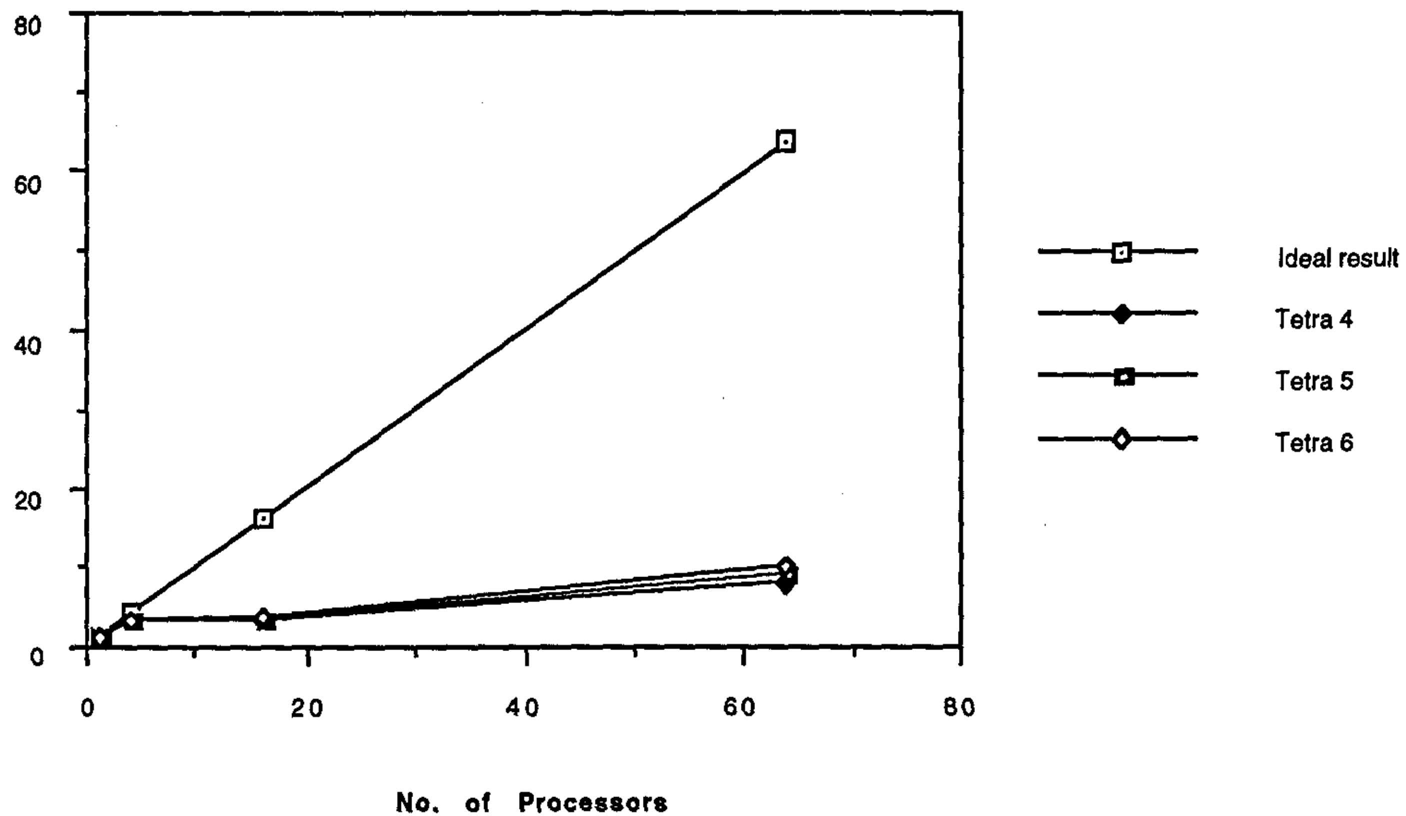


Figure 3·10: Speedup versus no. of processors for the recursive subdivision algorithm and tetra models.

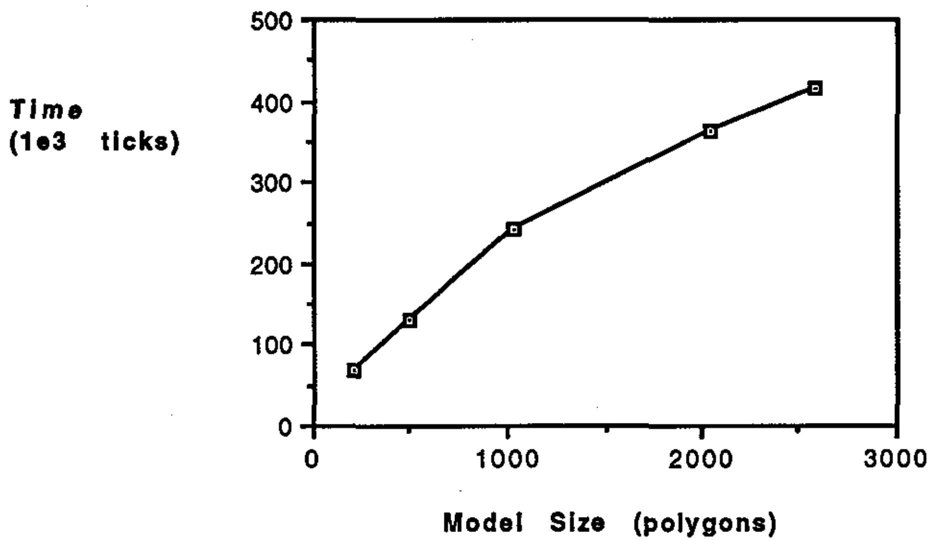


Figure 3·11: *Execution time versus model size for the parallel recursive subdivision algorithm and teapot models, using sixty-four processors.*

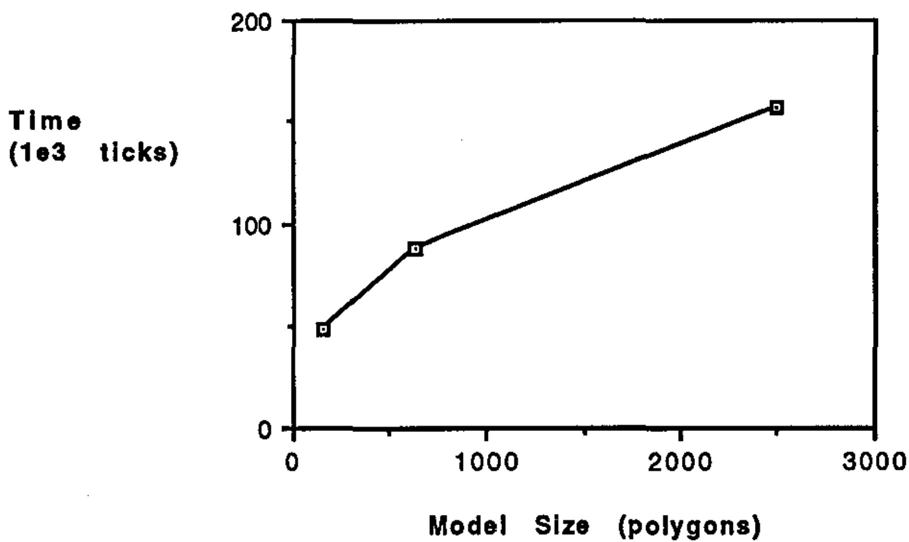


Figure 3·12: *Execution time versus model size for the parallel recursive subdivision algorithm and tetra models, using sixty-four processors.*

3·4·2. Scan Line Algorithms

The scan line algorithm proved to be quite efficient when parallelised. The basic, unoptimised version showed a gentle fall off in linearity as the number of processors was increased, (Figures 3·13 and 3·14), but was still some fifty to eighty percent of the ideal for one hundred and twenty-eight processors.

This small loss of speedup may be attributed to the unshareable cost of calculating various properties for each edge. This irreducible overhead become more important as the number of processors increases since all other costs are divided between the many processors. Thus there is not a simple, constant loss of speedup, but an increasing loss of speedup as the number of processors is increased.

The tetra scenes resulted in slightly lower speedup than the teapot scenes. This is due to the smaller polygon sizes of the tetra scenes causing the fixed overhead to be a greater part of the total cost than for the large polygons of the teapot scenes, since smaller polygons take less work to turn into scanlines than large polygons but the overhead cost is identical. This effect may also be noted separating the speedups of the various teapot scenes in Figure 3·13.

The growth of execution time with model size for the sixty-four processor case is approximately linear for both set of test scenes, (Figures 3·15 and 3·16). This is consistent both with the cost estimates for this algorithm and with the serial version of this algorithm.

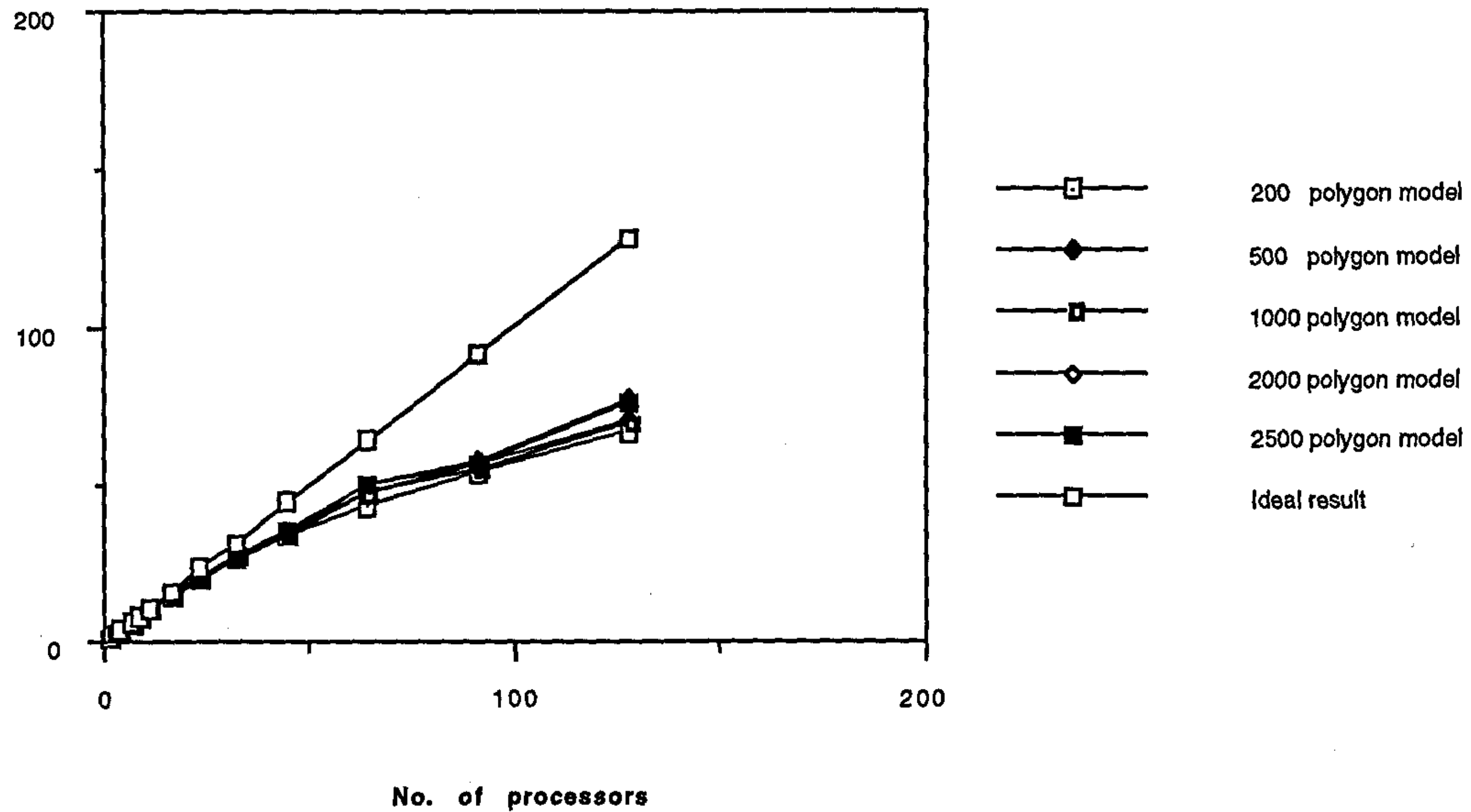


Figure 3·13: Speedup versus no. of processors for the scan line algorithm and teapot models.

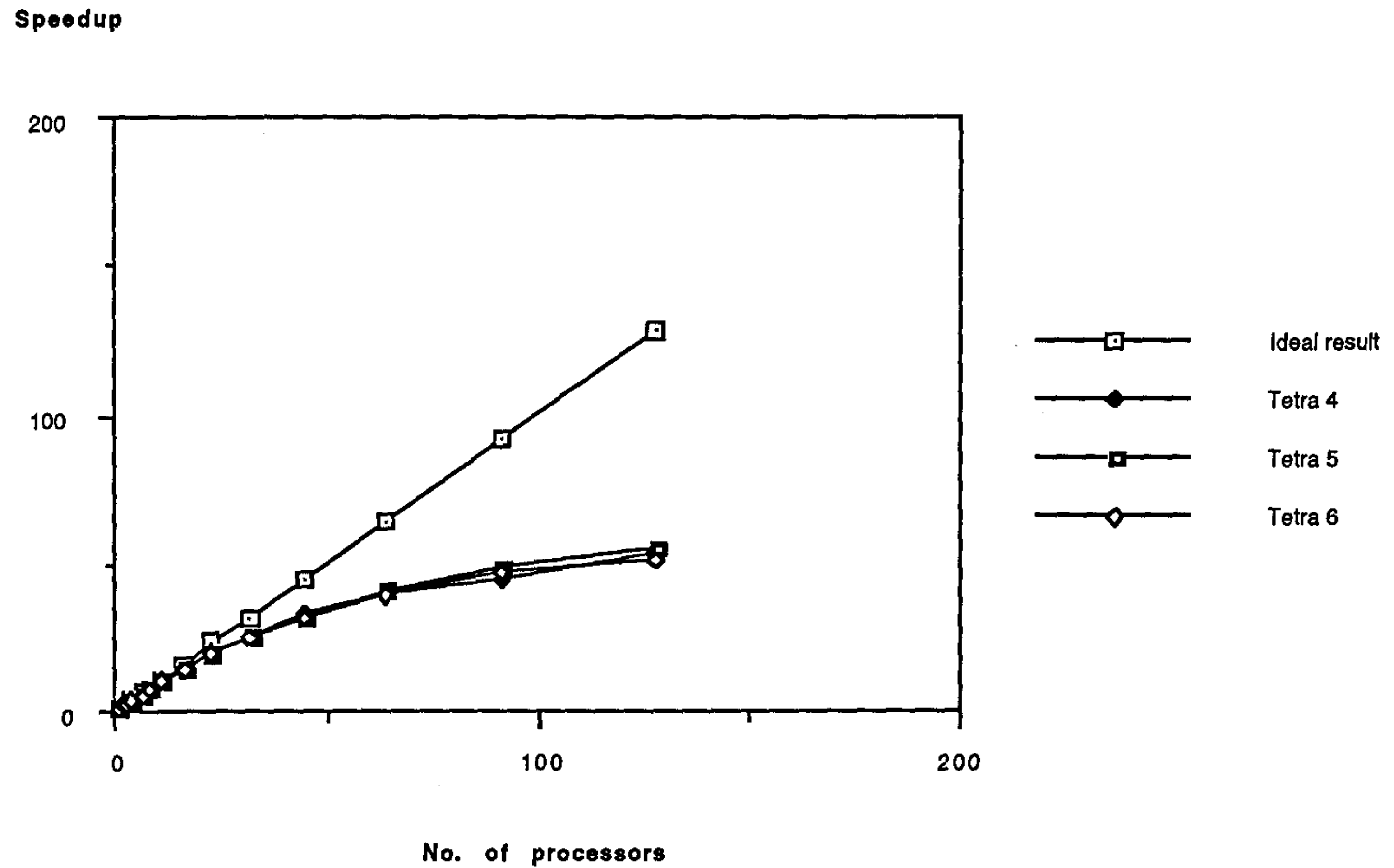


Figure 3·14: Speedup versus no. of processors for the scan line algorithm and tetra models.

Teapot model size	200	2500
Run time, 1 processor	208212	3559407
Run time, 16 processors	14597	236438
Run time, 64 processors	4803	71955
Speedup	43.4	49.5
Linearity	0.68	0.77

Table 3·2. *Sample execution times for the scan line (unoptimised) algorithm.*

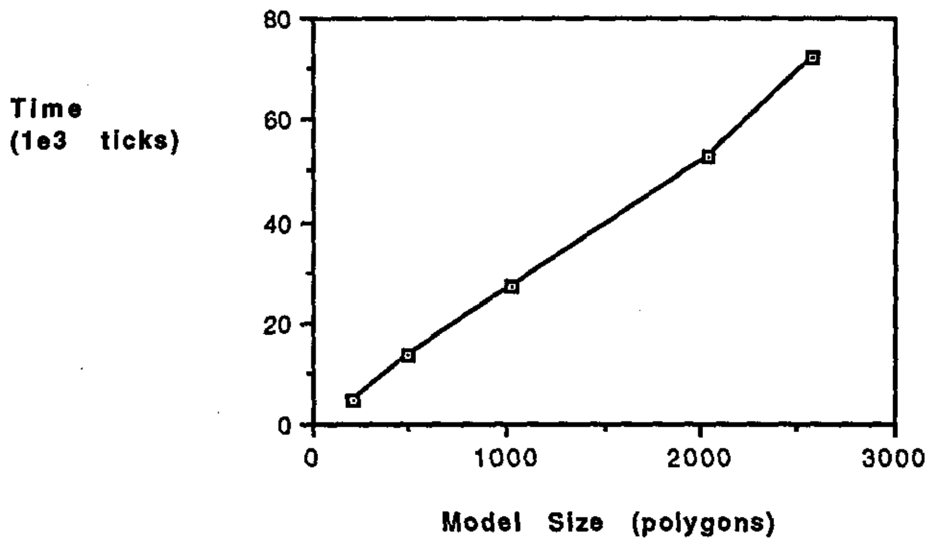


Figure 3·15: *Execution time versus model size for the unoptimised scan line algorithm and teapot models, using sixty-four processors.*

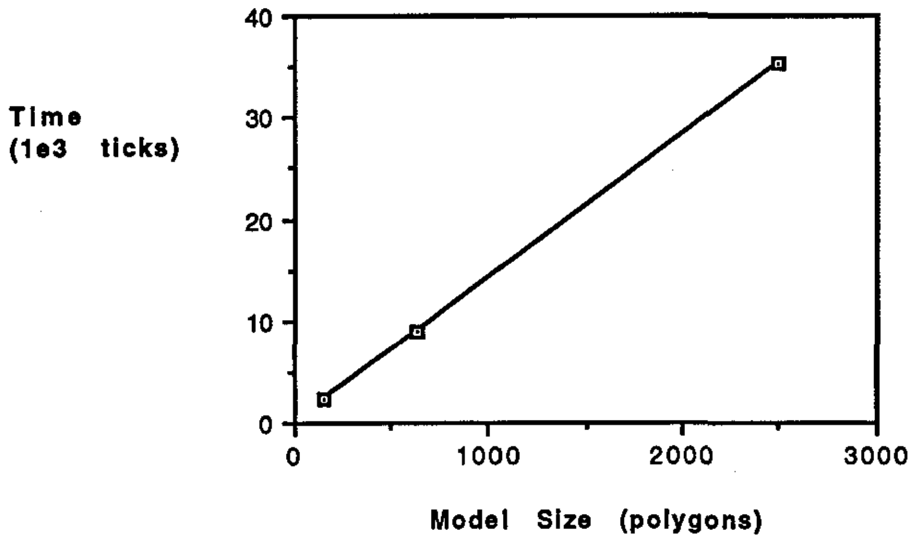


Figure 3·16: *Execution time versus model size for the unoptimised scan line algorithm and tetra models, using sixty-four processors.*

The time-cost estimate derived earlier for the scan line algorithm was

$$O(F_T) + O(n\,F_T / N) + O\left(\frac{1}{N}\sqrt{n\ m\ F_T\ D_C}\right) + O(D_C\,F_T\,m / N) + O(nm / N)$$

which, for a given set of test data, may be reduced to

$$A + \frac{B}{N}$$

where A is the cost of the serial, unparallelisable part of the algorithm and B is the cost of the parallel part of the algorithm. If the costs are normalised with respect to the one processor case then A and B become the fractions of the algorithm which are serial and parallel, respectively. Applying this to the test cases and using regression to calculate the best-fit values of the A and B constants for each scene gives Table 3·3.

Scene	A (fraction serial)	B (fraction parallel)
Teapot 200	0.00745	0.99255
Teapot 500	0.00639	0.99361
Teapot 1000	0.00669	0.99331
Teapot 2000	0.00543	0.99457
Teapot 2500	0.00574	0.99426
Tetra 4	0.01092	0.98908
Tetra 5	0.01003	0.98997
Tetra 6	0.01081	0.98919

Table 3·3: *Regression produced coefficients for the theoretical estimate of the performance of the (unoptimised) scan line algorithm.*

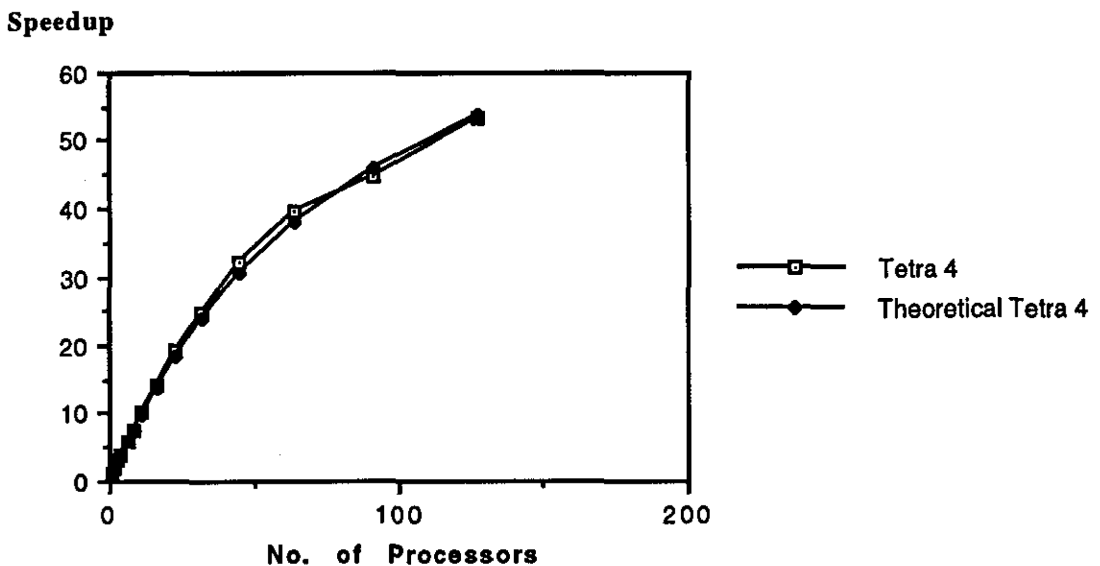


Figure 3·17. *Comparison of actual and expected performance, tetra 4 scene, unoptimised scan line algorithm.*

The cost estimate derived earlier is not perfect, particularly in that it takes no account of where the polygons are within a scene. This means that it is particularly difficult to accurately compare the performance of the HSE algorithm on different scenes, as attested by the wide variations in the A and B coefficients between various scenes, (see table 3·3). Another factor not taken into account in any of the cost estimates is bad load balancing in the cases where $\frac{\text{no. of screen lines}}{\text{no. of processors}}$ is non integer, i.e. some processors get one more screen line to solve than others. Fortunately, this effect is usually minimised by there little or no detail on the bottom lines of a scene. This effect may however be seen in some of the cases, particularly at the large no. of processors end of some of the tetra scenes where it causes the real ninety-one processor result to appear low.

For a given scene the cost estimate provides a very good prediction of real performance as the number of processors is varied. An example of this close match may be seen in figure 3·17.

The parallel edge-table optimised scan line algorithm showed a sharply limited speedup. The tetra case, (Figure 3·18) even showed almost no change in performance when using more than thirty-two processors. The teapot case was not much better, still being limited to a speedup of only about twenty times for one hundred and twenty-eight processors.

This "performance ceiling" effect is completely in agreement with the cost analysis, which concluded that the edge-table method considerably reduced the scope for the effective use of parallelism. This is a good example of Amdahl's law [50], with the serial portion of the algorithm - the creation of the edge-tables - limiting the performance increases available using parallel processing. While the edge-table method formed a useful optimisation for the single processor case, giving a gain of perhaps thirty percent

over the unoptimised scan line algorithm, it has become a serious problem for the multiprocessor case.

The parallel edge-table method also suffers from a corollary of its virtue. Since the spacing between subsequent scan lines on a given processor increases as the number of processors increases, then the similarity between these scan lines decreases - losing the property of coherence which the edge table method exploits. When no polygon has a height in scanlines greater than the number of processors, then there is no exploitable coherence remaining, and the cost of the edge tables brings no benefits. For small numbers of processors this effect could be partially avoided by having each processor handle every nth small group of scan lines rather than every nth scan line, but this might cause load balancing problems.

Figures 3·19 and 3·20 show the growth of execution time with mode size for the sixty-four processor case to be somewhat more linear than their serial equivalents. This is in agreement with the cost analyses which concluded that the linear growth overheads would become more important as the number of processors increased, and would thus reduce the effect of the small non-linear costs.

Teapot model size	200	2500
Run time, 1 processor	150692	2880572
Run time, 16 processors	15881	297713
Run time, 64 processors	8967	168752
Speedup	16.8	17.1
Linearity	0.26	0.27

Table 3·4. *Sample execution times and performance statistics for the scan line (edge table) algorithm.*

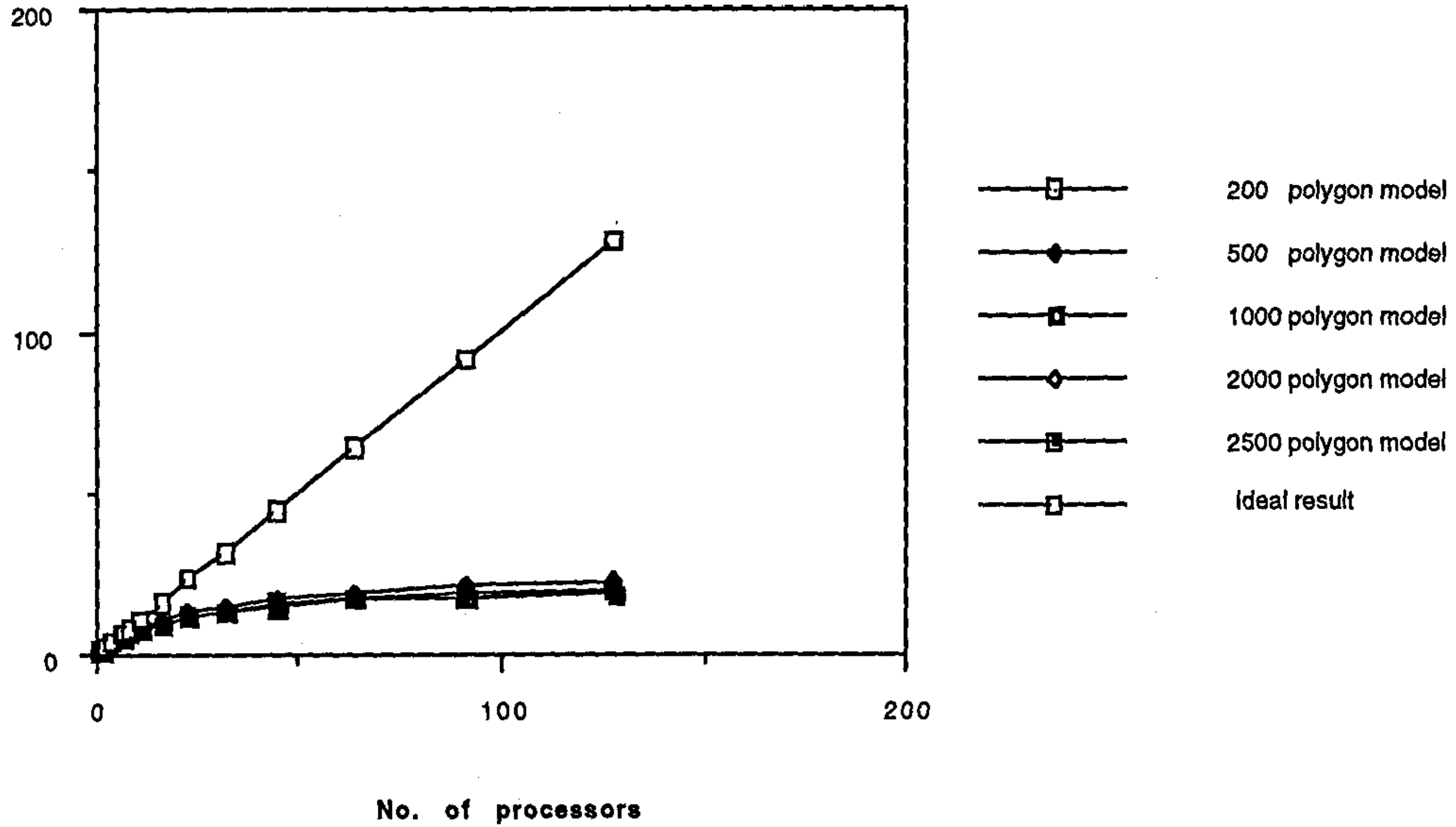


Figure 3·18: Speedup versus no. of processors for the scan line (optimised) algorithm and teapot models.

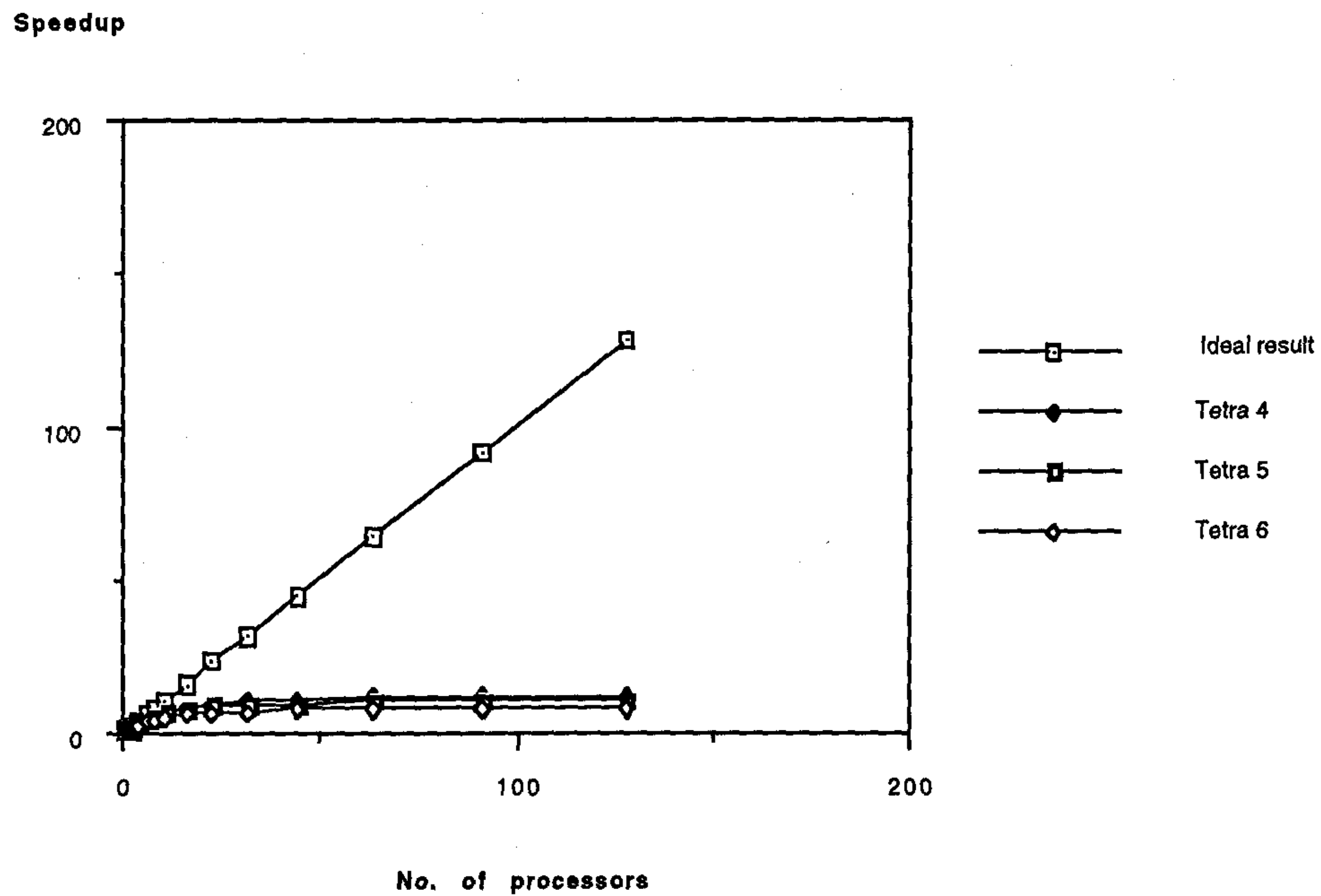


Figure 3·19: Speedup versus no. of processors for the scan line (optimised) algorithm and tetra models.

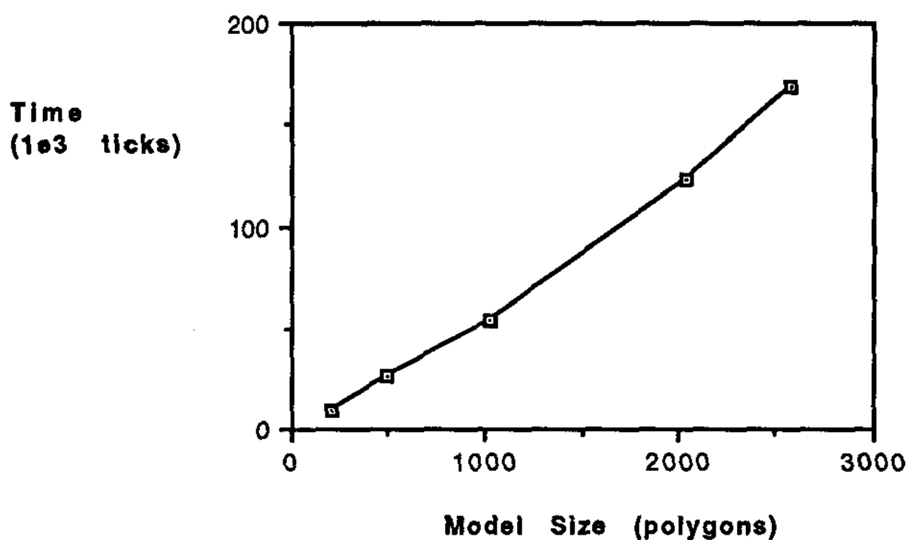


Figure 3·20: *Execution time versus model size for the optimised parallel scan line algorithm and teapot models, using sixty-four processors.*

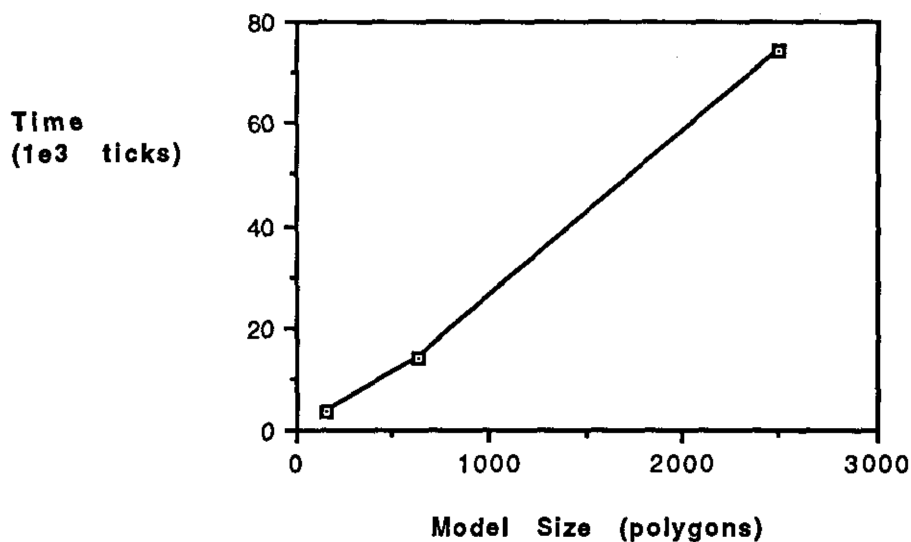


Figure 3·21: *Execution time versus model size for the optimised parallel scan line algorithm and tetra models, using sixty-four processors.*

The cost estimate derived earlier for the optimised scan line algorithm was

$$O(F_T) + O\left(\frac{1}{N}\sqrt{F_T\ m\ n\ D_C}\right) + O(n F_T / N) + O(D_C F_T m / N) + O(nm / N)$$

which, for a given set of test data, may be reduced to

$A + \frac{B}{N}$ as for the unoptimised case. Applying this to the test cases and using regression to derive A and B gives:

Scene	A (fraction serial)	B (fraction parallel)
Teapot 200	0.04511	0.95489
Teapot 500	0.03823	0.96177
Teapot 1000	0.04577	0.95423
Teapot 2000	0.04525	0.95475
Teapot 2500	0.04489	0.95511
Tetra 4	0.07519	0.92481
Tetra 5	0.08465	0.91535
Tetra 6	0.11669	0.88331

Table 3·5: *Regression produced coefficients for the theoretical estimate of the performance of the optimised scan line algorithm.*

Again, the variations in the coefficients between scenes is largely unpredictable (table 3·5), but the variation of performance with the number of processors for a given scene is very well predicted, (figure 3·22).

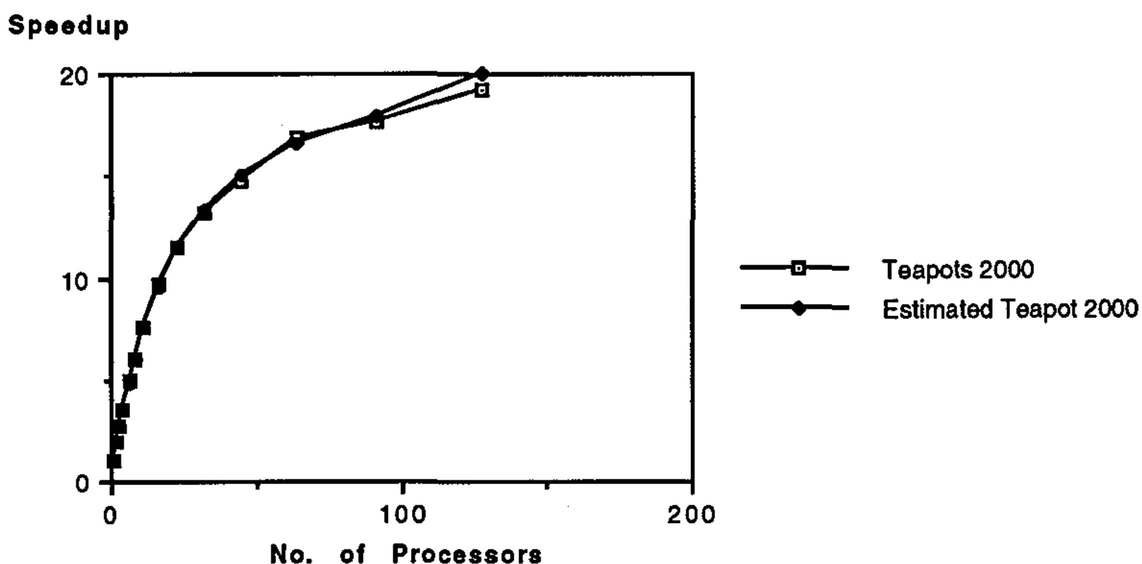


Figure 3·22. *Comparison of actual and expected performance, 2000 polygon teapot scene, optimised scan line algorithm.*

3·4·3. Z-Buffer Algorithm

When using a single processor for screen painting tasks, the z-buffer algorithm did not gain a significant speedup as more processors were used. This was because the screen painting formed a bottleneck in the system. When the bottleneck was avoided by allocating part of the screen painting process to each processor, the speedup obtained varied considerably with model size, (Figures 3·21 and 3·22).

This drop in speedup with increasing model size is consistent with the cost analysis which shows the fixed, unparallelisable cost of calculating edge properties becoming more important as the number of processors increases. Also this per-polygon overhead has a greater effect for small polygons where it forms a large fraction of the total cost, than for large polygons where it forms a smaller fraction of the total cost, (since painting costs are the same within each set of test scenes). Thus the speedup falls roughly in proportion to the height of the polygons in the scene, as may be seen in Figures 3·21 and 3·22.

The increases in execution time with model size is more linear for the sixty-four processor case, (Figures 3·23 and 3·24) than it was for the serial case because the fixed overhead is a linear cost and grows in importance with the number of processors. Indeed the tetra scenes, Figure 3·24, show an almost perfectly linear growth of execution time with model size. This is more linear than for the teapot case because the polygons in the tetra scenes are significantly smaller than those of the teapot scenes.

Teapot model size	200	2500
Run time, 1 processor	509952	581151
Run time, 16 processors	33698	43602
Run time, 64 processors	10243	16323
Speedup	49.8	35.6
Linearity	0.78	0.56

Table 3·6. *Sample execution times and performance statistics for the z-buffer algorithm.*

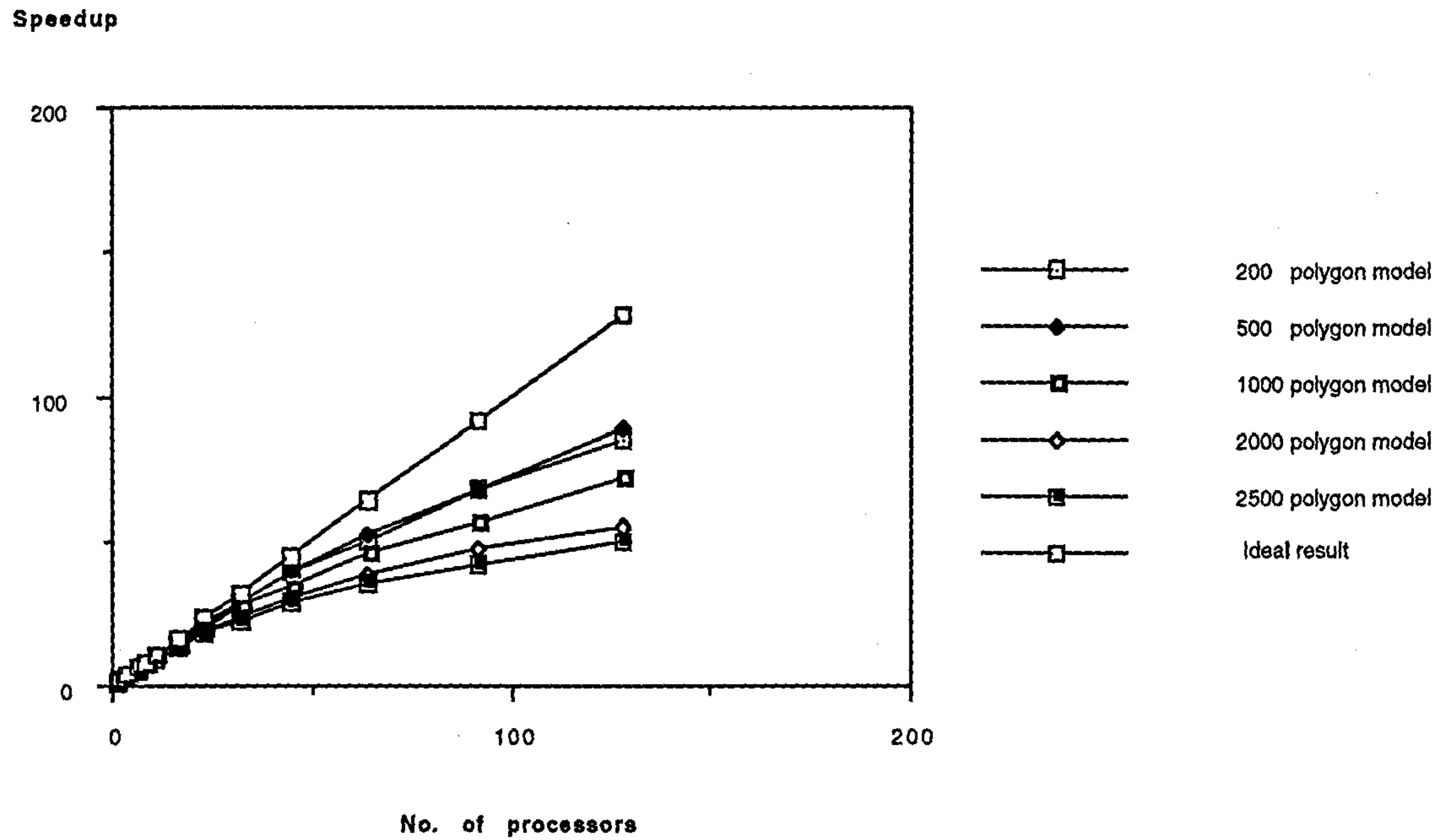


Figure 3·23: Speedup versus no. of processors for the z-buffer algorithm and teapot models.

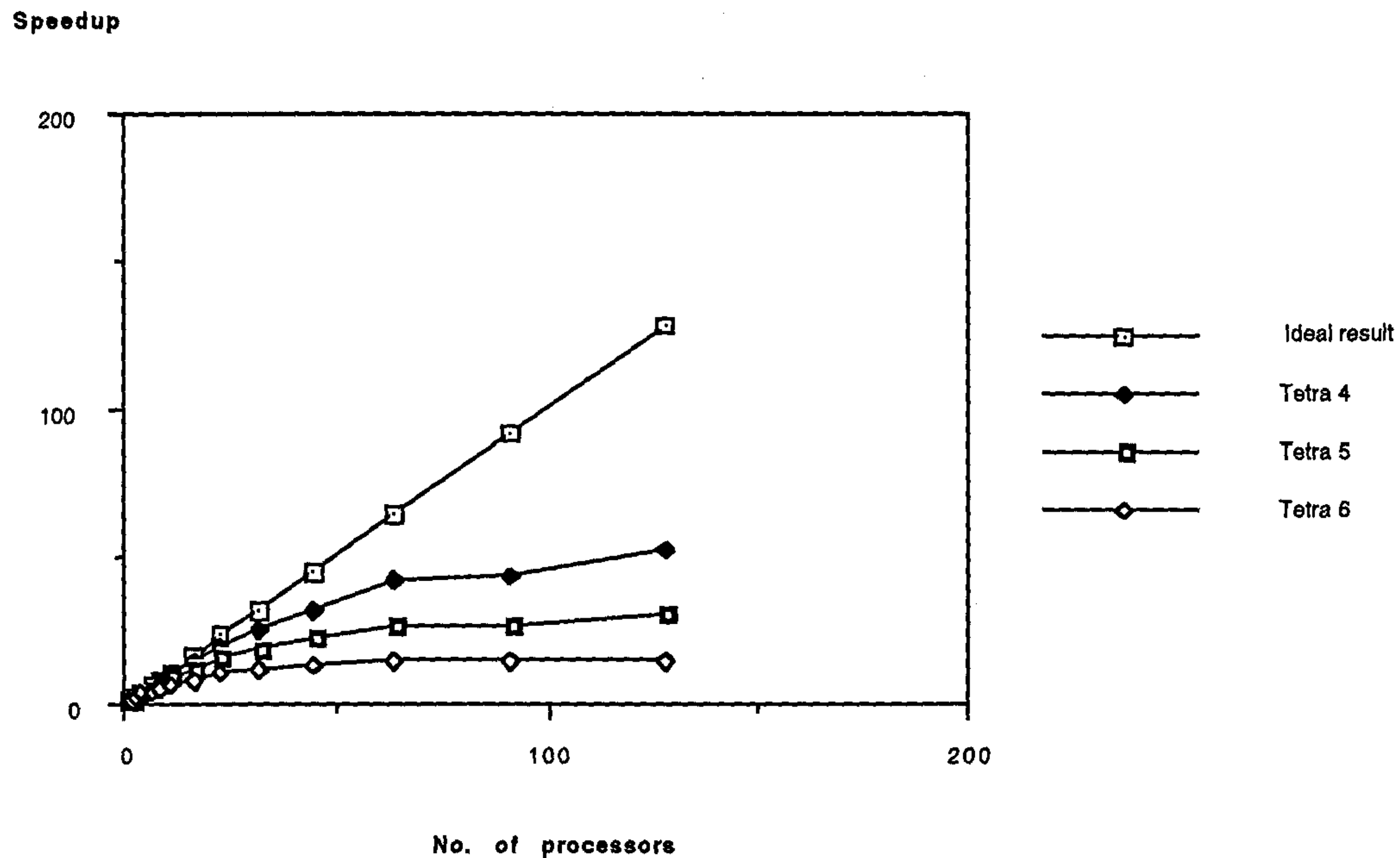


Figure 3·24: Speedup versus no. of processors for the z-buffer algorithm and tetra models.

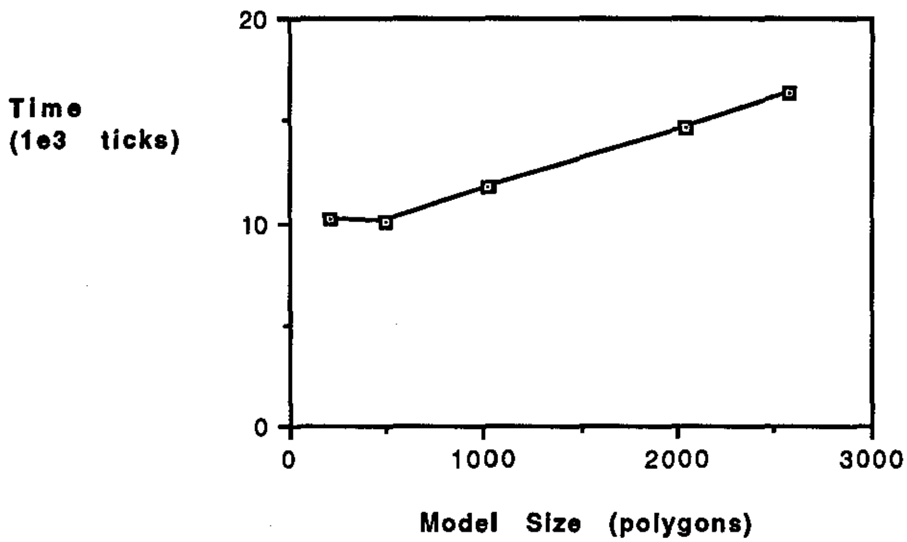


Figure 3·25: *Execution time versus model size for the parallel z-buffer algorithm and teapot models, using sixty-four processors.*

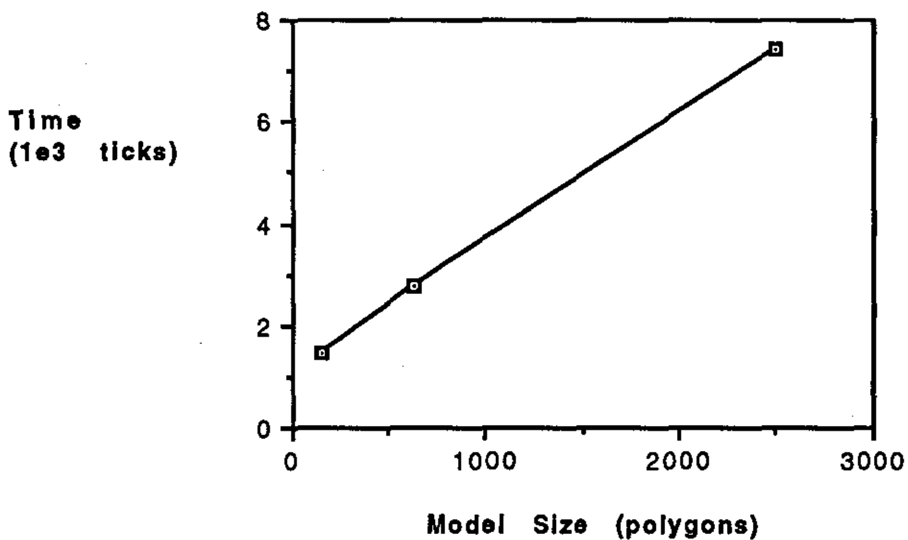


Figure 3·26: *Execution time versus model size for the parallel z-buffer algorithm and tetra models, using sixty-four processors.*

The z-buffer algorithm cost estimate derived earlier was

$$O(F_T) + O\left(\frac{1}{N}\sqrt{n\ m\ D_C\ F_T}\right) + O(n\,m\,D_C / N) + O(n\,m / N)$$

which, for a given set of test data, may be reduced to

$A + \frac{B}{N}$. Applying this to the test cases and using regression to derive A and B gives:

Scene	A (fraction serial)	B (fraction parallel)
Teapot 200	0.00397	0.99603
Teapot 500	0.00346	0.99654
Teapot 1000	0.00632	0.99368
Teapot 2000	0.01045	0.98955
Teapot 2500	0.01263	0.98737
Tetra 4	0.01105	0.98895
Tetra 5	0.02482	0.97518
Tetra 6	0.05926	0.94074

Table 3·7: *Regression produced coefficients for the theoretical estimate of the performance of the z-buffer algorithm.*

For the z-buffer case, as for the two scan line algorithms, the variation of cost between scenes is largely unpredictable (table 3·7). The variation of cost versus the number of processors is however very well predicted, (figure 3·27)

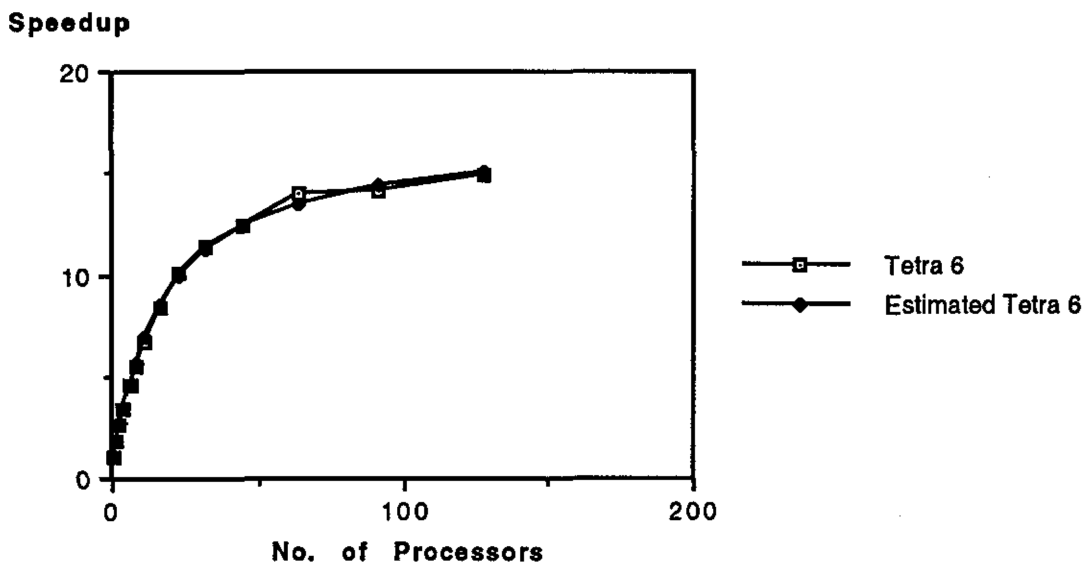


Figure 3·27. *Comparison of actual and expected performance, tetra 6 scene, z-buffer algorithm.*

3·4·4. Painter's Algorithm

For small numbers of processors, the painter's algorithm showed a good speedup, (Figures 3·25 and 3·26). However, for large numbers of processors the speedup became sharply limited and sometimes even fell for more than thirty-two processors. This agrees with the cost estimate, which found that much of the algorithm could not be shared over multiple processors with the per-polygon overhead and merge steps remaining fixed, and the pipeline filling part of the merge step actually growing with the number of processors.

The fall in speedup was due to the performance becoming absolutely limited by the merge step, with the addition of extra processors to the processor chain actually slowing down the transfer of information slightly. The merge step does not limit the smaller teapot scenes, which have large polygons and for which the merge step is not as large a part of the total cost. This problem of the merge step limiting performance may be trivially overcome by using a higher bandwidth merge channel or a tree

structured merge. Were this done, the results would be more similar to those of the z-buffer case, but would still not be as good due to the extra fixed cost of the merge step.

The growth of execution time with model size for the sixty-four processor, tetra case is more nearly linear than for the serial case, as shown in Figure 3·28. This corresponds to the increased importance of the fixed, linear costs as the other costs are shared amongst the many processors. The teapot case shows an interesting curve, (Figure 3·27). This is an artefact of the combination of the not seriously limited speedup for the smaller teapot scenes followed by the limited speedup of the larger scenes.

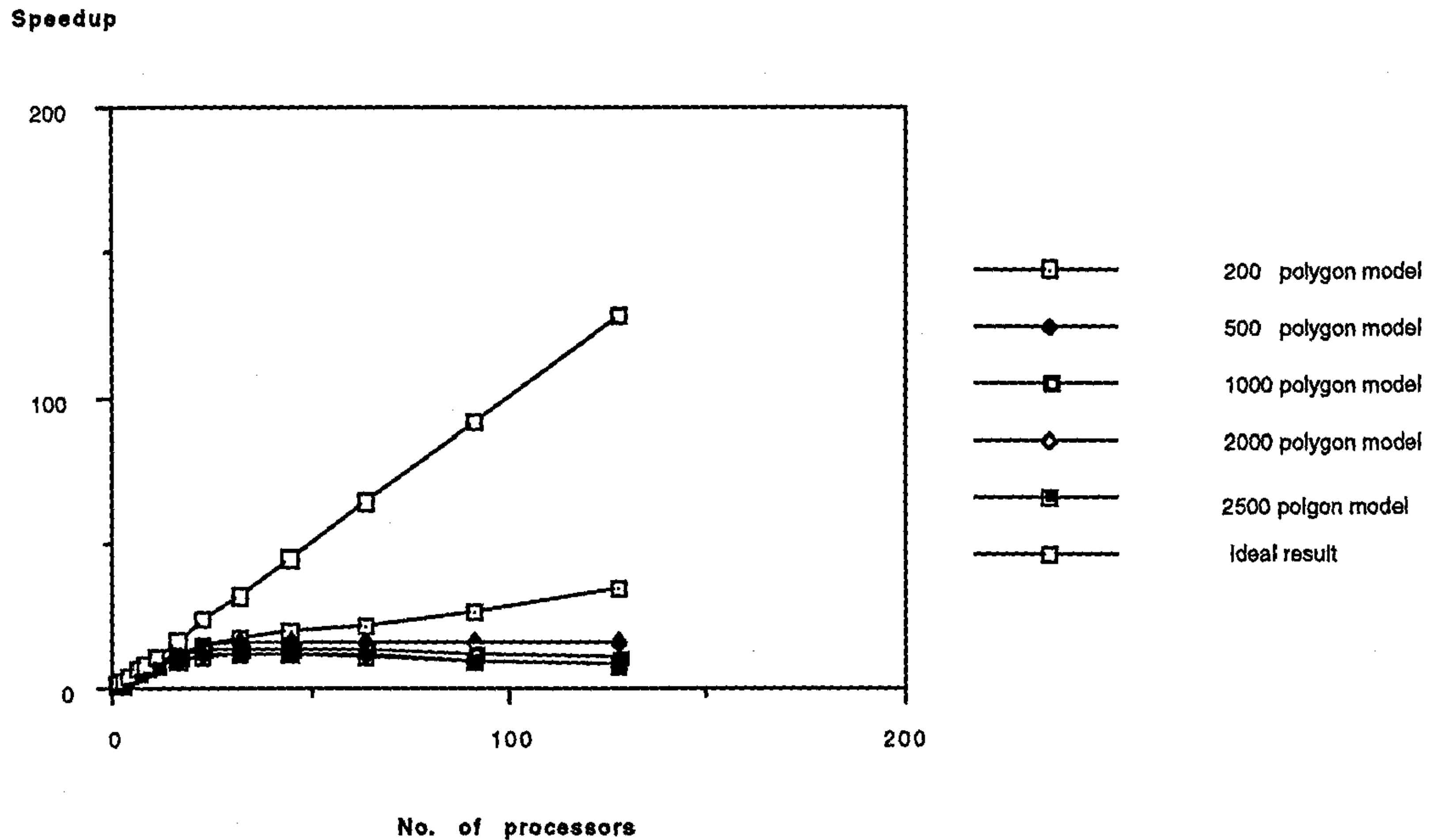


Figure 3·28: Speedup versus no. of processors for the painter's algorithm and teapot models.

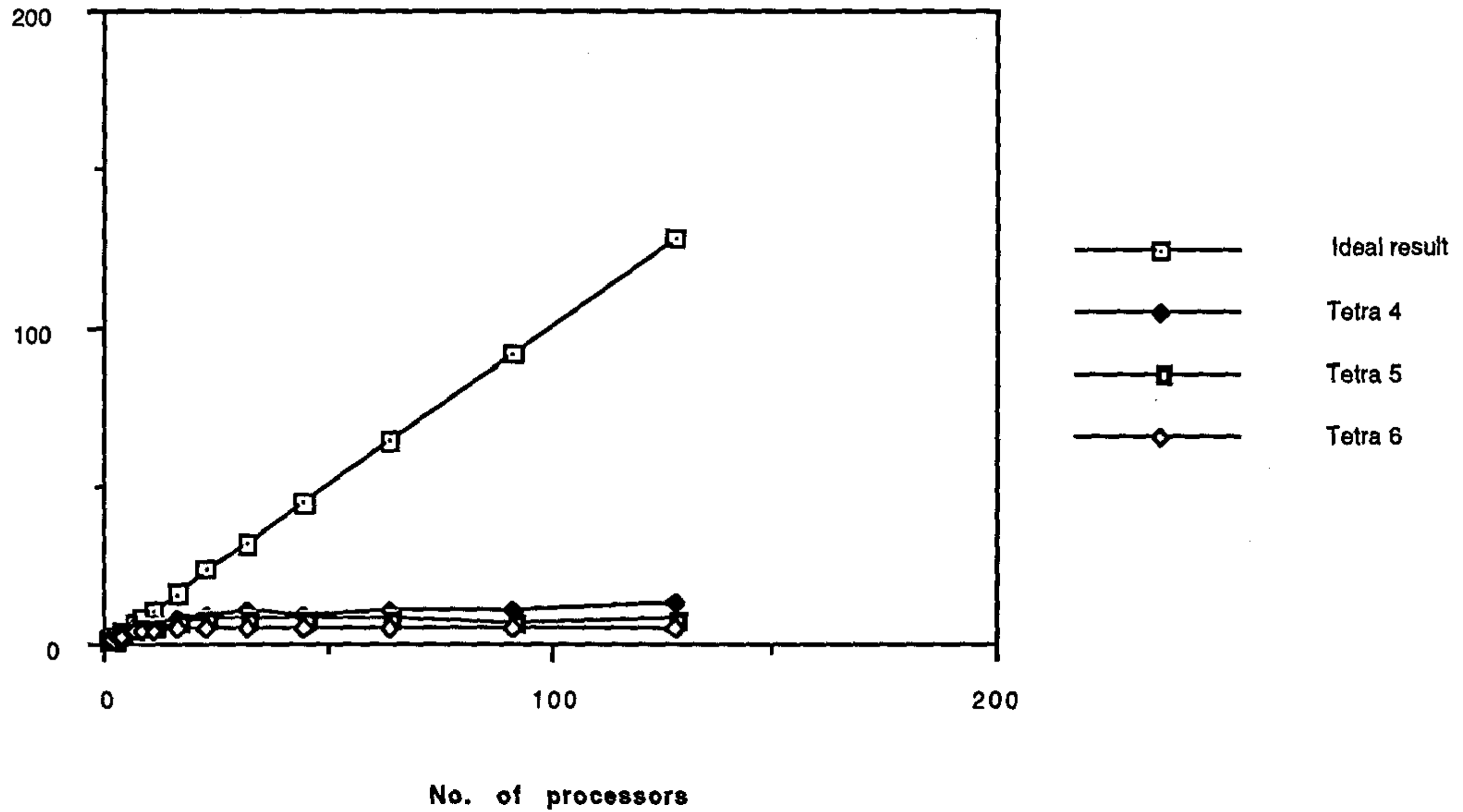


Figure 3·29: Speedup versus no. of processors for the painter's algorithm and tetra models.

Teapot model size	200	2500
Run time, 1 processor	194052	250232
Run time, 16 processors	16577	26967
Run time, 64 processors	9224	23350
Speedup	21.0	10.7
Linearity	0.33	0.17

Table 3·8. *Sample execution times and performance statistics for the painter's algorithm.*

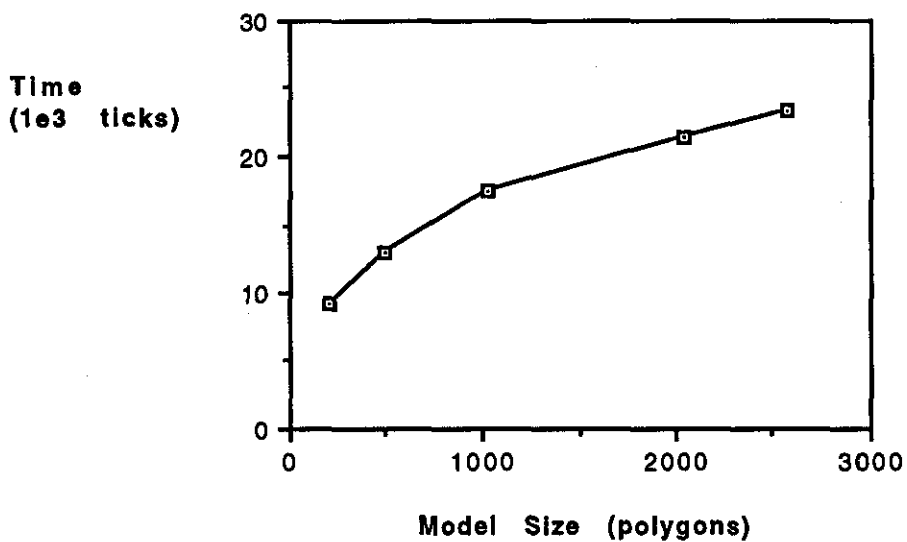


Figure 3·30: *Execution time versus model size for the parallel painter's algorithm and teapot models, using sixty-four processors.*

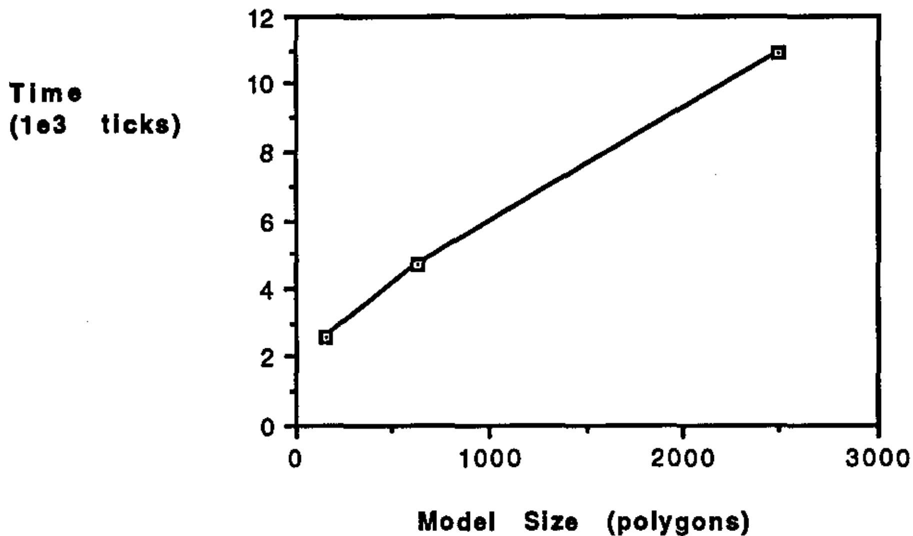


Figure 3·31: *Execution time versus model size for the parallel painter's algorithm and tetra models, using sixty-four processors.*

The painter's algorithm cost estimate derived earlier was

$O(F_T) + O(\text{buckets} + (F_T / N)) + O(F_T) + O\left(\frac{1}{N}\sqrt{n\ m\ D_C\ F_T}\right) + O(N)$ which, for a given set of test data, may be reduced to $A + \frac{B}{N} + CN$.

The CN term derives solely from the delay involved in filling the pipeline during the merge step. As discussed elsewhere this structure and therefore the cost could be significantly altered by altering the design of the communications network. Applying the cost estimate to the test cases and using regression to derive A, B and C gives table 3·9.

Again, the coefficients show no great predictability. However the cost versus the number of processors is very well predicted, even in the regions where the CN term becomes dominant and the performance reduces with increasing numbers of processors. Figure 3·32 shows an example where the CN term is not yet significant and figure 3·33 shows an example where the CN term results in the distinctive fall off of performance after about forty-five processors.

Scene	A (fraction serial)	B (fraction parallel)	C (fraction subserial)
Teapot 200	0.034865	0.965135	0.000000
Teapot 500	0.027207	0.972523	0.000270
Teapot 1000	0.025787	0.973640	0.000573
Teapot 2000	0.032883	0.966480	0.000637
Teapot 2500	0.037444	0.961926	0.000630
Tetra 4	0.075123	0.924861	0.000016
Tetra 5	0.089502	0.910100	0.000398
Tetra 6	0.140667	0.855653	0.000368

Table 3·9: *Regression produced coefficients for the theoretical estimate of the performance of the painter's algorithm.*

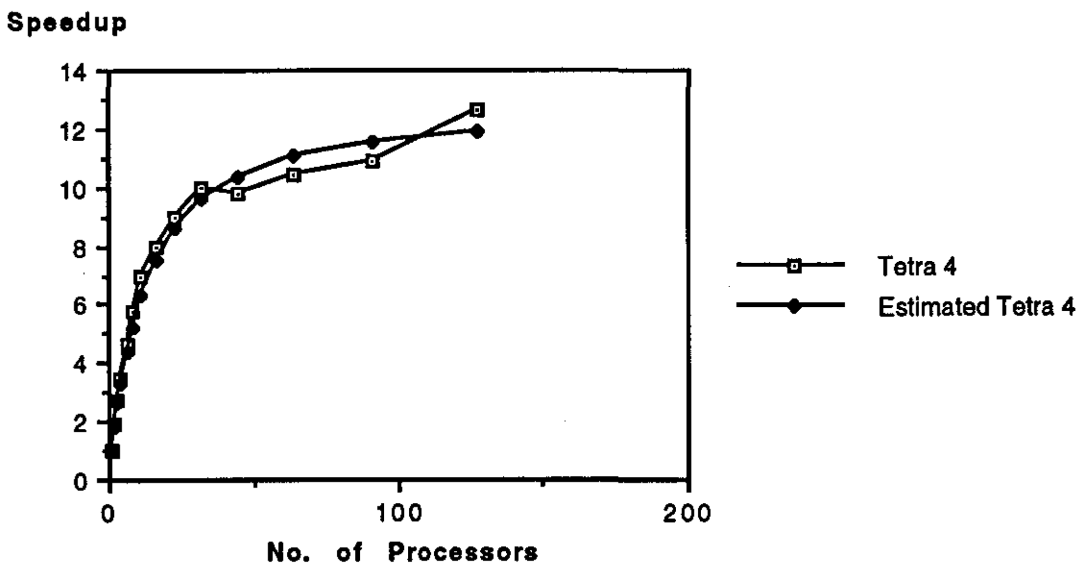


Figure 3·32. *Comparison of actual and expected performance, tetra 4 scene, painter's algorithm.*

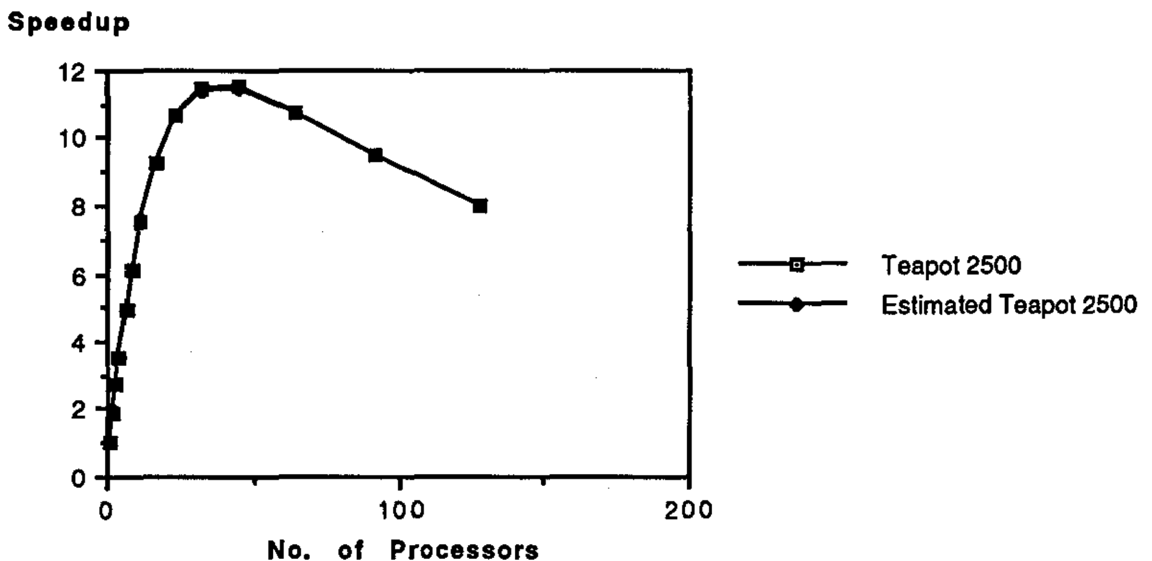


Figure 3·33. *Comparison of actual and expected performance, 2500 polygon teapot scene, painter's algorithm.*

3·5. Comparison of the Algorithms

The recursive subdivision algorithm showed useful gains in performance when parallelised, but was not particularly efficient in its use of the extra processors. It showed a speedup of between eight and twenty times for sixty-four processors.

The unoptimised scan line algorithm parallelised well, resulting in a speedup of between fifty and eighty times for one hundred and twenty-eight processors. Extrapolating these results suggests that this algorithm would still show good performance gains for even greater numbers of processors, although the algorithm is probably slowly approaching its limits for gains through increased parallelism.

The edge-table based scan line algorithm did not parallelise as well as the unoptimised version. It appeared to reach its limits of useful parallelism at around thirty or forty processors. When using up to twenty processors useful gains in performance were

realised, but there is little point in using more processors than this as the gains through adding processors quickly diminish.

The z-buffer algorithm parallelised well for small scenes but showed lower gains for large scenes. The speedups varied between ten and ninety for one hundred and twenty-eight processors. In some cases the algorithm appeared to hit its limits of useful parallelism at around sixty processors, although other cases showed only small signs of approaching such a limit. The limited cases were those which had many small polygons, so the irreducible per polygon overhead was more significant than for the other cases.

The painter's algorithm parallelised badly, with there being little point in using more than twenty processors to execute it. Although some of its limitations could be overcome, the remaining ones would still significantly limit the parallel algorithm's performance to below that of the z-buffer algorithm.

Measured Relative Cost - 1 Processor		
	Model Size (polygons)	
Algorithm	200	2500
Painter's	1.0	1.3
Z-Buffer	2.6	3.0
Scan Line (with Edge Tables)	0.8	14.8
Scan Line	1.1	18.3
Recursive Subdivision	6.3	25.5

Table 3·10. *Measured relative performance of the algorithms for the one processor case, (relative to the cost of the one processor, 200 polygon, painter's algorithm case).*

Measured Relative Cost - 16 Processors		
	Model Size (polygons)	
Algorithm	200	2500
Painter's	1.0	1.6
Z-Buffer	2.0	2.6
Scan Line (with Edge Tables)	1.0	18.0
Scan Line	0.9	14.3
Recursive Subdivision	12.8	49.9

Table 3·11. *Measured relative performance of the algorithms for the sixteen processor case, (relative to the cost of the sixteen processor, 200 polygon, painter's algorithm case).*

Measured Relative Cost - 64 Processors		
	Model Size (polygons)	
Algorithm	200	2500
Painter's	1.0	2.5
Z-Buffer	1.1	1.8
Scan Line (with Edge Tables)	0.9	18.3
Scan Line	0.5	7.8
Recursive Subdivision	7.6	44.9

Table 3·12. *Measured relative performance of the algorithms for the sixty-four processor case, (relative to the cost of the sixty-four processor, 200 polygon, painter's algorithm case).*

As may be seen in Tables 3·10, 3·11 and 3·12, the algorithms' relative performances do not alter greatly with the number of processors used, although there are several points worth noting. The painter's algorithm doubles in cost for the 2500 polygon case, (relative to the 200 polygon case) when moving from one processor to sixty-four processors. This is undoubtedly due to the merge sort step bottlenecking on the communications pipeline. This effect may thus be expected to grow in a linear fashion with model size.

The recursive subdivision algorithm for 2500 polygons slightly increases in cost compared to the 200 polygon case when moving to sixty-four processors due to bad load balancing in the parallel case - a consequence of the uneven complexity of the test scenes. The z-buffer algorithm performs just slightly better in this respect, and the two scan line algorithms show almost no change in the relative costs of the 2500 and 200 polygon scenes when moving from one to sixty-four processors.

In absolute terms the implemented algorithms performed reasonably well compared to existing graphics systems, particularly in view of their being generally unoptimised and that they were run on quite old hardware, (the T800 having been introduced in 1987).

3·6. Conclusions

The z-buffer algorithm consistently performed well, with execution times that were either the smallest or close to the smallest for all the test scenes. For test scenes of increasing size, this algorithm's execution time grew more slowly than any other and for the largest test scenes it was always the fastest full HSE algorithm tested. Only the painter's algorithm performed comparably, but the implementation of this tested did not provide the correct HSE solution for intersecting or interleaved polygons. When using many processors, the z-buffer algorithm was faster than even the painter's algorithm.

The z-buffer algorithm proves to be an even better choice of HSE algorithm for parallel execution than it did for serial execution.

Theoretical estimation of the HSE algorithms' performance variations with the number of processors used can clearly provide accurate predictions of the algorithms' real performance. It is also a valuable tool in understanding the underlying reasons for the algorithms' behaviour.

Chapter 4

Conclusions

4·1. Serial HSE Algorithms

The serial implementations of the hidden surface elimination algorithms examined in this thesis proved to perform in relative terms much as Sutherland et. al. estimated. The only major differences from their estimates were caused by their surprisingly high estimate of the basic painting operation for the z-buffer algorithm and by the author's choice of a different sort algorithm for the implementation of the painter's algorithm.

The algorithms' dependence on model size may be largely attributed to a combination of the sorting techniques used and some overhead costs. The cost of the z-buffer algorithm is almost entirely due to its painting process, giving it the favourable property of almost total independence of model size. Instead the z-buffer's cost changes with the depth complexity of the scene displayed. The cost of the painter's algorithm is mostly the cost of its main sort step. The implementation considered in this thesis used a low rate of growth bucket sort which gives the painter's algorithm a small linear dependence upon model size. The recursive subdivision algorithm's costs grow approximately as the square-root of model size for scenes with large polygons due to its two dimensional recursive division of screen area, but this becomes linear for scenes with small polygons where the per pixel sorting costs dominate. The unoptimised scan line algorithm's cost grows almost linearly with model size due to its collection of mostly linear growth operations. The optimised scan line algorithm's cost grows slightly more rapidly than the unoptimised version.

4·2. Parallel HSE Algorithms

The parallel z-buffer algorithm gained significant speedups, up to 90% of the maximum possible in some cases, but proved susceptible to much worse performance for models with small polygons. This was due to the unparallelisable per polygon overheads overtaking the parallelisable painting operations as the main cost component. This effect seriously limited the performance of the z-buffer in some cases.

The parallel recursive subdivision algorithm made limited gains from parallelism, never achieving more than about a quarter of the possible gains. This was almost totally due to bad load balancing. The algorithm's cost appeared to grow in a sub-linear fashion with model size for large numbers of processors, but this may be misleading since the bad load balancing probably swamps all other effects.

The parallel unoptimised scan line algorithm parallelised well, consistently reaching at least half of the maximum possible speedup. The parallel version's cost depends almost linearly upon model size, as for its serial ancestor. The parallel optimised scan line algorithm did not parallelise at all well, effectively hitting a performance limit at a speedup of between ten and twenty times. This was due to the optimisation destroying the algorithm's suitability for parallel implementation, by introducing unparallelisable steps.

The parallel painter's algorithm proved to be limited both by its per polygon overheads (as in the z-buffer case) and by its sorting step for all but the smallest models with the largest polygons.

For four of the five HSE algorithms, the performance improvements obtainable from parallelism were shown to be accurately predictable by theoretical means.

4·3. Overall Conclusions

For almost any HSE job where the output is to appear on a pixel type display, the z-buffer algorithm proves to be preferable to the other algorithms investigated. It provides a very good compromise solution with little dependence upon model size and high efficiency when parallelised. It was never much slower than the fastest HSE method in any of the tested cases.

This work has shown that hidden surface algorithms in general parallelise well and can with care be designed to make efficient use of a number of parallel processors, if adequate connections can be made between the processors and the frame buffer. A distributed frame buffer has been shown to both provide these connections and be well suited to the parallel HSE algorithms investigated.

This work has also shown that the performance of the parallel HSE algorithms investigated may be well predicted from their serial counterparts using theoretical means.

In absolute terms the implemented HSE algorithms performed reasonably well compared to existing graphics systems, particularly in view of their unoptimised state and that they were run on quite old hardware.

This work has also shown that a general purpose parallel computer may usefully be applied to near real time HSE.

References

1. I.E. Sutherland, R.F. Sproull and R.A. Schumacker, "A Characterisation of Ten Hidden Surface Algorithms", *Computing Surveys*, **6**(1), pp. 1 - 55 (1974).

2. B.J. Schacter, "Computer Image Generation For Flight Simulation", *IEEE Computer Graphics And Applications*, **1**(4), pp. 29-68 (1981).

3. J.K. Yan, "Advances In Computer Generated Images For Flight Simulation", *IEEE Computer Graphics And Applications*, **5**(8), pp. 37-51 (1985).

4. T.H. Myer and I.E. Sutherland, "On The Design Of Display Processors", *Communications Of The ACM*, **11**[6], pp. 410-414 (1968).

5. K. Akeley and T. Jermoluk, "High-Performance Polygon Rendering", *Computer Graphics*, **22**(4), pp. 239-246 (1988).

6. C.R. Priem, "Developing The GX Graphics Accelerator Architecture", *IEEE Micro*, **10**(1), pp. 44-54 (1990).

7. R.W. Swanson and L.J. Thayer, "A Fast Shaded-Polygon Renderer", *Computer Graphics*, **20**[4], pp. 95-101 (1986).

8. J. Clark, "A VLSI Geometry Processor For Graphics", *Computer*, **13**(7), pp. 59-69 (1980).

9. M. Deering, S. Winner, B. Schediwy, C. Duffy and N. Hunt, "The Triangle Processor And Normal Vector Shader: A VLSI System For High Performance Graphics", *Computer Graphics*, **22**[4], pp. 21-30 (1988).

10. N. Gharachorloo, S. Gupta, E. Hokenek, P. Balasubramanian, B. Bogholtz, C. Matieu and C. Zoulas, "Subnanosecond Pixel Rendering With Million Transistor Chips", *Computer Graphics*, **22**[4], pp. 41-49 (1988).

11. K. Guttag, J.Van Aken, M. Asal, "Requirements For A VLSI Graphics Processor", *IEEE Computer Graphics And Applications*, **6**(1), pp. 32-47 (1986).

12. B. Apgar, B. Bersack and A. Mammen, "A Display System For The Stellar Graphics Supercomputer Model GS1000", *Computer Graphics*, **22**(4), pp. 255-262 (1988).

13. B.S. Borden, "Graphics processing on A Graphics Supercomputer", *IEEE Computer Graphics And Applications*, **9**(4), pp. 56-62 (1989).

14. D. Kirk and D. Voorhies, "The Rendering Architecture Of The DN10000VS", *Computer Graphics*, **24**[4], pp. 299-307 (1990).

15. H. Gouraud, "Continuous Shading Of Curved Surfaces", *IEEE Transactions On Computers*, **C20**[6], pp. 623-629 (1971).

16. B.T. Phong, "Illumination For Computer Generated Pictures", *Communications Of The ACM*, **18**[6], pp. 311-317 (1975).

17. G. Bishop and D.M. Weimer, "Fast Phong Shading", *Computer Graphics*, **20**[4], pp. 103-106 (1986).

18. A. Appel, "Some Techniques For Shading Machine Renderings Of Solids", *AFIPS 1968 Spring Joint Computer Conference*, pp. 37-45, Atlantic City, New Jersey (1968).

19. T. Whitted, "An Improved Illumination Model For Shaded Display", *Communications Of The ACM* , **23**[6], pp. 343-349 (1980).

20. R. Pulleybank and J. Kapenga, "The Feasibility Of A VLSI Chip For Ray Tracing Bicubic Patches", *IEEE Computer Graphics And Applications*, **7**(3), pp. 33-44 (1987).

21. J.-D. Nicoud, "Video RAMs: Structure And Applications", *IEEE Micro*, **8**(1), pp. 8-27 (1988).

22. H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs and L. Israel, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor Enhanced Memories", *Computer Graphics*, **23**[3], pp. 79-88 (1989).

23. J. Goldfeather, J.P.M. Hultquist and H. Fuchs, "Fast Constructive Solid Geometry Display In The Pixel-Powers Graphics System", *Computer Graphics*, **20**[4], pp. 107-116 (1986).

24. R. Peterson, C.R. Killebrew, T. Albers and G. Guttag, "Taking The Wraps Off The 34020", *Byte*, **13**[9], pp. 257-272 (1988).

25. J. Grimes, L. Kohn and R. Bharadhwaj, "The Intel i860 64-bit Processor: A General-Purpose CPU With 3D Graphics Capailities", *Computer Graphics And Applications*, **9**(4), pp. 85-94 (1989).

26. N. Margulis, "The Intel 80860", *Byte*, **14**[13], pp. 333-340 (1989).

27. J. Pineada, "A Parallel Algorithm For Polygon Rasterization", *Computer Graphics*, **22**[4], pp. 17-20 (1988).

28. F.C Crow, "Parallelism In Rendering Algorithms." *Graphics Interface*, pp. 87-96 (1988).

29. W.R. Franklin and M. Kankanhalli, "Parallel Object Space Hidden Surface Removal", *Computer Graphics*, **24**[4], pp. 87-94 (1990).

30. E. Fiume, A. Fournier and L. Rudolph, "A Parallel Scan Conversion Algorithm With Anti-Aliasing For A General-Purpose Ultracomputer", *Computer Graphics*, **17**[3], pp. 141-150 (1983).

31. M.-C. Hu, and J.D. Foley, "Parallel Processing Approaches To Hidden-Surface Removal In Image Space", *Comput. & Graphics*, **9**[3], pp. 303-317 (1985).

32. T. Strothotte and B. Funt, "Raster Display Of A Rotating Object Using Parallel Processing", *Computer Graphics Forum 2*, North-Holland, pp. 209-217 (1983).

33. F.I. Parke, "Simulation and Expected Performance Analysis of Multiple Processor Z-Buffer Systems", *Computer Graphics*, **14**[3], pp. 48-56 (1980).

34. W.J. Bouknight, "A Procedure For Generation Of Three-Dimensional Half-Toned Computer Graphics Presentations", *Communications Of The ACM*, **13**[9], pp. 527-536 (1970).

35. G.S. Watkins, "A Real-Time Visible Surface Algorithm", Technical Report UTECH-CSC-70-101, Department Of Computer Science, University Of Utah.

36. C. Wylie, G.W. Romney, D.C. Evans and A. Erdahl, "Halftone Perspective Drawings By Computer", *Proceedings AFIPS Fall Joint Computer Comference 1967*, pp. 49-58, Anaheim, California (1967).

37. M.J. Flynn, "Very High-Speed Computing Systems", *Proceedings Of The IEEE*, **54**[12], pp. 1901-1909 (1966).

38. INMOS, *OCCAM 2 Reference Manual*, Prentice-Hall, (1987).

39. C.A.R. Hoare, "Communicating Sequential Processes", *Communications Of The ACM*, **21**[8], pp. 666-677 (1978).

40. INMOS, *Transputer Databook*, 2nd. Ed., Prentice-Hall, (1989).

41. INMOS, *The T9000 Transputer Products Overview*, SGS-Thompson Microelectronics Group, 1991.

42. Texas Instruments, *TMS320C4x User's Guide*, 1991.

43. J. Packer, "Exploiting Concurrency: A Ray Tracing Example", *INMOS Technical Note 7*. Also in INMOS, *Communicating Process Architectures*, Prentice Hall, (1988).

44. H.E. Bez, "On Parallel Scan-Conversion Algorithms for Transputer Networks", *Journal of Microcomputer Applications*, **13**, pp. 43-55 (1990).

45. E. Haines, "A Proposal for Standard Graphics Environments", *IEEE Computer Graphics And Applications*, **7**(11), pp. 3-5 (1987).

46. A.V. Aho, J.E. Hopcroft and J.D. Ullman, Data Structures and Algorithms, *Addison-Wesley*, (1983).

47. J.D. Foley, A. van Dam, S.K. Feiner and J.F. Hughes, *Computer Graphics - Principles and Practice* (Systems Programming Series), Addison-Wesley, (1990).

48. I.E. Sutherland, R.F. Sproull and R.A. Schumacker, "Sorting And The Hidden Surface Problem", *National Computer Conference 1973*, pp. 685-693, New York (1973).

49. R. M. Dixon, J. R. Vaughan, and G. R. Brookes, "Timing Analysis for Processor Farm Environments", *Microprocessors and Microsystems*, **15** (7), pp. 355-358 (1991).

50. G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities", Proc. AFIPS Spring Joint Computer Conference 30, pp. 483-485, Atlantic City, New Jersey (1967).

Appendix

Program Timings

The following tables provide the full results referred to in this thesis. They include the measured execution times of the HSE algorithms, stated in ticks of the transputer's low priority timer, (15625 ticks per second).

No. of Processors	Teapot 200	Teapot 500	Teapot 1000	Teapot 2000	Teapot 2500
1	1222352	2055422	2672128	4210430	4947562
4	592243	703353	985405	1510597	1779768
16	211406	256187	348863	705486	827771
64	69794	131977	241436	361502	413791

Table A·1. *Execution times of the recursive subdivision algorithm for the teapot scenes.*

No. of Processors	Tetra 4 (156)	Tetra 5 (624)	Tetra 6 (2496)
1	385115	784343	1528651
4	128521	252244	485449
16	122201	236158	445852
64	47771	87752	157246

Table A·2. *Execution times of the recursive subdivision algorithm for the tetra scenes.*

No. of Processors	Teapot 200	Teapot 500	Teapot 1000	Teapot 2000	Teapot 2500
1	208212	652933	1255844	2651283	3559407
2	104665	328671	630300	1332546	1786348
3	70645	220190	422975	895441	1198955
4	53287	166158	319729	672808	899815
6	36051	112417	214333	452265	608088
8	27306	85167	162616	345348	457518
11	20246	62696	120821	253379	339697
16	14597	44657	85700	177710	236438
23	10521	31954	62383	127840	173644
32	7980	24182	46346	94994	128768
45	6129	18262	36288	74850	100568
64	4803	13761	27032	52974	71955
91	3916	11598	22733	46303	62827
128	3121	9240	18044	34239	47266

Table A·3. *Execution times of the unoptimised scan line algorithm for the teapot scenes.*

No. of Processors	Tetra 4 (156)	Tetra 5 (624)	Tetra 6 (2496)
1	96931	363791	1403996
2	48913	183066	706769
3	32878	123267	474418
4	24947	93030	358186
6	16883	62816	242055
8	12837	47805	184205
11	9697	35722	136387
16	6929	25134	97434
23	5093	18424	71396
32	3922	14288	55424
45	3022	11362	44058
64	2447	8889	35402
91	2170	7571	29852
128	1824	6647	27211

Table A·4. *Execution times of the unoptimised scan line algorithm for the tetra scenes.*

No. of Processors	Teapot 200	Teapot 500	Teapot 1000	Teapot 2000	Teapot 2500
1	150692	483806	897526	2069488	2880572
2	78517	251021	466711	1080821	1501884
3	54803	172987	325956	756335	1044789
4	42925	134745	254319	585013	812650
6	30618	94995	181728	421279	584695
8	24778	76648	146886	342175	469237
11	19774	60115	116819	273510	375091
16	15881	47060	91770	215091	297713
23	12874	37815	79935	180562	249822
32	11145	32847	66262	156104	212589
45	9926	28537	61743	139615	193424
64	8967	25527	53266	122312	168752
91	8553	23835	51083	117175	161673
128	7919	22362	47826	107994	151000

Table A·5. *Execution times of the optimised scan line algorithm for the teapot scenes.*

No. of Processors	Tetra 4 (156)	Tetra 5 (624)	Tetra 6 (2496)
1	44162	146107	568946
2	23678	78507	314369
3	16775	56294	230254
4	13460	45072	188075
6	9959	34020	146185
8	8249	28210	125227
11	6971	23905	107978
16	5741	19749	93869
23	4872	17100	85421
32	4476	15856	79399
45	4117	15532	77372
64	3867	14137	74063
91	3953	14472	74276
128	3760	13940	73428

Table A·6. *Execution times of the optimised scan line algorithm for the tetra scenes.*

No. of Processors	Teapot 200	Teapot 500	Teapot 1000	Teapot 2000	Teapot 2500
1	509952	531721	539235	565572	581151
2	255528	266786	271415	286046	294902
3	171332	178371	182643	192996	199287
4	128214	134217	137912	146327	151524
6	86383	90055	93091	99717	103730
8	65084	67930	70847	76432	79875
11	47766	49881	52482	37437	60038
16	33698	34966	37138	41194	43602
23	23445	24732	27068	30667	32826
32	17980	18297	20128	23588	25660
45	13099	13443	15512	18829	20487
64	10243	10066	11785	14680	16323
91	7560	7784	9503	12047	13714
128	5984	5947	7489	10227	11707

Table A·7. *Execution times of the z-buffer algorithm for the teapot scenes.*

No. of Processors	Tetra 4 (156)	Tetra 5 (624)	Tetra 6 (2496)
1	60846	73994	104037
2	30651	37848	55301
3	20580	25786	39071
4	15578	19776	30914
6	10521	13732	22822
8	8031	10773	18833
11	5965	8328	15472
16	4244	6208	12406
23	3093	4827	10336
32	2403	4003	9200
45	1932	3427	8333
64	1462	2771	7415
91	1418	2767	7381
128	1160	2422	6991

Table A·8. *Execution times of the z-buffer algorithm for the tetra scenes.*

No. of Processors	Teapot 200	Teapot 500	Teapot 1000	Teapot 2000	Teapot 2500
1	194052	207781	216999	238060	250232
2	98030	105365	110840	123025	130205
3	66465	71283	75903	84904	90181
4	50293	54357	58350	65762	70281
6	34962	37609	40909	46891	50539
8	27222	29353	32485	37897	40814
11	21379	22836	25847	30476	33082
16	16577	17749	20349	24558	26967
23	13352	14859	17297	21194	23427
32	11664	13509	16149	19842	21893
45	9906	13578	16211	19789	21737
64	9224	12870	17401	21310	23350
91	7313	13321	19236	24361	26451
128	5807	13494	22124	28502	31243

Table A·9. *Execution times of the painter's algorithm for the teapot scenes.*

No. of Processors	Tetra 4 (156)	Tetra 5 (624)	Tetra 6 (2496)
1	27166	36167	59161
2	14226	19417	33563
3	9935	13866	25197
4	7803	11087	20936
6	5781	8454	16882
8	4733	7131	14948
11	3888	6188	13305
16	3416	5410	11933
23	3038	4896	11176
32	2736	4806	10719
45	2780	4785	10684
64	2609	4716	10881
91	2499	5161	11428
128	2156	5003	11295

Table A·10. *Execution times of the painter's algorithm for the tetra scenes.*