Real-Time Parallel OpenGL Applications on Compute Clusters

AnneMarie Walsh

B.A. (Mod.) Computer Science

Final Year Project May 2003

Supervisor: Michael Manzke

Acknowledgements

I would like to extend my appreciation to my supervisor, Michael Manzke, for his constant support and patience. The project is based on Chromium, and hence I acknowledge the great work of its producers at Stanford University Computer Graphics Lab. My friends and family have provided invaluable support throughout this project - in particular Charles Smith, ever helpful and tolerant, Julian Carroll, grammar-expert, and Catherine Quinlan - thank you all.

Produced with LaTeX.

Abstract

This project involves testing an interactive rendering system, Chromium, on a cluster of machines in real-time, using serial and parallel OpenGL applications. It draws together hardware and software systems in an effort to find a viable alternative to expensive custom-built supercomputers, the traditional system required to render intensive graphical applications. Exploiting its aggregate optimised graphics hardware and processing power, Chromium enables parallel applications to execute interactively across a cluster. ROAM, Real-time Optimally Adapting Meshes, is the terrain generation algorithm used for the majority of the testing applications designed. Serial and parallel versions of this program are implemented and their performance with and without Chromium evaluated. The improvement in the frame rate of the applications suggests that Chromium, running on a modest-sized cluster, is a worthwhile consideration for rendering demanding graphical applications in real-time.

Contents

List of Figures

Chapter 1

Introduction

Constantly expanding research fields, industrial and academic, in both hardware and software are actively exploring all avenues in an attempt to address the increasing demand for rendering power in computing. A plethora of techniques are under exploration in an effort to increase rendering and computational capabilities, employ available hardware and software more efficiently and reduce unnecessary and redundant rendering. It is into this category of research this project falls.

Brookshear [6] emphasises that computer science is a combination of theoretical research and rapidly progressing technology, where they coexist in a mutually beneficial relationship, each influencing the other. From this perspective, this project really does embody the ideas behind computer science, as it combines existing technology with new and current research areas, the theoretical findings not yet fully realised in practise. The goal of the project

is to substantiate the theories behind improving the performance of applications by running them concurrently on a graphics rendering system over a cluster of machines. It amalgamates a number of well-documented areas of efficient resource usage with new and experimental fields. In its own way, it is pioneer work, which has yet to be successfully realised.

1.1 Objective of the Project

The motivation for the project was to evaluate the performance of a scalable, interactive rendering system using graphical applications upon a compute cluster.

Interactive frame rates are still not being achieved by many real-time graphical applications in spite of the development of hardware accelerator technology. Even as memory and processing power increases, so too does the size and complexity of the data sets to be visualised. As far back as 1997, AAB Engineering modelled a 'coarsely tessellated model' of a coal-firing plant, comprising over 13,000,000 triangles - current laser range scans contain billions of polygons. Solutions to fluid mechanics and dynamics problems require several hundred million data points per frame, over thousands of frames. Up to quite recently, computing and rendering such models in real-time could only reasonably be executed on custom-built supercomputers.

Compute clusters are an evolving, and increasingly popular, architecture due to the cost-effective approach they offer for a wide range of intensive

applications requiring multiple node systems, as well as managing complex graphics with limited resources. Defined as *'a group of interconnected whole computers working together as a unified computing resource'* **??**, clusters are basically a group of off-the-shelf, inexpensive components loosely-coupled together, giving the illusion of being a single machine. They emulate the computing and rendering power of a custom-built, high-end parallel system, but are much cheaper, highly scalable and easy to upgrade. Cluster technology advances, such as optimised graphics accelerators, improved processing power, and high-speed memory has drastically improved their performance. The efficiency and scalability of the cluster are highly dependant upon the interconnect used between the nodes. Networks capable of routing the streams of graphical commands at acceptable speeds have led to a competitive alternative to these high-end parallel systems[13].

While this recent sophistication of clusters, described in some depth in *Section 2.2*, has offered very promising performance, there are still many applications, such as large scale scientific problem solving programs and simulators, that are unable to be executed in real-time upon them. Methods to increase the computational and rendering power of clusters have led to the development of rendering display systems. These systems have been designed and developed to increase the accessibility and abstraction of the underlying parallel hardware by exploiting the graphics accelerators in each of the processors in the cluster.

For reasons later expounded upon in *Chapter 2, section 2.2.3*, Chromium was the chosen graphics rendering system. *'Chromium makes it possible to visualise datasets and run applications that would not be able to run on a single workstations'* [11]. Chromium, an evolution of WireGL, is a relatively recent system developed to enable streams of graphical API commands to be intercepted and manipulated on clusters of processors. A completely extensible architecture, it provides a general mechanism for clusters to run interactive graphical applications. As a rendering display system, it takes advantage of the graphics hardware optimisations of each of the nodes, as well as exploiting the aggregate processing power of all of them. We concentrate on the support that Chromium offers for parallelising applications across a cluster, which is our principal aim in this project. Multiple functions can be performed by each processor on a stream of graphical commands through Stream Processing Units, SPUs, which will be expanded in the next chapter.

OpenGL, a graphics API based on C++, is ubiquitous in graphics and was an obvious choice for implementing the testing applications since Chromium was oriented to support OpenGL specifically. OpenGL is impressively fast, enabling two- and three-dimensional features to be rendered at interactive frame-rates. This becomes particularly relevant if hardware optimisations are available, a number of which Chromium was designed to exploit. Its ordered semantics and familiarity were other key reasons for its choice. A variety of graphical applications was researched and considered, their scala-

bility and the potential divisibility of their algorithms being decisive factors, in addition to the size of the input and the preprocessing time required.

Fractals and Terrain Generation were two areas that were dedicated particular attention; terrain visualisation finally being given preference for the bulk of the project. This field of graphics has been attracting much attention of late as it is only in the past five or six years that realistic and believable terrains have been modelledin real-time. The enormous computational and rendering requirements makes this an ideal prototype for parallelisation. Some of the diverse algorithms that were considered for implementation are outlined in *Section 2.3.2* , and the eventual choice, Real-time Optimally Adaptive Meshes (ROAM), is analysed in greater depth.

The implementation of ROAM on the cluster incorporating Chromium, and, in particular, parallelising the algorithm and enabling coherent communication between concurrent processes comprised the major part of the practical programming work on the cluster. Many sample and test programs had to be developed throughout this stage to fully comprehend different aspect of the intertwined systems and their interaction with one another. To acquire a realistic reading and understanding of the difference in performance, it was necessary to evaluate the system with numerous test programs, each with various modifications on the basic algorithm. These included serial and a variety of distinct parallelised versions, run both locally on a single node, and across the cluster of three processors. Chromium's inbuilt Stream Processing

Units, SPUs, which accommodate parallelisation, were implemented, as well as a number of manually parallelised versions of code. The programs were run both directly on Linux and using Chromium, some versions containing additional message passing to synchronise events. Separate tests determined the overhead communication costs associated with Chromium's own message synchronisation techniques and the extra message passing functions,(*MPI*) incorporated in the code. The results of extensive testing are included in *Chapter 4*, along with an analysis of their significance.

The resultant project still has scope for expansion, as discussed in the *Future Work* section concluding this report. These include: utilising features of Chromium currently under development, such as CRUT; further optimisations in both hardware and software, varying the size and interconnect of the cluster; upgrading components such as the graphics cards; most particularly, realising a practical application of the cluster, a CAVE.

1.2 Personal Motivation

From a learning perspective, this was a fantastic project to accept as I knew practically nothing about the area previous to its undertaking. With an interest in increasing the efficiency of computing power for graphical applications, coupled with a brief introduction to Linux from third year, and a preamble into OpenGL this year to assist me, the bulk of the project was uncharted territory. The amount of research alone, before any practical cod-

ing was implemented, made this a challenging project from the outset.

The principle disadvantage in taking on something this novel is, of course, the lack of documentation and help available. Hours and hours were spent trying to figure out how different systems interacted or how a line of code should be defined before use - things that could normally be checked rapidly on the Internet or located in documentation. From this aspect it took far longer than anticipated to fully grasp and then implement many aspects of the project, but because everything had to be dissected and built-up piecemeal, comprehension of how things actually worked was far greater than if the answers had been more readily available.

The approach taken to completing the project was first and foremost a serious commitment to research. A lack of familiarity with most of the areas the project draws upon meant that a vast amount of study and experimentation was necessary to grasp the import of the central aspects of it. The initial research concentrated on the main aim of the project: testing a rendering system across a cluster using graphical applications, including the principal factors influencing these, and how they were best achieved. The hardware implementation of a cluster was an obvious starting point since this was the architecture upon which the project was executed. A comprehension of the advantages of cluster configuration over its competitors, how the nodes in the system were connected together, and how communication takes place between them was an important base. Gaining a deepened understanding of

how graphical applications actually work on a more primitive level, as well as learning OpenGL, GLUT *(OpenGL's Utility Tollkit which allows user interaction)* and how their various libraries interact was a time-consuming element of the research. Associated with this was a thorough grounding in Linux and the editor vim, *vi improved*, in which both programming and this report are written, as well as additional features of Linux such as CVS, a current version system repository.

Regarding the progression of the project: while more concurrent than linear, researching the different rendering display systems occupied some time, and a variety are introduced before an in-depth discussion on the final choice, Chromium, is launched into. Studying the underlying theories and experimenting with Chromium's features was an ongoing part of the project, which lasted most of its life-cycle. Choosing the methods most appropriate to testing Chromium and, within that scope, which particular algorithms to implement involved another significant milestone. A crucial and fundamental element of the project was parallelising the application. This involved innumerable different attempts and much research into alternative approaches. The most significant are outlined in *Chapter 3, section 3.2*. Eventual performance testing was a focal part of the project, and a huge amount of testing was carried out as the final part of the practical work of the project. The evaluation and significance of these results are also expounded upon in the final chapter. The CVS repository of the code can be explored in the attached CD.

Chapter 2

Background

From the day the earliest computing machines were physically realised, people have been working on ways of improving their performance. With an ever-increasing demand for greater computing and rendering power, it seems that a stabilising of the supply-demand relationship will never be achieved. Yet many have dedicated their life to attempting to address this very issue.

With the visualisation of larger and more complex models, the demand for increasingly powerful computing is ever-growing. With many applications in educational, medical, commercial, scientific and military fields, as well as the enormous reliance of entertainment upon them, more versatile tactics to obtaining the best possible performance from machines at feasible costs are necessary. On the hardware side, we have seen many drastic and revolutionary optimisation techniques involving novel approaches implemented. The Scalable Rendering Systems, introduced in the previous chapter, abstracts

the hardware and improves upon the performance of the cluster. A number of these systems and their particular features will be examined later in this chapter. Moving further into developments in software optimisation techniques, we look at different methods to improve the performance of these algorithms, in particular examining terrain visualisation, and more specifically ROAM, Real-time Optimally Adapting Meshes, as the main testing application of the project.

2.1 Improving Hardware Performance

Conventional Von Neumann machines are referred to as 'control flow' computers due to the sequential manner in which the instructions are executed. This is inherently slow, and has led to a substantial research into increasing the throughput offered by such machines. Innumerable alternatives to this hardware implementation are the areas of research and design of countless members of academia and industry alike. Improving the performance of hardware designs generally focuses in on five areas of interest: throughput of the system (the number of instructions executed per unit time), Reliability, Flexibility, Availability and Scalability. The principle aim is to eradicate bottlenecks and to increase the total system throughput.

Parallelism, in both hardware and software, has long been recognised as a vital method of combating the under-performance of resources, rather than the conventional sequential approach. The obvious advantages of parallelising include more efficient use of these resources, as well as faster execution

times. Basically, the idea is to decompose a large application into a number of smaller, more manageable parallel tasks, and then running these concurrently. This is a more efficient method of processing data, the philosophy of which has inundated a huge range of different research fields. Running concurrent activities in the architecture is a cost-effective method of improving the throughput, and different computer architectures have been categorised by their degree of parallelism by Tse-yun Feng [16]. The maximum parallelism degree, P, is the maximum number of bits that can be processed within a unit time.

Pipelining is often viewed as one of the first steps to parallelisation. This technique allows the overlapping of instruction steps in the machine cycle. From this rather basic stepping stone, many other implementations have developed that increase the concurrency of computing systems, from time-sharing and multi-tasking, to the stage where multiprocessor systems have flooded the market. Designed to exploit parallelism to the maximum, data flow machines work by enabling an instruction to be executed as soon its required operands become available, as opposed to awaiting a program counter's permission. This is a radically different approach to that of the traditional von Neumann computers. Other designs embrace artificial neural networks as a manner of computing vast volumes of data that are too taxing for current systems. Interactive systems, where the activities occurring in the machine must be coordinated with those of the machine's environment, known as real-time processing, are becoming more significant as users' interaction with a

process increases in importance.

Under the aegis of maximising the potential of computers today, distributed systems are gaining increasing popularity and momentum in academia, industry and commerce. Our exploration delves into loosely-coupled systems, one of the two main branches of distributed systems, the other being tightly-coupled systems. Loosely-coupled, or distributed multiprocessors, are particularly suited to applications with minimal task interaction[16].

2.1.1 Clusters

Clusters are highly scalable, with the amalgamated processing, memory and bandwidth capacity increasing linearly with the number of machines in it, see *Figure 2.1.*

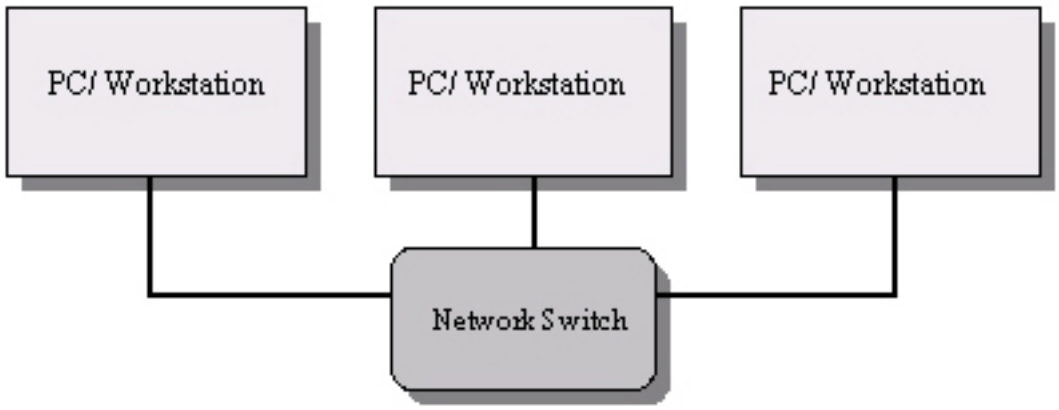

Figure 2.1: Cluster of PCs

Overhead communication costs, interface, compute, graphics and resolution

bottlenecks sometimes make this linear growth difficult to actually realise. The sophistication of PC graphics cards and accessible high-speed networks has experienced a dramatic escalation in recent years, and has heralded the development of clusters of PCs across which the rendering of the complex models are distributed [8]. This means the cluster can take advantage of the frequent upgrades of these components for each machine, improving their performance and power to realise such models at greater speeds, on a regular basis.

Where custom-built, extremely expensive high-end parallel machines were required for the rendering of highly complex models, the far more attractively priced PC cluster is becoming a viable alternative. Cluster computing combines the best features of both the network-based and parallel computer, resulting in powerful and, importantly, scalable systems, at competitive prices. Various considerations, such as Ethernet, point-to-point or SCI (Scalable Coherent Interface) interconnections are all fields of investigation where the chosen network, in particular, is an essential design choices. Connecting the processors in this manner enhances their cooperative ability to work on shared data with relatively low communication overhead. Clusters are very flexible, with non-problematic addition and removal of processors and they can manage a number of different types of tasks (not specialised), and can even be run on a variety of different machine types. The rate of improving off-the-shelf components tends to be much higher than that for custom-built hardware. With new, inexpensive graphics cards becoming available on a

six-to-twelve month basis, the cluster can easily be upgraded on a regularly without affecting the other elements of the cluster. Add to this a far more competitive price-performance ratio than high end machines, and it becomes apparent that the market for specialised supercomputers will surely dwindle.

This project was implemented on a cluster of three processors, Pentium II 450 MHz machines, each running Red Hat Linux version 7.3. Linux was an appropriate choice as it really allows one the freedom to fully explore an operating system, and permits full control over it. This encourages a far greater understanding of how the system works, supplemented with the freely available documentation.

2.2 Rendering Display Systems

To improve the performance of graphical applications on a cluster, an idea emerged to execute them on a software parallel rendering system that exploits the power and graphics hardware offered by the cluster, while achieving the same interactive frame rates obtainable from supercomputers. Interactive frame rates are generally deemed to be above 10 to 20 frames per second. The enormous three-dimensional models now being modelled require huge rendering power, and achieving realistic looking output in real-time is proving a significant challenge to computer graphics developers. Parallelising these loads is an obvious approach to realising these applications at interactive frame rates. By merging the rendering power of all the graphics accelerators in the cluster, a graphics rendering system provides a virtualised interface

to the graphics hardware through the OpenGL API. This uses immediate-mode semantics which mean it allows time-varying data to be visualised far easier than by using retain-mode or scene-graph APIs. However, some effort at storing display lists and texture objects is made, which is sustained by Chromium, by saving this data on the server for reuse. A number of Rendering Display Systems are currently under development. Some investigation was required to find a working version that supported high-resolution image rendering on a cluster-based tiled display; particularly as it had to meet the challenge of doing so in real-time, with performance comparable to that of high-end specialised parallel systems. Outlined below are a number of those explored.

2.2.1 Systems Considered

On researching various software packages for tiled displays, a number suggested themselves as potential choices . Syzygy, Aura, VIRPI and, the eventual choice - Chromium, are all outlined below. The aim was to find a suitable system for rendering graphical applications on a tiled display system. The obvious disadvantage to these is the fact they are still at a rather experimental stage of development, and many require the program to be rewritten in their own particular API.

Syzygy [5] was designed to enable a broad range of Visual Reality applications (traditionally run on SGI Onyx) to be run on a PC cluster, with optimal performance. It also focused on allowing cluster-based applications to run

between different architectures. A product of *vrsource.org* in the University of Illinois, Syzygy was designed to provide a set of lightweight tools for building heterogeneous distributed systems. It attempts to group all the VR and OpenGL code and libraries together and run it over a cluster of processors. Unfortunately, it never really took off since it didn't work very efficiently in practise, but is still under development. Another exploratory system in this area was **A**ura, a WireGL type application which requires the user to learn another set of calls, separate to OpenGL.

Having rewritten all the OpenGL programs in the low level Aura language, **A**ura then ports the graphics using different processors. The necessity of rewriting all the programs nullifies any performance advantages Aura has to offer, which eliminated it as a choice of tiled rendering system.

VIRPI is a high-level program which implements Aura, acting on top of it as such. Boasting an ease of programming equal to that of C++, though this is yet in a rudimentary stage of development, it is anticipated that VIRPI will be popular with amateur programmers playing around with virtual reality capabilities.

Argonne's Tiled Display(TD) is a graphics library focused on rendering GLUT applications on a tiled display. It uses MPI *(Message Passing Interface)* barrier calls to keep the animations synchronised in parallel GLUT applications. The viewing commands are redefined with TD substitutes. The output is sent to each node in the display, with its rendering split in accordance with the MPI-appointed identification number. Handling user

interaction was the main stumbling block for TD. Poor performance resulted from separate window catch-calls - especially with mouse interaction. Mouse events generate a vast quantity of tiny packets of data which prove an unsuitable format for MPI, since it involves huge communication costs. It also didn't support a number of expected and quite basic things, such as random number generation. In its favour, since each process runs locally, TD suffers no overhead from network transmission and unpacking data, which means attractive performance rates. Nonetheless, this system is a considerably less invested venture and incomparable in scope to the WireGL and Chromium libraries.

2.2.2 WireGL

WireGL was a research project by Stanford University Graphics Department to explore Tiled Rendering Displays [15]. The initial software package developed, WireGL, allowed graphics applications to render to a cluster of workstations and output to a tiled display. Developed to distribute OpenGL graphical applications across a tiled display, it at least supported the application types of interest to us. WireGL intercepts OpenGL function calls and packs and sends them over a network to the nodes of a tiled display where a 'pipe server' awaits the incoming data. On receipt of the packets, the functions are unpacked and rendered to a local window, which is created by the pipe server. Using a 'sort-first' algorithm, WireGL pre-determines which nodes will be displaying the geometric data, which is sent accordingly, as illustrated in *Figure 2.2*, [13].

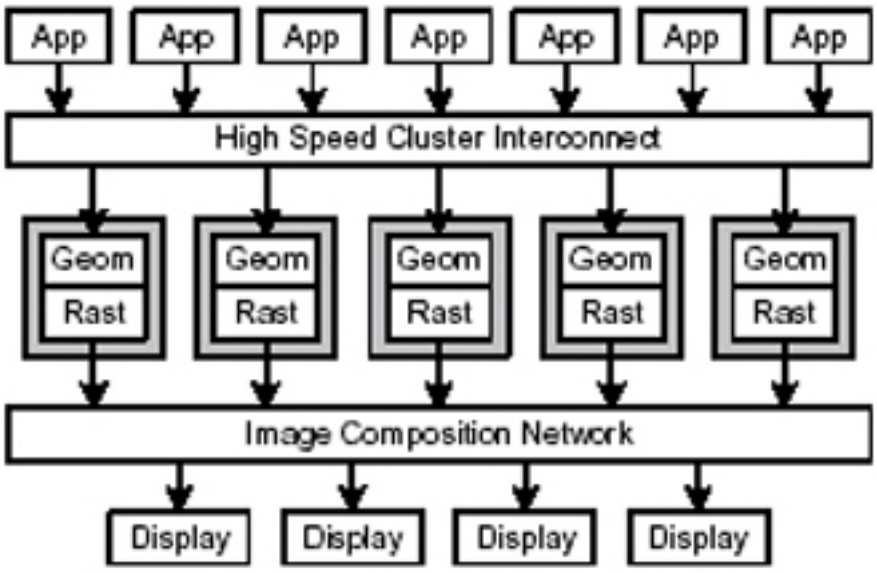

Figure 2.2: Communication in WireGL

Many graphical APIs have ordered semantics, which means that if function A is called before function B, then the resultant image should look as though A was called before B. However, this sequence is not always maintained with concurrent programming, and the ordering can be undefined. To ensure the required and expected results, sorting the output is necessary. Traditionally, a graphical stream cannot be modified once it has been sent to the rendering pipeline, and for this reason two main schools of sorting parallelised images have evolved - sorting the image **before** it is sent to the rendering pipe, which is *sort-first*, and sorting **after** it has been rendered, *sort-last*.

Sort-first is the division of the two-dimensional screen into rectangular regions or tiles for parallel rendering, each tile being assigned to a different

processor. Sort-last splits the dataset into arbitrary subsets, which are then distributed among the nodes. The renderer computes the pixel values for its subset, regardless of where on the screen they actually fall. The data is sent to the compositing processors. These are responsible for the final image being depth-sorted, or z- buffered, back together from a fragmented frame-buffer. Objects close to the view-port are rendered, and then those objects behind the foreground ones that should be visible are shown, so the final image is sorted according to distance from the view-port. The sort-last method scales well to the data intensity, since each primitive is only rendered exactly once - no overlap. It does require a high-bandwidth though, and rules out a number of rendering algorithms, such as anti-aliasing, since the depth of a given pixel is only computed at the last stage. Sort-first is more suited to our purposes as it utilises the complete pipeline for a section of the screen, fully exploiting the graphics cards, which are optimised for this purpose. It also takes advantage of frame-to-frame coherence - an optimising technique which plays a vital part in many dynamic applications.

These parallel graphics architectures are an efficient alternative to broadcasting the OpenGL functions to all the nodes, which is far more taxing. WireGL supported neither multi-rendering contexts nor display lists, which are essential to many of today's graphics programs. Though it is relatively straightforward to use, requiring that a program be recompiled against WireGL's rather than OpenGL's libraries, and executed by starting the pipe servers on the destination nodes, WireGL does fail in the performance department,

which eliminates it as a potential testing tool. It has since evolved, or been rolled into Chromium.

2.2.3 Chromium

Like WireGL before it, Chromium is an open-source project, which means the code is completely free and available. It runs on an impressive range of operating systems, including Linux, IRIX and SunOS as well as Window-based systems. Embodying the idea behind software display systems, Chromium abstracts the underlying architecture and network topology, and, by intercepting and manipulating the API command streams, enables a range of applications to run in different environments.

Chromium accommodates the implementation of parallel rendering algorithms, providing an environment that allows the customisation of OpenGL commands via SPUs, Stream Processing Units, which means the user can redefine or filter the streams of graphics commands. The entire WireGL library encapsulated inside the 'tilesort' SPU,which redefines the GL calls such that they are packaged and sent over a network to the destination node(s). This 'tilesort' SPU uses the *sort-first* approach of WireGL. Chromium also offers sort-last rendering, which depth-sorts the image, as explained above. It incorporates hybrid parallel rendering as well, whereby the stream filters can be set to a combination of sort-first and sort-last parallel sorting architectures. These SPUs are completely extensible and can easily be modified programmatically by users, allowing entirely new SPUs to be created and to-

tal customisation of the stream transformations. Such generality means that practically any cluster-parallel rendering algorithm can be either inserted into the Chromium or implemented upon it [14].

Designed with OpenGL as the applications' environment, Chromium simply re-links this API's libraries against its own newly defined SPU libraries, replacing the existing ones. This is a huge advantage as, not only do we avoid the tedious rewriting of the applications in another language, as required by many of the alternative systems studied, but many applications can be run directly on Chromium with no modification to the original code. It also contains its own Chromium-specific synchronisation primitives, barriers and semaphores, to allow applications to be modified with python for parallelisation across the cluster. This will be explored in more depth when application parallelisation is discussed. Another feature of Chromium is the many shared libraries it builds, identifiable by the *.so* extension, which can be used by any program. These are normally linked in the Makefile, a file which defines all the libraries and tools required by a program, along with their location in the system.

How it Actually Works

Chromium is based on three primitive nodes: the configuration mothership; a Chromium application faker, the *crappfaker*; and a Chromium server, *crserver*. It lies upon the operating system, and communication between these primitive nodes takes place through the network of the cluster, as

shown in *Figure 2.3*.

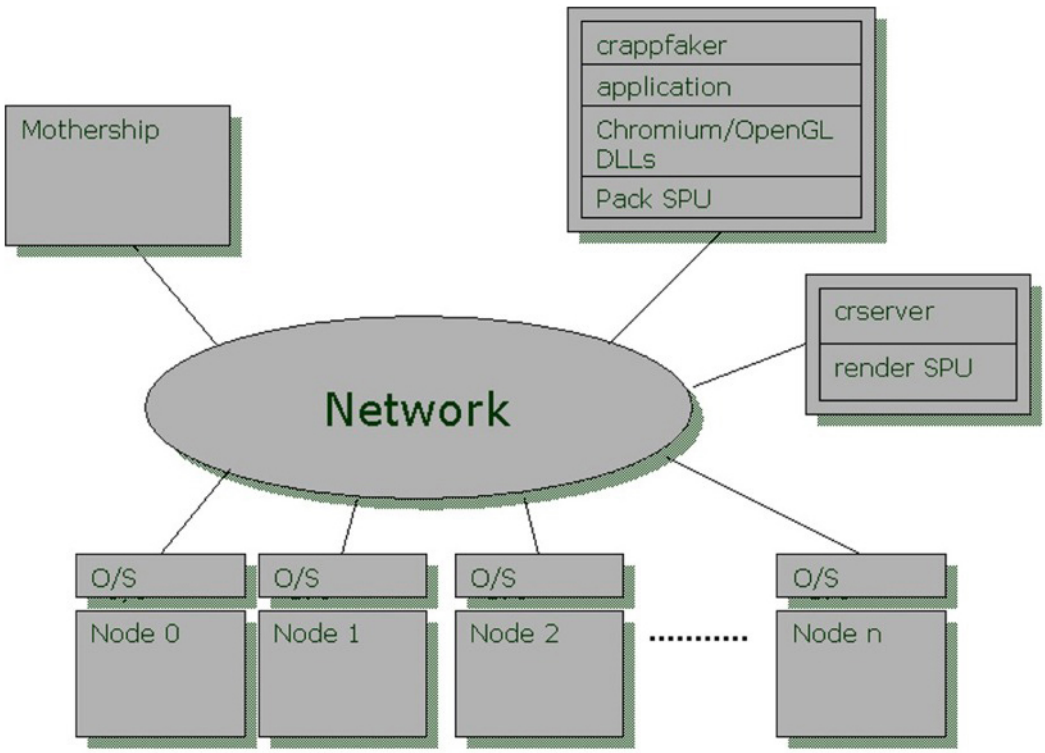

Figure 2.3: Chromium Structure

The mothership controls all of the programs running on Chromium. Written in Python, it supplies all information to the various components as to what they should be doing in any particular run. This includes which SPUs should be implemented, the name and directory information of the application, and the sorting SPU used. It loops infinitely, awaiting response from the crappfaker(s) and crserver(s). All of this occurs dynamically with the starting of the mothership: *python*2 *script.conf* where *script* is the name of the configuration file the user designs to run the application. Working on the principle of one or more clients issuing OpenGL commands simultaneously

to one or more servers. The crappfaker is basically a loader program, or launcher, that initialises another unaltered application and ensures it finds the *Chromium* OpenGL shared library, the replacement library, rather than the system's. The manipulated calls are then sent over the high-speed network to the servers. The crserver dispatches the incoming stream of API commands to the first SPU hosted by the server which renders the geometry. All the components configure themselves by questioning the mothership, which supplies them with the required information. The render SPU creates an OpenGL rendering context upon its' startup. SPUs are typically linked together in chains, where the next link in the chain inherits all from the previous link. To implement an initial test application, it is necessary to make it 'Chromium aware' by linking it directly to the SPU libraries (in the Makefile in this case), or, alternatively, allow Chromium's 'crappfaker' to launch the desired OpenGL program and relink it against the SPU libraries. In some cases the user doesn't even have to recompile the program.

2.3 Selecting Applications for Testing Chromium

In choosing applications best suited to testing the performance of Chromium, a number of factors had to be taken into consideration and prioritised. The program would have to be highly scalable to enable us to evaluate its ability to meet different computational and rendering parameters. To get a realistic testing of this rendering system, the selected application should be computation and rendering intensive. If there is a low requirement of these, then the

advantages of using the rendering system would be outweighed by the overhead in implementing it. Another reason for a high processing and rendering requirement is that these applications are suited to parallelisation as the most efficient manner of managing demanding workloads. Additional optimisation techniques incorporated in the algorithm would give a more realistic feel to the project, as these will be applied to the sort of applications the system would be used for in industry. A degree of platform-independence would be helpful since it should be easier to design and write most of the code in the windows environment and then translate it across to the Linux machines on the cluster.

2.3.1 Fractal Art

The foray into potential testing applications began with fractals. Highly iterative, fractals were interesting to explore both mathematically and visually. The first requirement of the potential applications being met immediately as, by their very nature, fractals are highly scalable. Huge numbers can be set for the amount of iterations, increasing the level of detail required and hence the computational and rendering power proportionally. A fascinating aspect of fractals is that the more you zoom in on a part of a fractal, the finer the detail you discover. Since they are iterative, the same pattern is repeated in smaller and smaller dimensions. Found in nature - fossils for instance - fractals seem to hint at whole Euclidean idea of nature obeying mathematical laws, or perhaps mathematicians attempts to impose mathematical laws upon nature. Innumerable Internet sites are dedicated to this

new art form, fractal art, with many packages, some open source, enabling non-programmers to invent their own, leading to some incredibly beautiful ones being produced, see websites [4] and [3] in particular.

Beginning with some of the more simple, well-known fractals, the Mandelbrot and Julia Sets, which were familiar from mathematical studies became the starting point. The first working program completed was Sierpinski's Gasket, or Triangle. This works on the principle of iterating through a theoretically infinite loop of an iterative mathematical formula, the underlying ideology of most fractals. The algorithm involved establishing three initial points, forming a triangle. The first step requires one to randomly choose two of the three points and obtain the midpoint between them. From there, a loop is entered whereby the last point computed is added to a randomly chosen point of the original three, and the midpoint of these forms the next point, see *Figures 2.4*, [7], and *2.5* for clarification of this concept.

This was implemented on the Linux machines, with the necessary modifications being made for this operating system since all the applications were developed in Windows environments. The next stage involved running it on Chromium. Using the in-built capabilities of Chromium, two programs were developed - one running directly on a single processor; the other split the program, rendering the application on one node, and outputting the display on another, all controlled by the mothership, which is set up on the third

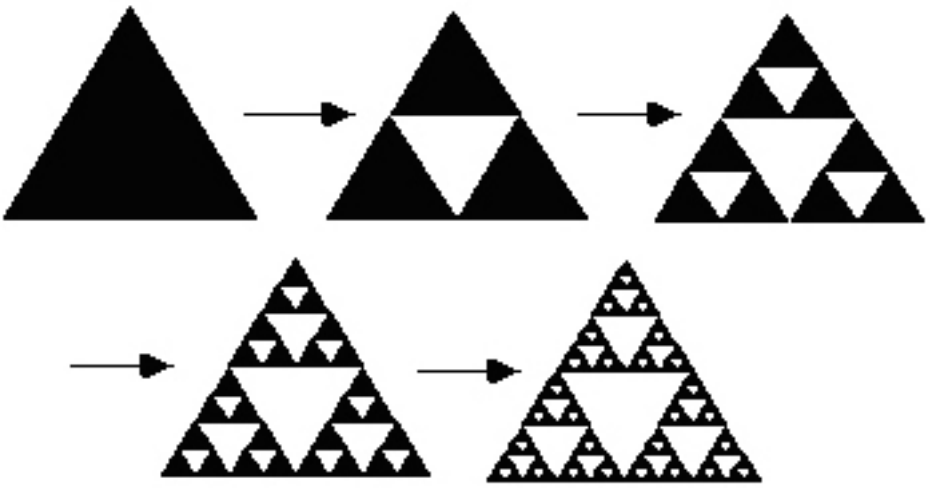

Figure 2.4: Diagram of generation of Sierpinski's Gasket

node, *cagnode12*, as seen in *Figure 2.6*.

The user's ability to alter features of the program was not as achievable. A blanked out screen was created on one node to catch the user input, and this data was then passed across the network to the other nodes where it was then applied to the rendered object. This was chronically slow, and since real-time rendering was a vital aspect of the project, too expensive frame rate wise, to implement. The main reasons for not to persevering with this particular fractal, Sierpinski's Triangle, was that it could only be scaled to a certain threshold intensity before further increases were indiscernible. Added to this was the fact that the computing and rendering of this algorithm does not lend itself to parallelisation very readily.

```
-- An Array of three to hold the initial values of the Triangle
T[3] = {initial value for triangle};

-- An randomly generated number between 1 and 3 to index the initial points T[] above
int index = chooseRand(3);

-- Assign the values of the randomly chosen initial point
point=T[index];

-- Draw the initial lines along the x, y and z planes
glVertex3f(point.x, point.y, point.z);

-- The principle loop of the algorithm
for(int i=0;i<10000;i++)
{
-- Randomly choose a point
index=chooseRand(3);

-- Obtain the new point from the midpoint of last and one of originals
point.x = (point.x + T[index].x)/2;
point.y = (point.y + T[index].y)/2;
point.z = (point.z + T[index].z)/2;

-- Draw the new point
glVertex3f(point.x, point.y, point.z);
}//end of for loop
```

Figure 2.5: Pseudo-code of main loop of Sierpinski's Gasket

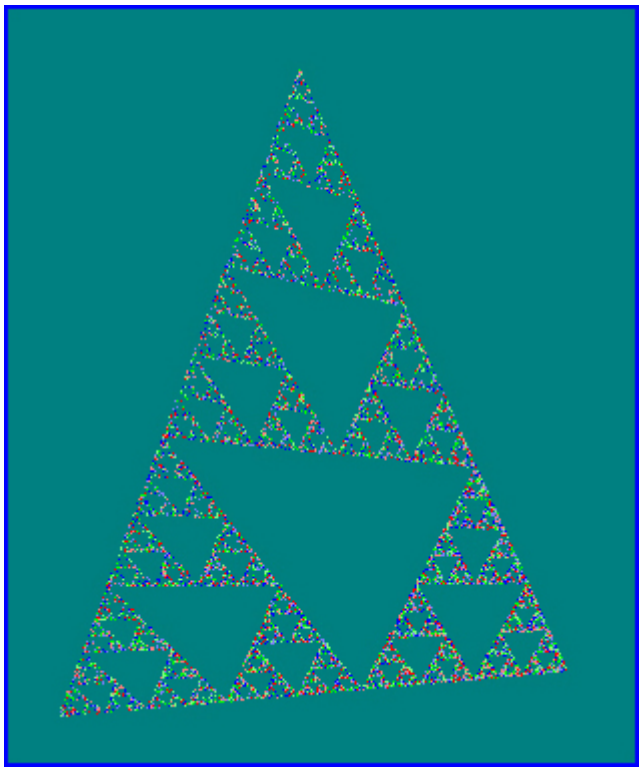

Figure 2.6: Output showing Sierpinski's Gasket

2.3.2 Terrain Visualisation

One of the purposes for which fractals have been adapted is producing recursive artificial terrains. The first of this type of algorithm implemented was a *Mid-point Displacement Algorithm*, also known as the *Diamond Square Algorithm*. This involves a simple algorithm, $E = \frac{(A+B+C+D)}{4} + RAND(d)$, where *RAND(d)* is a random value in [-d,d]. The value of d represents the maximum displacement in the current iteration. This formula is executed iteratively, and generates a square terrain with $(2^n + 1) \times (2^n + 1)$ dimensions, where n is the number of times the function is called, as shown diagrammatically in *Figure 2.7*, [17]. The advantage of such an algorithm,

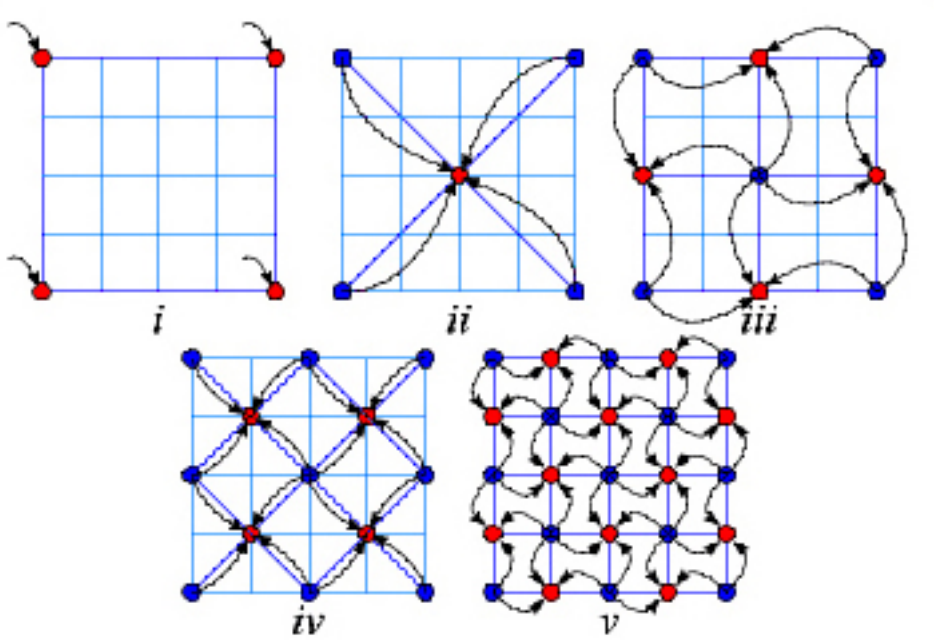

Figure 2.7: Diamond Square Algorithm

as with the second of its kind implemented, the Circle's Algorithm, lies in its simplicity, which means rapid computation. While a working version of the Mid-point displacement algorithm was implemented on a windows machine, it was never transferred to the cluster. No significant optimisation is incorporated in these applications, which is a key element in large-scale models and therefore important to include in the testing applications. The applications used to generate highly-complex models of a magnitude normally requiring supercomputers are of a far more sophisticated nature. For this reason, the simple applications above were discarded in favour of a more refined and involved graphics application. Further research into terrain generation revealed a rapidly expanding field. 'Terrain databases are well beyond the interactive rendering abilities of even high-end graphics hardware'[19] was the consid-

ered opinion of some developers as recently as 2001. Terrain Visualisation, with its ubiquity across so many fields, is an important area in computing as it is seen as the next big obstacle to be overcome in the pursuit of realistic interactive entertainment. It makes intensive use of graphics hardware while trying to achieve acceptable rendering speeds as well as high computational requirements, which makes it both an attractive field for testing Chromium, as well as an ideal candidate for parallelisation of the workload. A number of highly effective and optimised algorithms were explored based on the premise that these are the type of applications with which Chromium would be most effectively tested.

The scalability of the application was considered an essential feature as the eventual testing of the system would involve ascertaining the performance at different levels of complexity. Many terrain generation algorithms are amenable to this as the level of complexity can usually be increased from quite a coarse low polygonal count model to an enormously detailed one without too much difficulty.

The two main aspects in the rendering of a terrain are the initial representation of the terrain inside the machine, and turning this into a two-dimensional image on the computer screen. The simplest form of terrain representation is as a height field. Consider a regular grid in the plane XZ, with evenly spaced points, and a height attributed to each point, see *Figure 2.8*, [17] .
This representation requires little hard disk space, saving a lot of storage

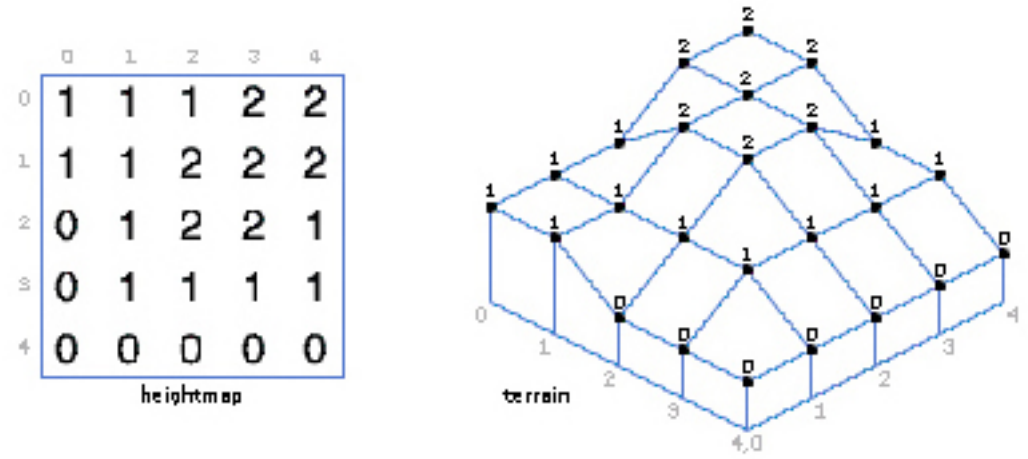

Figure 2.8: Height Map

as only the heights are needed along with a reference point in the terrain - the centre point in the plane XZ for instance. Since the grid is evenly spaced, both x and z values for each point need not be stored. This data in a two-dimensional array of the heights *(a height field)* of a landscape taken at regular intervals. A particularly useful format, the array can be accessed in linear time and is also compatible with bitmap images, allowing editing and viewing of the terrains in external applications. The level of complexity can be modified by sampling the data points less frequently; at every third instead of every entry for example. The most limiting disadvantage to this structure is that overhanging terrain cannot be stored, since each point has only one height value. This is acceptable for our purposes, so height fields have been chosen as the most appropriate structure to use.

The data is read into memory and an accurate mesh of it is generated, theoretically at least, by the visualisation program. The actual generated mesh is formed with a seamless merging of the points of the height field, resulting in a landscape that is then rendered to the screen.

Optimising Terrain Generation

Terrain visualisation can be viewed as a continuous level of detail rendering of height fields. Level of Detail (LOD) is a term used to describe using many different resolutions within a scene - for instance, objects or terrain blocks in the distance need not be rendered with the same level of complexity as those near the camera. Due to the complexity involved in modelling real terrains faithfully, it is essential to optimise the applications where reducing the overall complexity of the model naturally improves the runtime performance since less rendering is required. As early as 1976, James Clarke outlined the advantages of having several different resolutions within a scene - a concept which has permeated almost all terrain generation algorithms ever since. The main approaches taken are simplifying the polygonal geometry of unimportant objects, and flat or distance regions. This elimination of redundant geometry incorporates techniques such as generating different levels of detail or complexity for triangle geometry and texture blocks and triangle stripping. Triangle stripping, see *Figure 2.9*, where one connects a number of independent polygons or triangles into a strip using a greedy algorithm, is one of the easiest ways of speeding up OpenGL applications. The initial approach was getting a viable working version of an optimised

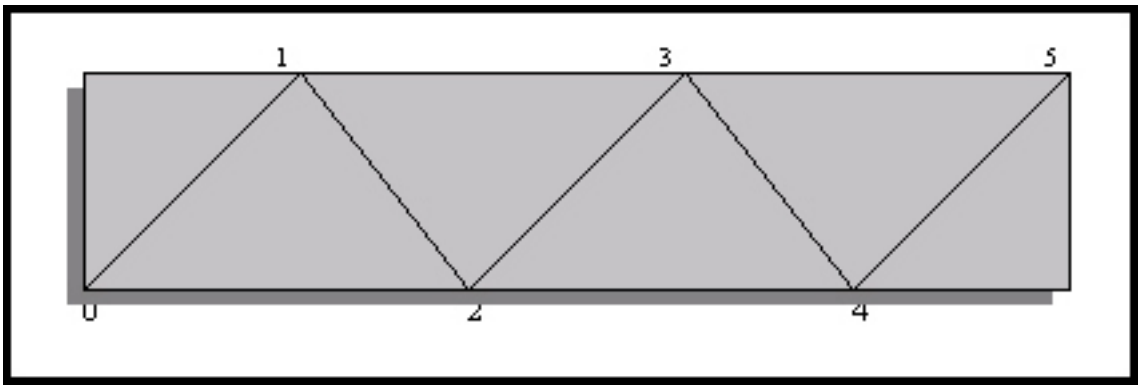

Figure 2.9: Triangle Stripping

terrain visualisation algorithm implemented; additional features of the landscape could be incorporated at a later stage. After further research into the huge range of available algorithms, it became apparent that their implementation depended a lot on their motivation. By prioritising a number of benchmark measures of efficiency it was easier to eliminate options. The main measurement factors were:

- structural *(or numerical)* fidelity - *how closely the mesh matches the terrain to be modelled;*
- visual accuracy - (*popping, silvery triangles, visual defects*;
- preprocess time - *algorithm complexity*;
- the size and complexity of the models that could be visualised satisfactorily and whether out-of-core preprocessing would be necessary (*when the model is too complex to fit in memory available*).

Two main branches of the simplification of such applications are considered, the latter eventually chosen. Conventional static simplification works by sev-

eral versions of the same object at differing levels of detail, and converting them to triangular strips during preprocessing with a final compilation of these as individual display lists. However, with their increasing popularity, dynamic polygonal simplification techniques have been incorporated in many current designs.

Dynamic rendering works on the basis of extracting the desired LOD from a data structure which encodes a continuous spectrum of detail. Better granularity is achieved with this method since the LOD for each object is precisely quantified - rather than choosing from a limited number of precreated options. This is exploited in view-dependant systems, where the LOD is selected for each new view, as detailed by David Luebke in his analysis of polygonal simplification [19]. A high LOD gives good visual accuracy but drastically reduces the frame rate. The other extreme, a low LOD, provides an interactive frame rate but is visually unacceptable. Popping and cracks, see *Section 2.3.5*appear in the surface of the terrain when it is broken into blocks of different resolutions. View dependant simplification systems concentrate on a balance between high resolution close to the view-port with decreasing levels of detail proportional to distance, with a smooth transition between the different resolutions. Disadvantages include the increased runtime load of choosing and extracting the best level of detail, but this is combated with frame coherence, where the new frame is computed from updating the previous frame, as opposed to working from the initial frame settings.

2.3.3 Methods Explored

A number of different approaches to terrain generation are under investigation, including:

- Progressive Meshes;
- Contour lines;
- Hierarchal TINS (Triangular Irregular Networks), which encompass vertex-morphing with queue-driven top-down refinement[18];
- Quad trees, which include approximate least square fit in pre-processing, priority queues and top-down refinement of quadtrees.

Some of those studied in greater depth are briefly outlined below.

CLOD

CLOD - Real-Time Continuous Level of Detail Rendering for Height Fields[18], works by first coarsely simplifying the height field to determine the detail levels of various blocks of landscape. Consequently these blocks are finely triangulated according to the predetermined detail levels *(the more time consuming stage)*. These finely triangulated blocks are then taken and merged together, outputting the resulting mesh to the screen [12].

Geometric Mip-Mapping

Fast Terrain Rendering, using geometric mip-mapping, lets the 3D card render as much as possible, leaving a minimum of rendering for the main processor. Regular blocks of the terrain height field are drawn having selected

a suitable LOD for that block, where the LOD is changed as the camera moves. Basic view-frustum culling is implemented, where inviable terrain is not rendered. The lower detail meshes are pulled from the height field directly by sampling [9].

2.3.4 ROAM

Real-time Optimally Adapting Meshes, ROAM, was actually designed for a synthetic aircraft sensor simulation and addresses many of the principal problems associated with real-time visualisation. It provides a real-time display of complex terrains by computing multi-resolution binary triangle tree meshes for dynamically changing views. ROAM can be implemented at a very basic level, just using the central concept, to very sophisticated and advanced optimisations, where the real advantages of this algorithm crystallise. It incorporates dynamic updating of the height field and rapid tessellation of patches. ROAM concentrates on error metrics, frustum culling, priority updates, split and merge operations, and triangle stripping.

With geometric accuracy as its driving measure of success, a vital feature of ROAM is the guaranteed error bounds. The distance between where the point *should* appear in the screen space and where the triangulation actually places the point is used as the error metric of the algorithm. A priority is associated with each triangle according to this dynamic measurement, and splitting and merging operations are called accordingly. Being a view dependant system, it optimises flexible error metrics which alter with every new

change in the view-port. This means that the level of detail in the generated terrain increases with proximity to the view-port and with increased complexity of the scene, as examined in the next section. Frustum culling is based on the idea of minimising the computation and rendering by eliminating redundant aspects of the terrain, see *Figure 2.10 (diagram based upon Figure 2 in [10])*, where the dark region lies outside the frustum, the white, within it, and the light grey overlaps the boundary.

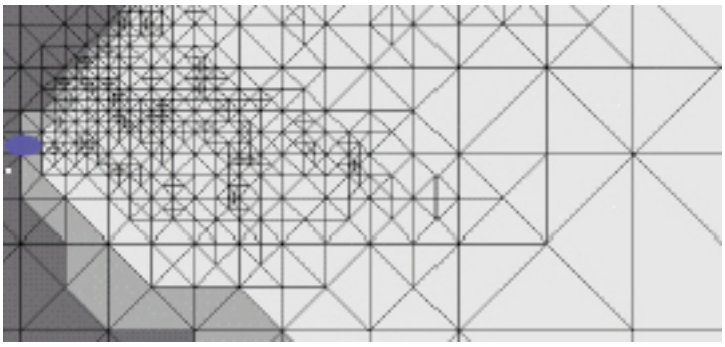

Figure 2.10: Frustum Culling, with the eye placed at the purple ellipse, looking right

With a constantly changing view-port, it is clearly improvident to render the entire scene. Therefore, only that which is visible within the view frustum at any given time is actually rendered. The rest of the terrain is clipped to this view-port, as shown in the screen shot *Figure 2.11*.

Frame-to-frame coherence is an integral optimisation of the ROAM algorithm. This uses the the last computed frame and updates it, as opposed to computing the difference from the initial frame. Another measure of the efficiency of an algorithm is the size of the output; that is the number of triangles outputted directly - ROAM achieves specified counts directly.

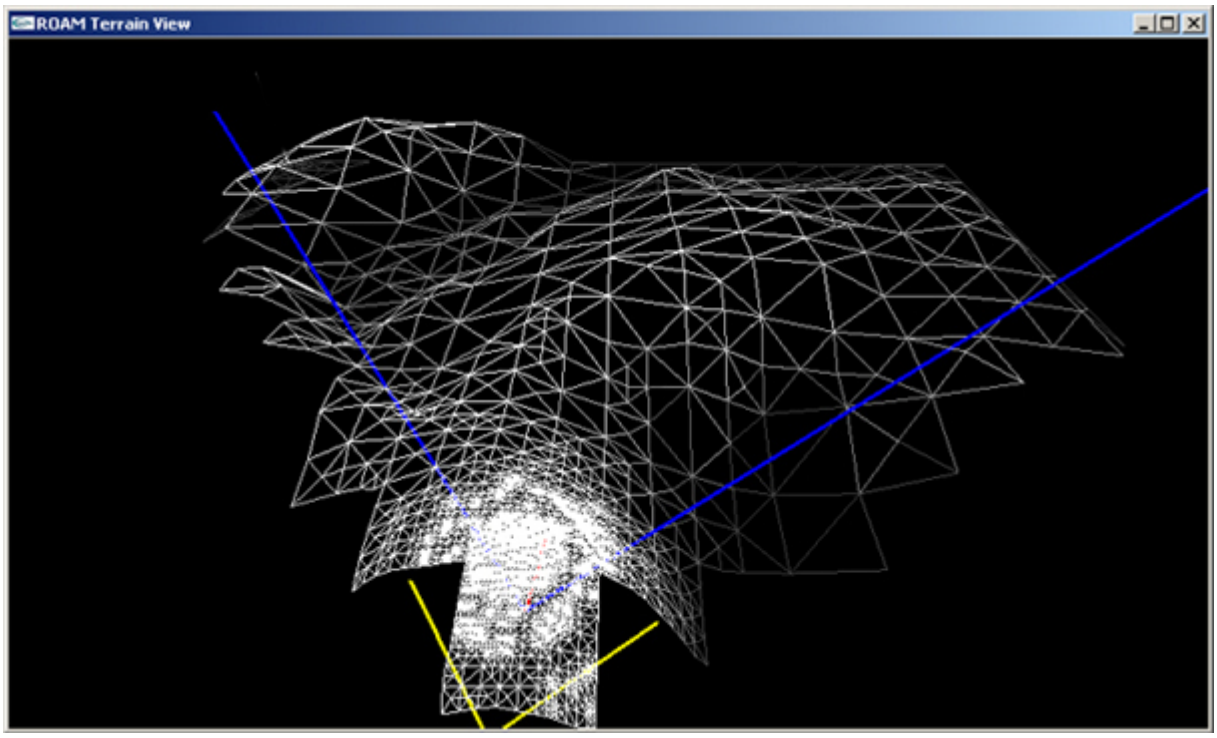

Figure 2.11: Terrain clipped to rotating View-port

Underlying Structure

ROAM is based on the Binary Triangle Tree (BinTree) structure - the counterpart of the more familiar square-shaped quadtree - with *split* and *merge* operations. The Binary Triangle Tree stores the implicit screen coordinates of the triangles, which saves up to thirty-six bytes of memory per triangle [21]. It is a far more attractive and efficient alternative to maintaining a vast array of triangle coordinates for the mesh. At the lowest, or root level, the tree consists of a simple right-isosceles triangle. Recursively splitting the triangle from its apex to the centre of its' hypotenuse produces two child triangles. These are in turn are split, producing higher and higher levels of detail as required, as shown in *Figure 2.12*, [10]. Any triangulation can be acquired from a sequence of split and merge operations, where merging triangles achieves a coarser level of complexity.

The BinTrees are embodied in the TriTreeNode structure, which defines five key relationships relevant to all triangles in the mesh. The pseudo-code can be viewed as such in *Figure 2.13*: A continuous mesh is formed when two triangles overlap at a common vertex or an edge, forming a set of bintrees (bintree triangulation). Each neighbour is either at the same level, or one level coarser or finer than a given triangle, which prevents cracks in the mesh. The TriTreeNodes are assigned from a static pool, which enables very quick and efficient resetting of the state, since invalid nodes are eliminated and new patches created in the *Patch Reset()* function. It also means avoiding the cost of dynamic memory allocation, which is considerable when one bears in

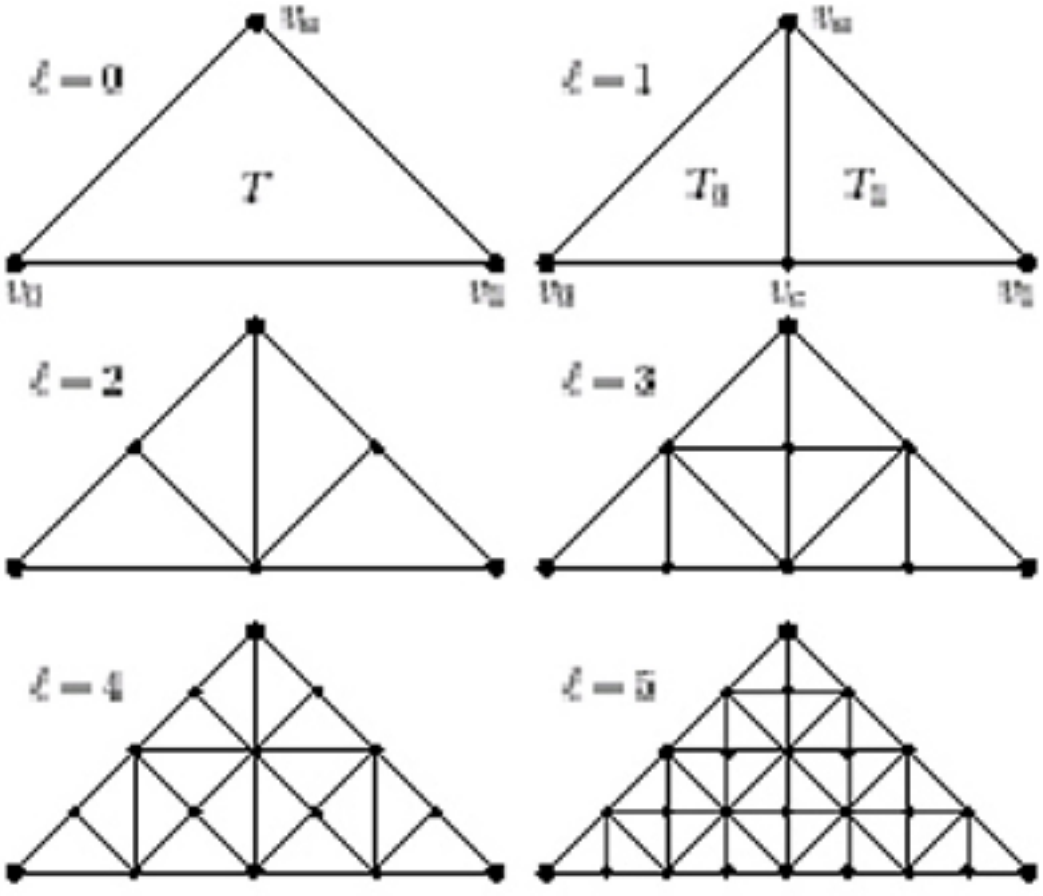

Figure 2.12: Splitting Binary Tree to increase LOD

```
struct TriTreeNode\{
        TriTreeNode *LeftChild;
        TriTreeNode *RightChild;
        TriTreeNode *BaseNeighbour;
        TriTreeNode *LeftNeighbour;
        TriTreeNode *RightNeighbour;
```

Figure 2.13: Binary Tree with Children and Neighbours

mind the testing includes methods which ran at 25,000 BinTree tessellations per frame and the frequent creating and destroying of these nodes.

How ROAM Works

ROAM tessellates the mesh using the binary triangle trees for precomputation, and, at runtime, builds an accurate model by increasingly more precise split and merge operations. This involves a preprocessing stage and four runtime components. A view-independent, bottom-up error bounds for a binary triangle tree is produced during preprocessing. During runtime, the view-frustum culling is updated recursively; the priority of the output triangles that may be split or merged are amended; the triangulation is renewed; triangle strips are modified if affected by the culling. To understand how these are accomplished, we look at a breakdown of how the algorithm actually works.

The height maps, as explained above, are loaded into memory and are associated with a *landscape* object, where terrains of infinite size can be generated by linking multiple instances of the landscape class together. Each new object divides the height field into large square sections and sends them to a new instance of the *Patch* class, which achieves two aims: limiting the size of the areas curtails their depth, and, since the Bintrees expand RAM usage exponentially with depth, this amounts to a significant saving. The other advantage to using small areas is that it enables the rapid re-computation of the height field with the dynamic updates of the view-port. Larger patches

would be prohibitively slow to recompute to run at interactive frame rates. Each *patch* object is comprised of two Binary Triangle Trees merged to form a diamond shape. Storing the coordinates in this BinTree structure saves enormously on memory. The output mesh rendered is created from the triangles represented by the leaf nodes of the BinTrees - which are the finest level of detail of the tree. This is conducted in a two-phase fashion: firstly, the children are added to the tree recursively, until the desired resolution has been reached; the second phase involves traversing the tree again, and the resultant mesh is displayed on screen. Since the triangles to be rendered are actually computed and rendered during the traversal - there is no need to store this data - again making substantial RAM available for other use. The *Patch* class methods control the most important aspects of the ROAM engine. They are called for each instance of the class by their parent *Landscape* functions. Different sized terrains are accommodated by scaling the patch size appropriately, in the *Init()* method.

The most important variable in the ROAM algorithm, and the driving force behind it, is the *variance*, which is the error metric used for deciding if, and to what depth, a BinTree node should be divided. Essentially, it is a measure of the difference between the actual height field area and the BinTree node representing it. It is computed by calculating the difference in height of the interpolated midpoint of the hypotenuse of the node, and the sample point from the height field. It has to be calculated deep inside the tree to get a more accurate value. *'A variance tree is a full-height binary tree written into*

a sequential array' [21]. Each node in the array is filled with a single byte

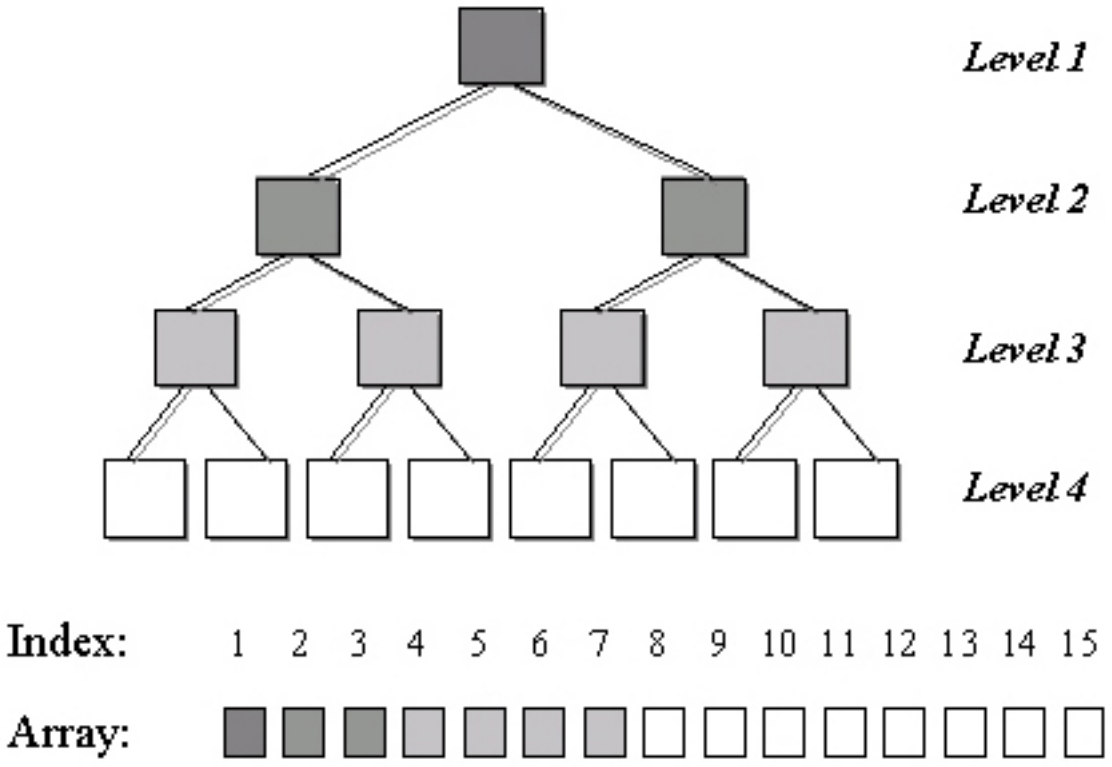

Figure 2.14: Variance Array with Binary Triangle Tree

value of the difference per node, as depicted in *Figure 2.14*.

The *Patch* class contains two such arrays: one for the Left and one for the Right binary triangles. The variance error metric is checked when deciding whether or not to split a given Binary Triangle node. If the terrain being represented by said node is complex, or rough, then the variance will be high, and the triangle should be split into its children nodes for a more accurate mesh approximation. These children nodes are then followed down,

and the process is repeated, the variance being recomputed (around halved) at each iteration of this method, until the difference between the position of the triangle to be rendered and where it's supposed to be rendered is deemed sufficiently low. It is at this stage of the splitting processes that the rule of neighbouring triangles only being permitted one Level Of Difference, LOD, is implemented. Pointers to neighbours are used to keep the LODs synchronised during the mesh tessellation. A split is only allowed to occur if the current node and its base neighbour both point to one another, forming the diamond shape alluded to earlier, and there exists not more than one LOD between them, as depicted in *Figure 2.15*, [10]. 'Force' splitting occurs if

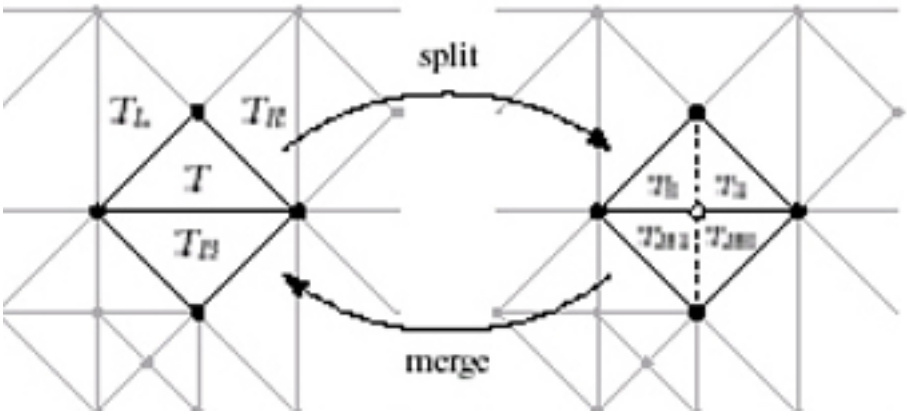

Figure 2.15: Splitting a Diamond

a diamond can not readily be formed, requiring the forced division of the base neighbour to supply the second triangle for the diamond. A split in one triangle of the diamond can be mirrored by a split in the second without the possibility of cracks. Subsequent divisions, or merges, require the same operation to be conducted on the other triangle, so that only one difference in resolution is tolerated. The diamonds are then relinked back into the mesh,

where this procedure is recursively called until the variance has been satisfied. The other influencing factor upon the variance is its distance from the camera. Since ROAM is a view-dependant system, with greater complexity nearer the view-frustum, a higher variance is allocated to these nearby triangles, requiring them to be split further than those in more distant regions. To recapitulate, for each frame, the variance is recomputed for each triangle as a function of its distance from the point it represents on the height field, and its distance from the camera. The landscape is tessellated, or created, by using the variance as a measure of the depth each BinTree node is to be split to, in order to obtain finer levels of detail. This tessellation step is reiterated on the child nodes until the desired level of detail is attained.

2.3.5 Aesthetic Appeal

An essential element of terrain visualisation is that it appears realistic and believable. Certain parameters for visual accuracies are used to ascertain how well it meets the required standard. A number of research areas are involved with determining what level of accuracy the human eye requires for a scene or model to be believable. The Binary Triangle Tree structure, along with the *split* and *merge* operations, which are the backbone of the ROAM algorithm, eliminate the need for complicated legitimacy rules for changes in the resolution, which are a fundamental requirement for irregular, or even quadtree structured spaces.[10]

Common visual discrepancies include slivers, streaks, pops and cracks in the

terrain. Slivery, thin triangles sometimes appear during the triangulation of the mesh, but this numerical problem is avoided as the triangles are always right-isosceles in shape. Streaks occur when too many triangulations are computed at a given vertex - also not an issue in this algorithm. Popping occurs when triangles are suddenly inserted and removed from the mesh. This is not really an issue with ROAM with its control and gradual transition between multi-resolutions, but one that can be addressed with geomorphing, an interpolation method on vertex heights. Cracks are breaks in the terrain often caused by a significant difference in the level of detail of one region and another. This cannot occur in our program as ROAM ensures only one level of difference between neighbouring triangles is permitted.

2.4 Parallelising ROAM

The performance of the application could clearly be further improved upon by parallelising the actual code. Parallel rendering is characterised by two or more processes simultaneously issuing commands to the rendering system. Usually, each process will render a part of the entire scene or model, and the pieces are combined at some stage to form the final image. A number of different approaches to dividing the computation and rendering are explored and attempted, as discussed in the next chapter. The running of concurrent processes does, however, require additional control, and communication between these processes is often necessary.

There are several different avenues of distribution of the workload on the

cluster that were under consideration for the project, a number of which required quite refined inter-process communication and control. Chromium offers some limited control, namely barriers and semaphores, in an attempt to address the ordering constraints of OpenGL applications. Functions such as broadcasting to all the nodes, however, are not supported by Chromium, hence the need for a more sophisticated message passing system. As Chromium is not sufficient for more complicated process management, alternative communication techniques between processes have been researched.

Message Passing between the Nodes

The first system explored was the Parallel Virtual Machine (PVM). This is a software package that allows a heterogeneous network of computers (parallel, vector, or serial) to appear as a single concurrent computational resource - a virtual machine. Though used in some older systems, this was rejected in favour of the industry standard, Message Passing Interface (MPI)22. This, is a library specification for message passing. Representatives of approximately 40 bodies of academia, industry and government were key in its' development [1], from 1992 - 1994. Drawing on years of experience with a number of different systems and libraries, MPI was developed as a common standard, one which is practical, efficient, flexible and highly portable, for writing message-passing programs.

The inherent advantages of standardisation in technology - portability and ease of use - were realised with MPI. This has led to some manufacturers

providing hardware support which enhances scalability, as well as a very definite base set of efficient functions to build upon. Its features have been drawn from a range of different message passing programs, notably the work at the IBM T. J. Watson Research Center, Intel's NX/2, p4, PARMACS, Express and nCUBE's Vertex. While many applications only ever use the six most common functions of MPI, this is a highly developed system, with an impressive range of functionality associated with it. MPI also works across Ethernet, which is the interconnect used in our cluster. It appeared to be by far the most fitting interface to use for this project.

Chapter 3

Implementation

The implementation of ROAM used in this project incorporates many of the optimisation techniques discussed in the preceding chapter. Much of the original implementation was coded by Bryan Turner[21], modified for use in this project. Having succeeded in getting a version of the algorithm working on the cluster, the next stage of development was getting this serial version running with Chromium, removing and replacing GLUT functions, inserting *python* commands directly into the program, figuring out how MPI works through research and writing test applications and, finally, getting all of these working together - the most significant obstacles to which are outlined briefly below.

Further to this was the actual parallelisation of the code. Splitting up the processing and rendering in an effective and balanced manner required intensive redesigning and recoding, as attempts to implement theoretically sound ideas proved insurmountable at times, and just terribly inefficient at others.

A few of these are mentioned, along with an analysis of why they were discarded. The most promising implementations, which were eventually used for testing Chromium are then examined in detail.

3.1 Serial versions of ROAM

The first program was prepared on a windows operating system, in Microsoft's Visual Studio environment. This was a sequential version of the ROAM algorithm illustrated above. The original images are limited in size so that dimensions are to a power of two as the graphics hardware available require this constraint. Alterations had to be made to enable the execution of this program on the Linux operating system. This involved changing definitions of certain routines and calls, replacing any windows calls with Linux versions and creating a Makefile to link all the required libraries and directory paths on the cluster nodes. This done, the application ran without too much difficulty, once the errors were understood and repaired.

Having successfully initiated a basic building block application, running the application with Chromium was the next step. Chromium could run the application directly, with no real requirement for python code to be inserted into the application itself. This run meant modifying a Chromium python script, set it up according to the number of servers and application nodes desired, and giving it such information as required to find the executable file. With some experimenting and working out how the different calls and SPUs worked, and building on the experience of getting the Sierpinski's Tri-

angle application working, this succeeded. It was run locally on one node, tilesorted and then distributed across the cluster. Naturally, this was more a learning experience of Chromium than actually implementing something worthwhile as no performance improvement was evident. Chromium simply duplicated the serial code across the cluster as required.

To achieve performance speedup, the program would have to be parallelised, and some control used to maintain the ordered semantics of the original OpenGL code. This meant incorporating python script barriers and semaphores into the code. Since MPI was going to have to be used as well as Chromiums' own synchronisation techniques, efforts were made to include this in the code as well. It was decided to first get these working together in the serial application, to ascertain the overhead using the Chromium and by initialising the program with MPI instead of OpenGL.

3.1.1 Problems Encountered

Having researched and implemented enough demonstration and test versions of code in Linux, Chromium and MPI, familiarity with these fields led to the first attempted implementation of ROAM, incorporating all of them. Since GLUT is not supported by Chromium, the first issue was its removal from the ROAM program. Removing all the GLUT code wasn't as straight-forward as I hoped, since it was heavily invested in in the application. Nonetheless, OpenGL and Linux-friendly replacements were eventually found for all of the necessary code. Problematic function calls such as `glutMainLoop()` for

instance, had no simple substitute as an equivalent method does not exist. GLUT is a strictly event driven library [2]. Essentially, all GLUT applications must set up callbacks for all the events they are interested in, whether it's input from the keyboard, mouse, resizing or redisplaying the screen, and it is here that `glutMainLoop()` is called. This is an event processing loop, which, once called, will never return. Because it has a central role in controlling the entire program, where other library calls have been adjusted to make allowances for this, attempts to write routines to emulate its functionality proved more difficult than one would initially suppose. This mastered, something as undemanding as printing to the screen, a pretty basic operation one would think, also proved complicated, as this is quite am involved process on Linux. Even after locating and loading the required fonts and parameters, it still conflicted with the present OpenGL code.

The first serious problem area encountered was implementing Chromium and MPI together. MPI alone would work, and, after some experimentation, Chromium worked alone with ROAM as well. Merging these two test programs together proved to be a very involved and complicated undertaking. Initial crashing out was being caused by the Chromium library being called before the MPI one, a very simple problem, but when one gets page upon page of MPI errors, it seemed reasonable, having checked a number of potential problematic areas within the code, to look to MPI itself. Having attempted to link certain required libraries separately, changing pathnames and locations of different files, moving the order of calling functions and

swapping things around, the cause of the crashing finally transpired. This was the first of several difficulties in getting Chromium and MPI to work as a cohesive system.

After eliminated any reported errors, including repeated *p4* errors and innumerable unrecorded ones, from the program, executing it led to two empty tile frames being rendered to the screen. This blanked out screen output continued for over a week, in which time more and more extreme stripping and whittling down of the code took place. This included changes to view-port settings, the frustum parameters, modifying the projection matrices, and the flushing and swapping of buffers at various stages - a huge number of alternative versions and approaches to coding this were attempted. Eventual stripping of the code to the bare minimum required to run an OpenGL program, removing all the terrain generation, Chromium and MPI, led to the printing of a coloured line, and this was gradually built into a simple cube. Notions of a breakthrough were premature however, as efforts to rebuild the application - inserting snippets of code followed by lengthy and thorough testing of each piece - proved in vain. The code was impossibly messed up, and, after another two weeks of hacking it to shreds, stripping and rebuilding it, a stage was reached where the view-ports and projection matrices had changed beyond recognition. The code was unsalvagable in the sense that the time requirement of fixing this program would be too costly. As more intricate problems were encountered with amplified complexity, it was decided to abandon this application.

It taught a good lesson in creating regular backups, the value of using a CVS repository and why commenting and keeping logs are vital activities for any programmer. While frustrated at the loss of time, the experience gained in trying to fix this program, initially written off as over three weeks wasted effort, proved invaluable when it came to starting from scratch again with the first ever working version of ROAM that had been run on the cluster. It was only by tearing the code apart like that, and trying innumerable methods of getting the Chromium and MPI to work in conjunction with one another that a far deeper and more precise understanding as to how the components worked, both as single entities and as part of the system as a whole, was gained. This was imperative for designing and modifying the applications from then on.

3.1.2 Resulting Applications

Starting with the first working serial application as a base, a number of different versions were designed and programmed - this time with proper CVS backups! Apart from a number of testing applications, incorporating different functionality and techniques, which were later disposed of, three main serial versions of ROAM were used in the performance testing. **s**erRoam is just an unadorned version of ROAM. **s**erChrom has python script included, the Makefile is relinked to the Chromium libraries and the required SPUs specified. A number of configuration files were also developed, using python, to specify how the execution of the program was to be structured. **s**erMpi was originally developed just using MPI and not Chromium. *MPI_Init()*

replaces the OpenGL *main()* call to initialise the application. It is also in charge of setting the rank and size of the processes, and this information is shared with the other classes.

The Makefile is modified to compile using mpiCC rather than g++. The MPI library is wrapped around the g++ compiler, which negates the need for explicit linking of libraries, as long as MPI is properly installed and the library is accessible. After thorough testing of this application, Chromium was also added. This was a more involved venture, as conflicts, between, for instance, the barrier control of both, had to be carefully avoided. Barriers are used to ensure all of the data has been processed up to their location in the code, before the next step in the program is taken. With multiple processes, it is essential to ensure barrier conflicts, deadlock, more than one flushing of the buffers and countless other potential inconsistencies do not occur. The difference in performance of these three applications in particular are explored in the next chapter - *Evaluation*.

3.2 Parallelisation of ROAM

Parallelising was the next and a crucial element of the project. As explained in *Chapter 2*, Chromium was designed to support parallelised applications, and it is only with these do the benefits of using it become apparent. Using various tilesorting and depth-buffering techniques with serial versions, one does get a feel for how the system works. The performance of serial versions of code running concurrently, as revealed in the following chapter, is inferior

to that of the parallel version, as it involves additional duplication of the workload, rather than the distribution of it. A countless number of techniques for parallelising were explored and developed, some abandoned quite quickly, others fully implemented before their inefficiency caused them to be discontinued, before the more promising algortihms were used for testing purposes.

3.2.1 Methods Explored

I made an initial design error by approaching the parallelisation of the algorithm bottom-up, which meant trying to divide at the lowest levels and propagating this up through the binary tree, then the patches, until, eventually, the entire landscape was evenly divided. The first attempt implemented began at a very basic level where, on dividing a binary tree into its two children, the left child would be rendered by one processor, the right by the other, and the main processor, *cagnode12*, would act as controller and synchroniser of these events. *note: the 'cagnodes' are so called from cag - Computer Architecture Group* This worked well in theory. Trying to incorporate it into the original algorithm soon became extremely complicated. The greatest obstacle however, was of a less surmountable nature and the impracticalities of such an approach soon became apparent. Both nodes would have to be aware of the level of detail of all the nodes around it, the essential kernel of how the ROAM algorithm works. Since every triangle can only be at the same or one value different from the level of detail of its neighbours, each processor would have to track the LOD of all the nodes bordering theirs that

were being rendered on other machines. With a constantly rotating viewport, the overhead of such a feat quickly made this parallelising technique an infeasible option.

The next division attempt was at a slightly higher level. Every time a *Patch* was rendered, it would rotate between, originally two, machines. On being called initially the patch would be rendered on one processor, and on the other at the next invocation, controlled by a simple swap method in the function. This also involved too much duplication of code and didn't really improve performance as one processor was idle while the other worked and then vice-versa. The process was alternating rather than two running concurrently. Another divided the workload by allowing one processor to manage the computing and rendering of all left nodes, another the right nodes, and the third machine, the apex nodes of the tree and, yet a different effort to parallelise, meant dividing the split and merge queues among the machines of the cluster, the principle under which ROAM operates. These were endeavours which ended in failure.

The cost of overhead communication in all of these methodologies made them prohibitively expensive. Efforts to alleviate this overhead led to attempting higher level approaches. Two of the most promising and efficient algorithms, which were used extensively and very successfully in the testing are detailed below.

3.2.2 Most Efficient Algorithms

The first really successful and worthwhile version of this was quite abstracted and high-level in comparison to previous efforts. Working on the principle of dividing the computation and rendering of the patches among the nodes in the cluster in an alternating fashion. The second highly efficient algorithm parallelised the application by dividing the view-port angles and partitioning the rendering and processing accordingly.

Parallelisation by Division of Patch Rendering

The idea of the patch, a combination of two binary tree triangles was introduced in *Section 2.3.4*. The landscape is comprised of innumerable patches linked together. This parallelisation technique divides the processing and rendering of the patches among all the processors in the cluster. It does so by applying a mod. function to each patch to be rendered, and the resulting value is matched to the machine that is to process it, using the identification number assigned to this processor by MPI. Since each patch is specifically associated with one machine in each frame, no patch is rendered twice. *Figure 3.1* clarifies this concept. If there are six patches, for instance (naturally the real value is measured in thousands), and three processors on the cluster, then the function returns the remainder from dividing the number of processors into the patch number. That is, if patch number five is divided into the cluster size of three, a remainder of two is obtained. This patch will then be rendered by processor number two.

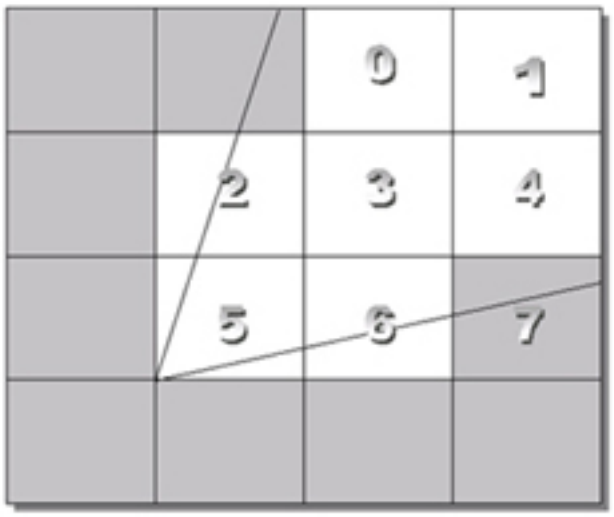

Figure 3.1: Parallelisation by Dividing the Patch Rendering

Parallelisation by Splitting the View Frustum

The division of the viewing frustum angles and allocating each subdivision to a different processor is the basis of this method. This distributes the work extremely evenly, and is highly scalable. Some initial problems with clipping of the view-ports and aligning different angles and divisions needed to be overcome, which actually took quite some time, where clipping is the process of limiting the visual representation of the mesh to a specific area on the display. It is also necessary to clip to each line dividing the viewing frustum into subsections, to avoid overlap.

Initially, the viewing frustum was divided for tessellation, using interpolation between the points to achieve smooth transition. The *Field of View* angles

were then divided according to the number of subsections created. In this case, there are three nodes on the cluster and the workload is to distributed evenly among them. With this in mind, we split the view-frustum into three equal parts, and associate each with a different processor. When a patch is to be rendered, a check as to which section its *centre* lies within is made, and, on that basis, the work is assigned to one of the three processors. Each patch is only rendered once, as, having been allocated for rendering by a particular machine, it cannot be reassigned. This prevents overlap at the meeting points of the partitions. *Figure 3.2* below shows a snapshot of this method running, where the processing and rendering of each segment is allocated to a different machine. This could be modified to allow for the scaling of the cluster by changing the division of the angles and a number of related variables.

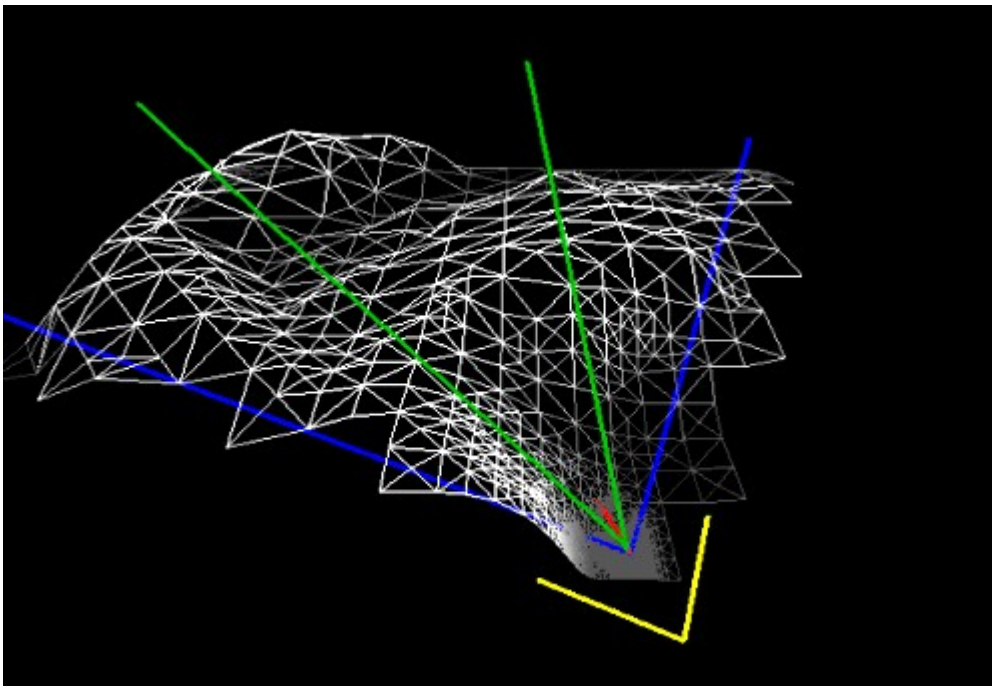

Figure 3.2: Parallelisation by Splitting the View Frustum

3.2.3 Further Difficulties

The desired image accuracy was an important consideration in the testing, as well as the frame count per unit time. Serious visual defects such as popping and cracking on the modelled structure could not be tolerated. Indeed, it was indicative of a problem when they did appear, since ROAM specifically operates against allowing that to happen. For this reason, problems with the applications which resulted in freezing, stripped and cracked terrains had to be sourced and eradicated. This took some time as the potential causes for such problems seemed an ever-growing, rather than shrinking, list as this debugging stage progressed.

Firstly there was the issue of swapping and flushing buffers at appropriate times; this cannot be allowed to occur more than once in a frame. Double buffering is a technique, implemented in both hardware and software, which supplies two complete buffers to avoid flicker - one is displayed while the other is being drawn. Flushing the buffers is function called after drawing which ensures all the data has been processed and sent to the display. A number of difficulties arose following the actual parallelisation, often stemming from simple problems but which sometimes took days to find. On one such occasion it was deduced the cause of blanking out and fragmented outputs was from the OpenGL code, which led to readjusting and redefining a number of viewing functions and matrices.

A week later it was discovered the cause had more to do with the placement

of the synchronisation calls, then with them changing the projection settings. The entire algorithm has been stripped down to almost a skeleton application a number of times, only to be rebuilt on finding the problem area. This involved a huge amount of tedious testing of fragments of code, but a learning experience nonetheless and the final products were satisfactory in effectively testing Chromium.

Chapter 4

Evaluation

The tests were conducted upon a cluster of three 450MHz Pentium II machines, each with 256 Mb RAM, and S3 Verg graphics cards. An Ethernet network is used between the nodes. Each processor runs a Red Hat version 7.3 of Linux. The first step was deciding how to measure the performance. Then a series of results are laid before us, along with their source and an analysis of their significance. The serial programs are presented first, followed by the parallel and several different tests with varied aims, comparing a number of these programs. Overall conclusions are then drawn from the implemented work.

4.1 How to Best Measure the Performance

The first decision to be made for testing was what to use as measurement. This led to researching a number of alternatives, including Chromiums' own support. Chromium has own performance testing SPU, *perfSPU*, which was

implemented and used in some of the test applications. This Stream Processing Unit takes snapshots of the current state of how a number of features are performing. Using either a *time* or *frame* based method, it obtains a dump of statistics, which can be saved in a log file. Unfortunately the statistics reported weren't really relevant for our purposes, so, although interesting merely from an academic point of view, they added nothing to the performance testing of the system.

The other principle methods considered were measuring the CPU usage, the time taken to execute an application and the frame rate. Since we are analysing the performance of parallel and serial programs, we are looking at ways to compare the speedup of the applications. $speedup = runtime\ of\ the\ fastest\ sequential\ algorithm/\ runtime\ of\ the\ parallel\ algorithm$

In parallel processing, efficiency is best measured as a function of speed, be it the run time, or the number of frames rendered per second, since the aim is to complete the execution as quickly as possible, not the effective use of the processors. The speedup of a parallelised application run on a parallel system is computed by comparing of its performance on a single machine to that on a multiprocessor system - cluster of not. Obviously, one can never achieve greater than a linear speedup, that is, for N processors, and constant c, a speedup of cN. This is rarely reached as overhead costs generally have a notable bearing.

It was decided that the frame rate per second would be the most suitable mea-

surement of the performance of the applications. This is a very well-known method of gauging graphics programs performance, which is relatively easy to implement in GLUT, OpenGL and MPI, which was necessary to traverse the range covered in the testing.

4.1.1 Frame Rates

The frame rate of the application varies from frame to frame, depending on the work of the Operating System, moving cameras, and changing dynamics within the application. For this reason, we concentrate on evaluating the frame rate per second, *fps*, and avoid the costly computing of the frame rate of each and every frame. For the applications running Chromium alone, with no additional message passing, OpenGL's Utility Toolkit, **G**LUT, provides a function, *glutGet()* which returns the time the application started and finished in milliseconds, when called at the appropriate stage. The frame rate can thence be evaluated. Using the same formula, the frame rate can be attained for the applications using just OpenGL, the Chromium only versions for instance, and those containing MPI, using the *MPI_Wtime()* function in place of the *glutGet()*.

4.2 Results

A huge number of experiments were run to evaluate the performance, image accuracy and scalability of Chromium. Each test consisted of a modified version of the ROAM algorithm for a fixed 50 frames. The key difficulty in

the testing was the graphics cards on the cluster. The easy upgrading of such components on cluster architecture has not been taken advantage of as the current cards, *S3 Verg*, are seriously outdated and produce diabolical frame rates. Though the processors are only 450 MHz Pentium II, the system is graphics, and not compute, limited as revealed in *Chapter 2*. For this reason, it is difficult to ascertain whether interactive frame rates really are being achieved. Testing an application on a windows machine with optimised Intel Brookdale G graphics cards, gives frame rates of up to 30 fps, where the same application, just slightly modified for Linux, merely achieves 8 - 10 fps (frames per second) run on a single processor of our cluster. Therefore, even though 10 - 20 and above are considered acceptable real-time frame rates, we have to re-adjust our parameters in this case, and allow for the probable fps of the applications, were the cluster updated. Using this as a rough translation guide, we can get an idea of the potential performance of the system.

4.2.1 Serial Versions

Running a serial version of ROAM, with GLUT and neither Chromium nor MPI *unnecessary considering this will execute as a single serial process*, gave frame rates of between 8 and 10 fps when executed in wire-frame mode. Wire-frame simply means that only the perimeter lines of the triangles are drawn, instead of filled solid or textured polygons. When texture and light are enabled, these rates fall to an average of *3.05* fps. This drastic plunge in fps led to a significant amount of the testing and comparisons being con-

ducted in wire-frame mode - since the other was so chronically slow. While methods to address this congestion in the rendering, specifically to manage texturing and lighting in the preprocessing stage, have been designed to some extent for Chromium, these are not yet fully operational.

To give an idea of the impact such lighting and texture have on different applications, tables *4.1* and *4.2* outline some of the results from appropriate test runs. These results were obtained with the number of Binary Triangle tessellations at 10,000 per frame. Subsequent testing alters this parameter, pushing the system to test its computing and rendering capabilities. The

	SerRoam	*serChrom*	*serMpi*
With Texture & Lighting Enabled	3.050	0.779	0.791
Wire-frame	8.45	6.385	6.277

Figure 4.1: Test results comparing overhead in Serial versions

above table, *4.1* outputs the results of three serial versions of the same underlying ROAM application, one consisting of just the algorithm, one with Chromium and a third with MPI. Its purpose is to show the overhead, and ensuing impact in performance, relating to each. serRoam is just a serial version of ROAM with GLUT, no Chromium, no MPI, Makefile uses g++ compiler and run with *./roam*, where *'roam'* is the executable file.

serChrom is a Chromium-based serial version of the same algorithm, with the GLUT removed - hence disabling user interaction with the program. This

means that Chromium code has been incorporated in the program, and the Makefile contains the pathname to the Chromium, and links the required libraries for this application. All three nodes play a part in each Chromium run, the mothership, server and application faker, as explained previously. Having compiled the program, it is run by starting the mothership,

*python*2 *serChrom.conf*

with the appropriate python script, serChrom.conf. This configuration file will contain all the required information for the run. The path to the program files, the number and location of crservers and crappfakers, *(in this case one of each, both run locally on cagnode12)* as well as their rank, size and order will all be supplied by this script. An example configuration file can be seen in Appendix A. The 'crappfaker' re-links the OpenGL library calls to the Chromium library and the 'crserver' renders it to the display tile. No visible defects were apparent, despite intensive testing of the application, so it was deemed to meet the visual accuracy standard. Adding a server node means two tiles are created *(from the AddTile() command, see Appendix A)* and the image is split between them, using the sort-first 'tilesort' SPU. This implements the theory behind tilesorting by producing a large image by tiling together two individually rendered images. Adding an application node has a detrimental effect on the performance as, because it is a serial program run on one processor, it simply duplicates the workload. The fps drop to 3.2 is evidence of this.

serMpi is another sequential version of ROAM, this time with MPI as well as

Chromium code incorporated. As it is serial, and if it is run as one process, there is no requirement for the MPI, but it is used to initialise the program, and assign rank and size. This is more for testing the overhead of using MPI than any real test of communication among processes for the stated reason. The first test program for MPI, *cpi.c*, simply initialised and issued some of the more basic commands. An opportunity to get Chromium and MPI working together and to evaluate the extent of the overhead of using message passing code arose with serMpi. When the mothership is started, it looks for the crserver and the crappfaker in the same manner as before *(one of each as above)*. This time, however, although the crserver is initialised as previously, the mpirun command: *mpirun −np* 1 *roam*, where *np* means the number of processes, *1* is the number in this run and *roam* is the executable file. launches the application in place of the crappfaker. Executing this application with two server and two application nodes results in a dire slowdown, from 6.2 to 2.8 fps, since the computation and rendering is replicated and all carried out on one machine, *cagnode12*.

4.2.2 Parallel Versions

Parallel applications running on *cagnode12* alone, shown in table emph4.2 below are noticeably slower in a sequential environment, comparing poorly to the serial versions outlined in table *4.1*. This is an analysis of the additional computational cost in dividing up the processing and rendering, but running them all on the same machine, as such, increases the workload, hence the inferior frame rate.

		parChrom	*parRnMpi*	*parMpi*	*par2Mpi*
With Texture & Lighting Enabled	1 app. node 1 ser. node	1.134	1.986	1.544	1.73
	2 app. nodes 2 ser. nodes	0.757	1.425	0.930	0.712
	3 app. nodes 2 ser. nodes	0.635	0.987	0.968	0.447
	3 app. nodes 3 ser. nodes	0.667	1.621	1.500	0.410
Wire-frame	1 app. node 1 ser. node	4.458	5.624	4.185	12.68
	2 app. nodes 2 ser. nodes	5.020	4.253	5.387	5.208
	3 app. nodes 2 ser. nodes	4.751	3.425	5.116	4.158
	3 app. nodes 3 ser. nodes	2.667	2.986	3.345	2.258

Figure 4.2: Test results comparing overhead in Serial versions

parChrom uses the first parallelising technique specified above - where the rendering of the patches is divided among the nodes of the cluster. The parallelising works beautifully, with a 'cycle' parameter altered with each call of the *Reset()* function, sending the patches to be rendered by the appropriate machine, as described in section 3.1, **??**. The above table only shows the results for running the programs locally on a single node, which means that the advantages of parallelising are not apparent. It is of interest, however, as a basis for evaluating the overhead costs in using Chromium, Chromium and MPI together and MPI alone associated with each. Inserting both Chromium and the MPI incur certain costs, as supported by the lower fps values obtained.

parRoamMpi is simply the first parallelised method, initialised with MPI rather than OpenGL commands. There is no Chromium in this version and

it is executed using *mpirun −np* 3 *roam*, where *3* is just a sample number of applications that could be run. As can be seen in the table, *4.2*, this achieves quite impressive results locally, as there isn't a huge amount of additional rendering required to use *mpirun* rather than the *./roam* execute command. **p**arMpi, which is parallelises ROAM using the first technique of the two most efficient, as with **p**arChrom, obtains the best performance rates locally as the number of server and application nodes increased - with an average of 4.508 in wire-frame mode, with parChrom and par2Mpi *(second parallelising technique referred to - divided view-port)* competitive. **p**ar2Mpi is a highly efficient method of dividing, but a glance at table *4.2* will suggest that the frame rate when running one server and one application node is slightly unusual, at 12.68 fps. This is accurate since this algorithm works on the premise of culling to the view-port associated with it. Hence, if there is only one application and server node, the algorithm will assign the correct subdivision of the viewing frustum to this machine, and not render the rest of the terrain outside these angles. This could be rectified by setting the angles dynamically rather than hard-coding the divisions directly. Not a very complicated task, but nonetheless a time-consuming one, which is the reason for not implementing this feature.

The performance results suggest that the overhead in having the program initialised and ordered by MPI as opposed to the OpenGL *main()* are negligible. When running on a single node, communication between the machines is obviously not an issue, nor is inter-process communication when run using

only one application node - the distribution of multi-processes is examined in the next part.

This initialises two processes, or three depending on the settings coded in the configuration script, and output checks prove the patch rendering alternating between the number of processes initialised. The difference in running them across the cluster instead of locally, as here, simply means that these processes are computed and/or rendered on other nodes, not just all on one, again this information is established in the configuration script, *parMpi.conf*. Again, there is an element of additional overhead involved as the parallel version is slower than its serial equivalent, as apparent from tables *4.1* and *4.2*

Dividing the processing and rendering across the cluster

To run MPI across the cluster, it is necessary to assign the appropriate machines for the application faker in the MPI-shared file: *machine.LINUX* and the location of the mothership, since it was set on *cagnode12*. This was done by hard-coding:
setenv("*CRMOTHERSHIP*", "*cagnode*12", 1);
into the programs to be run, just after initialisation in the main program. It is simply a time-saving method, since the alternative, actively exporting the mothership component to *cagnode12* is perfectly viable, just more time-consuming.

We can see immediately from table *4.3* that distributing the workload across the cluster yields improved performance of the applications, not only of the same applications run locally on one processor, but superior to the serial applications. This confirms the underlying theory of the project, that parallelising improves performance.

Different LODs - Scalability

This table also shows the results of the next stage of the project - which involved testing the system at different resolutions - the number of Binary Triangle tessellations per frame. Previous testing had this parameter set to 10,000. The table below, *4.3*, outputs the results gained from some of the more extreme rates tested, specifically examining fps at 1,000 and 25,000 tessellations per frame. It was discovered in the course of numerous test runs that all of the applications crashed at greater than 25,000 bintree tessellations per frame.

Tessellations	*serChrom*	*serMpi*	*parChrom*	*ParMpi*	*par2Mpi*
1000	12.923	12.804	15.09	16.345	12.389
10,000	6.385	6.277	7.177	7.730	6.000
25,000	4.436	4.45	5.386	4.772	4.409

Figure 4.3: Test results comparing overhead in Serial versions

These results are pictorially visualised in the chart, *chart* and the graph *4.5* below.

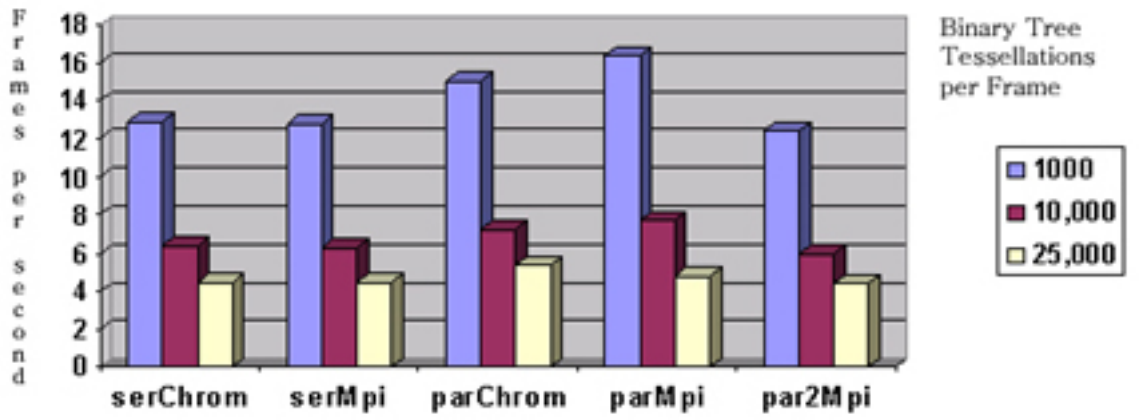

Figure 4.4: Comparisons of fps between different Serial & Parallel versions

4.3 Significance of Results

Ideally, increasing the number of nodes on the cluster would lead to a linear rise in the frame rate with no detriment of the image quality. This depends however on the complexity of the problem, since, with greater computational and rendering needs, the scalability is going to be closer to the theoretic linear improvement. However, with slightly less taxing applications, such costs as overhead communication, duplication efforts, additional overlap causing escalating polygonal throughput and I/O bandwidth requirements have a more discernible impact.

From the illustrated results, complementing further testing, we conclude that processing images with multiple PCs can be more efficient than processing images with only one machine. The improvement upon the performance due to the use of Chromium is also clear. The performance testing indicates that

Chromium does improve and facilitate the execution of intensive graphical applications across a cluster, in real-time. As indicated above, there is a slight overhead associated with using Chromium, as with MPI, but this is negligible when compared to the increase in the number of frames per second achieved the more intense applications distributed across the cluster. The chart, *4.4*, above and the graph, *4.5*, below attempt to clarify this by depicting the results in context of one another.

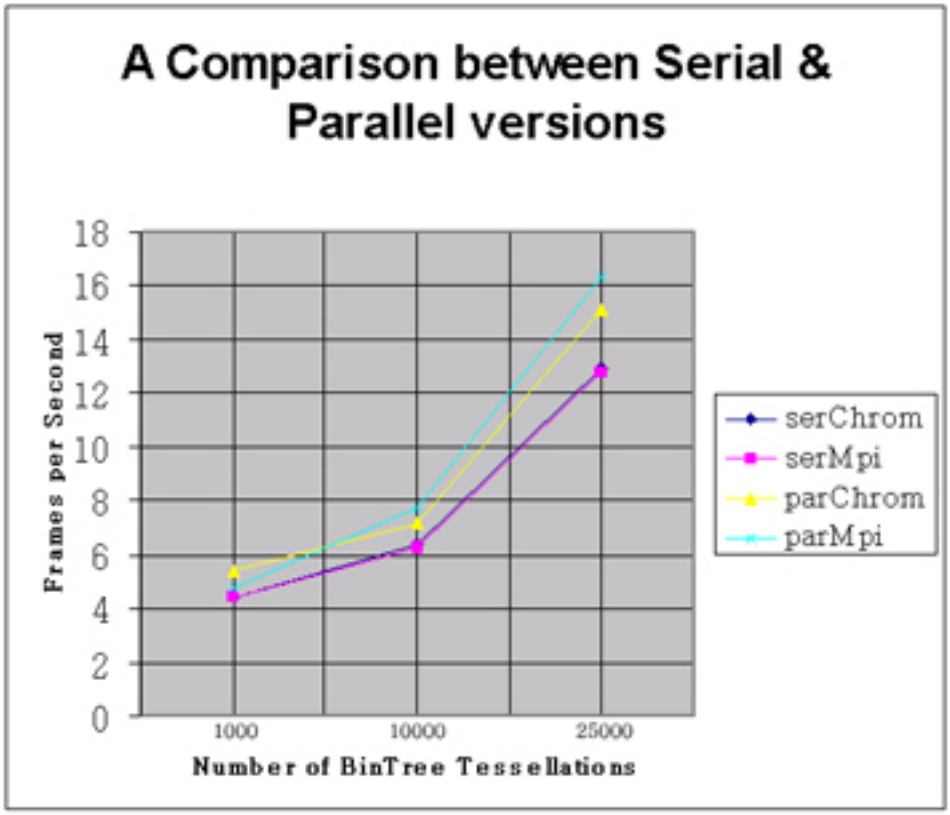

Figure 4.5: Test results comparing overhead in Serial versions

4.4 Conclusion

Using a cluster of inexpensive PCs to drive the display, a highly scalable graphics system capable of rendering multi-resolution images of large terrain visualisation models has been presented. Chromium, the scalable graphics rendering system used, is explored and tested in depth, the results showing a significant performance improvement. The generated models are capable of running at interactive frame rates and, allowing for upgraded graphics cards and a larger cluster, models of greater size and complexity could also be rendered. The theory of improving performance with a rendering system has been validated. Chromium enables graphics intensive parallel applications to execute in real-time on a modest-sized cluster. This means that a cluster of off-the-shelf components can be used to render models, which would conventionally require very expensive high-end graphics systems, at a far more affordable price.

4.4.1 Future Work

This research has shown promising results for implementing image processing techniques in a multiprocessing environment. There are a number of different aspects of the project that would make interesting further research areas. One of the features of Chromium which is capricious in this version, but is promised to be reliable in the next, is CRUT. Implementing CRUT, Chromium's Utility Toolkit, would allow user interaction with the applications in real-time. A greater level of detail management would test the system

more thoroughly. Optimisations which are not yet fully-developed in this version of Chromium, such as computing the texture and lighting at preprocessing, and implementing precise load balancing schemes, would also contribute significantly to the performance improvement. It would be interesting to consider ways of utilising different numbers of processors, maintaining the constraints on the original image, which would offer a more accurate testing of the of the capabilities of the cluster and facilitate locating the threshold number of processors that achieve speedup in this system. Another worthwhile project would be to use SCI (Scalable Coherent Interface), see [20] for further details, rather than Ethernet as the interconnect, since this has an enormous impact upon the scalability, as well as the capacity, of the cluster. upgrading to high-speed graphics cards on the processors of the cluster would give a far more realistic idea of the performance gain, as the current ones installed are dire and responsible for a graphics bottleneck in the system. A final application and worthy purpose of this project would be putting the Chromium, on the improved cluster, to a practical use, such as realising a CAVE. This is a cuboid room with tiled display projectors situated on all sides, of which there are currently only a few hundred in existence.

Appendix A

Configuration File

The following is an example of a configuration file, *A.1 and A.2*. Written in *python*, this is started with the mothership to supply all the required information for Chromium run. This particular file was designed to run a parallelised version of ROAM (the second method - dividing the view-port) across the cluster. It runs three server nodes and three application nodes, the computing and the rendering divided out across the cluster with one of each node running on each machine.

```
# Demonstrate parallel submission with sort-first rendering.

import sys
sys.path.append( "../server" )
from mothership import *

Demo = '/home/am/par2Mpi/roam'

TileWidth = 320
TileHeight = 480

# Set up the first server node
servernode1 = CRNetworkNode(  )
renderspu = SPU( 'render' )
renderspu.Conf( 'window_geometry', [0, 0, TileWidth,
TileHeight] )
servernode1.AddSPU( renderspu )
servernode1.AddTile( 0, 0, TileWidth, TileHeight )

# Set up second server node
servernode2 = CRNetworkNode( 'cagnode13'}
renderspu = SPU( 'render' )
renderspu.Conf( 'window_geometry', [TileWidth+5, 0,
TileWidth, TileHeight] )
servernode2.AddSPU( renderspu )
servernode2.AddTile( TileWidth, 0, TileWidth, TileHeight )

# Set up third server node
servernode3 = CRNetworkNode( 'cagnode14' )
renderspu = SPU( 'render' )
renderspu.Conf( 'window_geometry', [TileWidth+50, 0,
TileWidth, TileHeight] )
servernode3.AddSPU( renderspu )
servernode3.AddTile( TileWidth, 0, TileWidth, TileHeight )

# Set up first app/client node (note -clear and -swap)
appnode1 = CRApplicationNode(  )
appnode1.SetApplication( '%s -rank 0 -size 2 -clear -swap' %
Demo )
appnode1.StartDir('/home/am/par2Mpi/')
spu = SPU('tilesort')
spu.Conf('fake_window_dims',[TileWidth*2,TileHeight])
appnode1.AddSPU( spu )
spu.AddServer( servernode1, protocol='tcpip', port=7000 )
spu.AddServer( servernode2, protocol='tcpip', port=7001 )
spu.AddServer( servernode3, protocol='tcpip', port=7002 )
```

Figure A.1: Configuration File for Parallelised Method II

```
# Set up second app/client node
appnode2 = CRApplicationNode( 'cagnode13' )
appnode2.SetApplication( '%s -rank 1 -size 2' % Demo  )
appnode2.StartDir('/home/am/par2Mpi/')
spu = SPU('tilesort')
spu.Conf('fake_window_dims',[TileWidth*2,TileHeight])
appnode2.AddSPU( spu )
spu.AddServer( servernode1, protocol='tcpip', port=7000 )
spu.AddServer( servernode2, protocol='tcpip', port=7001 )
spu.AddServer( servernode3, protocol='tcpip', port=7002 )

# Set up third app/client node
appnode3 = CRApplicationNode( 'cagnode14' )
appnode3.SetApplication( '%s -rank 2 -size 3' % Demo  )
appnode3.StartDir('/home/am/par2Mpi/')
spu = SPU('tilesort')
spu.Conf('fake_window_dims',[TileWidth*2,TileHeight])
appnode3.AddSPU( spu )
spu.AddServer( servernode1, protocol='tcpip', port=7000 )
spu.AddServer( servernode2, protocol='tcpip', port=7001 )
spu.AddServer( servernode3, protocol='tcpip', port=7002 )

cr = CR()
cr.MTU( 16*1024 )
cr.AddNode( servernode1 )
cr.AddNode( servernode2 )
cr.AddNode( servernode3 )
cr.AddNode( appnode1 )
cr.AddNode( appnode2 )
cr.AddNode( appnode3 )
cr.Go()
```

Figure A.2: Configuration File for Paralysed Method II, continued

Bibliography

[1] The history of mpi. http://www.erc.msstate.edu/misc/mpi/.

[2] http://sjbaker.org/steve/software/glut_hack.html. Date accessed: 26nd February 2003.

[3] http://spanky.triumf.ca/www/fractint/fractint.html. Date accessed: 4th January 2003.

[4] http://www.fractalus.com/galleries/index-thumbs.htm. Date accessed: 4th January 2003.

[5] Multi-monitor tiled display, Date accessed: February, 2003.

[6] J. Glenn Brookshear. *Computer Science, an Overview*. Addison-Wesley, 6th, s-len 500.164 m84*5;1 edition, 2000.

[7] Kenneth Butler. http://www.kpbsd.k12.ak.us/kchs/jimdavis/calculusweb/sierpinski's%20tri
Date accessed: 27th April, 2003.

[8] Wagner T. Correa, James T. Klosowski, and Claudio T. Silva. Out-of-core sort-first parallel rendering for cluster-based tiled displays. 2002.

[9] Willem H. de Boer. Fast terrain rendering using geometrical mipmapping. *E-mersion Project, October 2000, www.connectii.net/emersion*, 2000. http://www.flipcode.com/tutorials/geomipmaps.pdf.

[10] Mark A. Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. *ROAMing terrain: real-time optimally adapting meshes.* 1997. citeseer.nj.nec.com/duchaineau97roaming.html.

[11] CRUT The Cluster Rendering Utility Toolkit for Chromium. *Dale Beerman.* 2002.

[12] John Hamill. Real-time terrain rendering. May 2001.

[13] Greg Humphreys, Matthew Eldridge, Ian Buck, Gordon Stoll, Matthew Everett, and Pat Hanrahan. Wiregl: A scalable graphics system for clusters. http://graphics.stanford.edu/papers/wiregl/.

[14] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: A stream-processing framework for interactive rendering on clusters. citeseer.nj.nec.com/humphreys02chromium.html.

[15] David Jones. How wiregl & tiled display work, Accessed March 2003.

[16] Faye A. Briggs Kai Hwang. *Computer Architecture and Parallel Processing.* McGraw-Hill International Edition, 1985.

[17] Project leader: Dr. Mo Jamshidi. vlab.unm.edu/documents/terrain_model.ppt. Virtual laboratory for Autonomous Agents.

[18] P. Lindstrom, D. Koller, W. Ribarsky, L. Hodges, N. Faust, and G. Turner. Real-time continuous level of detail rendering of height fields. *Proceedings of SIGGRAPH'96*, pages 109–118, 1996. citeseer.nj.nec.com/article/lindstrom96realtime.html.

[19] David P. Luebke. A developer's survey of polygonal simplification algorithms. *IEEE Computer Graphics and Applications*, 21(3):24–35, / 2001.

[20] Michael Manzke. http://www.cs.tcd.ie/michael.manzke/research.html. January, 2003.

[21] Bryan Turner. Real-time dynamic level of detail terain rendering with roam. 2000. http://www.gamustra.com/features/20000403/turner-01.htm.