Terms and Conditions of Use of Digitised Theses from Trinity College Library Dublin

Copyright statement

All material supplied by Trinity College Library is protected by copyright (under the Copyright and Related Rights Act, 2000 as amended) and other relevant Intellectual Property Rights. By accessing and using a Digitised Thesis from Trinity College Library you acknowledge that all Intellectual Property Rights in any Works supplied are the sole and exclusive property of the copyright and/or other IPR holder. Specific copyright holders may not be explicitly identified. Use of materials from other sources within a thesis should not be construed as a claim over them.

A non-exclusive, non-transferable licence is hereby granted to those using or reproducing, in whole or in part, the material for valid purposes, providing the copyright owners are acknowledged using the normal conventions. Where specific permission to use material is required, this is identified and such permission must be sought from the copyright holder or agency cited.

Liability statement

By using a Digitised Thesis, I accept that Trinity College Dublin bears no legal responsibility for the accuracy, legality or comprehensiveness of materials contained within the thesis, and that Trinity College Dublin accepts no liability for indirect, consequential, or incidental, damages or losses arising from use of the thesis for whatever reason. Information located in a thesis may be subject to specific use constraints, details of which may not be explicitly described. It is the responsibility of potential and actual users to be aware of such constraints and to abide by them. By making use of material from a digitised thesis, you accept these copyright and disclaimer provisions. Where it is brought to the attention of Trinity College Library that there may be a breach of copyright or other restraint, it is the policy to withdraw or take down access to a thesis while the issue is being resolved.

Access Agreement

By using a Digitised Thesis from Trinity College Library you are bound by the following Terms & Conditions. Please read them carefully.

I have read and I understand the following statement: All material supplied via a Digitised Thesis from Trinity College Library is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of a thesis is not permitted, except that material may be duplicated by you for your research use or for educational purposes in electronic or print form providing the copyright owners are acknowledged using the normal conventions. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone. This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.

Exploiting Commodity Parallel Hardware for Computer Graphics Applications and Architectures

A Thesis

Submitted to the Office of Graduate Studies

of

University of Dublin, Trinity College

in Candidacy for the Degree of

Doctor of Philosophy

by Keith O'Conor

April 2006

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work. This thesis may be borrowed or copied upon request with the permission of the Librarian, Trinity College, University of Dublin. The copyright belongs jointly to the University of Dublin and Keith O'Conor.

Keith O'Conor

January 29, 2007

For Elaine

Acknowledgments

First and foremost, my sincerest thanks to my supervisor Carol O'Sullivan. Carol has never been anything less than enthusiastic and supportive of all my endeavours here at the ISG, and I would not be where I am today without her.

The ISG has been a great place to work and live over the last four years. The people I have met here have become good friends, and I will not soon forget them. My thanks go to John Dingliana for taking on the final year project that led to my position in the ISG. Thanks also to Simon for the engaging work we have done together, and for all the entertaining chats about everything from graphics to jazz. I also owe a big thanks to Andrew Brosnan for his late-night proofreading during the final crunch.

To my parents and my brother Hugh who have always believed in me, even when I didn't believe in myself. They are a constant source of encouragement, strength and inspiration.

Finally, to my friend and my love, Elaine. She makes it all worthwhile.

KEITH O'CONOR

University of Dublin, Trinity College
April 2006

Related Publications

Isosurface Extraction on the Cell Processor

K. O'Conor, C. O'Sullivan and S. Collins.

Proceedings of Eurographics Ireland Workshop, October 2006

A Scalable and Reconfigurable Shared-Memory Graphics Architecture

M. Manzke, R. Brennan, K. O'Conor, J. Dingliana and C. O'Sullivan.

Proceedings of ACM SIGGRAPH 2006 Sketches & Applications, 2006, p. 182

Geopostors: A Real-Time Geometry/Impostor Crowd Rendering System

S. Dobbyn, J. Hamill, K. O'Conor and C. O'Sullivan.

ACM Transactions on Graphics, 24(3), 2005

Geopostors: A Real-Time Geometry/Impostor Crowd Rendering System

S. Dobbyn, J. Hamill, K. O'Conor and C. O'Sullivan.

Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games 2005, pp. 95 - 102

Perceptually Adaptive Graphics

C. O'Sullivan, S. Howlett, Y. Morvan, R. McDonnell and K. O'Conor.

Eurographics State of the Art reports, September 2004. pp. 141-164. Also to appear in Computer Graphics Forum.

3D Visualisation of Confocal Fluorescence Microscopy Data

K. O'Conor, H.P. Voorheis and C. O'Sullivan.

Proceedings of Eurographics Ireland Workshop 2004, pp. 49 - 54 (Best Paper Award)

Abstract

The processing power available in today's commodity parallel hardware has enabled realism and detail in graphics that has never before been possible. With the advent of the programmable Graphics Processor Unit (GPU), the full potential of parallel architectures to accelerate graphics algorithms has become apparent. This parallelism is becoming more ubiquitous in other processors, and research that exploits this parallelism is ongoing.

In this thesis we apply the knowledge learned of rendering clusters to the design of a new tightly-coupled cluster architecture for parallel rendering, and describe a software infrastructure for implementing distributed rendering by taking advantage of the unique mix of parallel hardware available. We then concentrate on the application of this commodity parallel hardware to two important fields of computer graphics applications; scientific visualisation and entertainment.

Under scientific visualisation, we describe the use of the programmable pipeline for direct volume rendering of datasets captured by confocal fluorescence microscopy, as well as introducing a simple method for fast volumetric simplification which allows broad feature preservation while allowing faster isosurface extraction and noise reduction when applied to confocal datasets. We also introduce a novel algorithm for performing isosurface extraction on Cell, the recently developed high-profile multicore processor from IBM, Sony and Toshiba. We give an overview of the processor and detail how to exploit it for algorithmic acceleration.

In the field of entertainment applications, we describe the use of the programmable graphics pipeline to accelerate and improve the rendering of impostor-based crowds made up of a large number of virtual humans. We discuss the shortcomings of previous methods when applied to state of the art graphics hardware, and detail a new algorithm that can be applied to achieve superior results.

Contents

List of Figures

List of Acronyms

AGP	Accelerated Graphics Port
API	Application Programming Interface
ARB	OpenGL's Architecture Review Board
ASIC	Application-Specific Integrated Circuit
BAR	Base Address Registers
BEI	Broadband Engine Interface
CBE	Cell Broadband Engine
COTS	Commodity Off-The-Shelf
CP	Command Processor
CPU	Central Processing Unit
CT	Computerised Tomography
DLL	Dynamic Link Library
DMA	Direct Memory Access
DSM	Distributed Shared Memory
DVI	Digital Visual Interface
DVR	Direct Volume Rendering
EIB	Element Interconnect Bus
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GART	Graphics Address Remapping Table
GLSL	OpenGL Shading Language
GPGPU	General-Purpose Computation on GPUs
GPU	Graphics Processing Unit
HLSL	Direct3D's High Level Shading Language
HPC	High-Performance Computing

IBR	Image Based Rendering
LC	Link Controller
LOD	Level Of Detail
LRU	Least Recently Used
MFC	Memory Flow Controller
MIC	Memory Interface Controller
MIMD	Multiple Instructions Multiple Data
MMIO	Memory Mapped Input/Output
MPI	Message Passing Interface
MRI	Magnetic Resonance Imaging
MT	Marching Tetrahedra
NUMA	Non-Uniform Memory Access
PCI	Peripheral Component Interconnect
PIO	Programmed Input/Output
PLD	Programmable Logic Device
PPE	PowerPC Processor Element
PPU	PowerPC Processor Unit
PSGL	Playstation Graphics Language
PVM	Parallel Virtual Machine
RISC	Reduced Instruction Set Computer
SCI	Scalable Coherent Interface
SDK	Software Development Kit
SIMD	Single Instruction Multiple Data
SMP	Symmetric Multiprocessing
SPE	Synergistic Processor Element
SPU	[Cell] Synergistic Processor Unit
SPU	[Chromium] Stream Processing Unit
SRAM	Static Random Access Memory
TCL	Transform, Clipping and Lighting
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

Chapter 1

Introduction

Since the introduction of the modern integrated circuit, computing power has been increasing rapidly due to improvements in technology and manufacturing methods. Furthermore, this power does not come at a commensurate price. Quite the opposite is true; in 1997 the equivalent price of one gigaflop of power was $30,000 [140]. Today in 2006, that price has dropped to under $1 in the case of the most recent Graphics Processor Units (GPUs).

The widespread availability and low price of computing power has been the cause of a major boom in personal computing and the consequent ubiquity of desktop workstations seen today. Even a moderate desktop computer is now capable of billions of floating point operations per second - orders of magnitude faster than a machine that would have been termed a 'supercomputer' 30 years ago. The low cost of this hardware enables ordinary commodity off-the-shelf (COTS) systems to perform complex simulations in real-time at interactive rates to a degree that was never before possible without extremely expensive hardware or dedicated custom architectures.

However, this continuing growth is not just because of increasing chip speeds and decreasing costs. Indeed, the physical limitations of current manufacturing techniques are becoming apparent as the performance gains that can be had by further miniaturisation start to reach diminishing returns. Therefore a different approach is being turned to in order to supply the need for ever-increasing processing speed. Modern systems can exploit parallelism at many levels in order to scale to

multiple processors and take advantage of the inherently parallel nature of many algorithms.

1.1 Parallelism

The key concept of parallel computing is the decomposition of a problem into discrete components that can be solved individually. This can be carried out in two forms, classified by where in the system the parallelism is implemented.

Implicit parallelism, where the system automatically partitions work to be distributed among processors, can be seen in systems such as the GPU pipeline and certain SIMD-optimising compilers. In this case the developer does not need to specify any details of how work will be segmented. However, for performance-oriented applications it still helps to take it into consideration at the design stage, in order to allow the work to be parallelised as efficiently as possible.

On the other hand, explicit parallelism can be seen in areas such as multithreaded applications or distributed systems. In this case, developers must specifically design the application around the distribution model, taking into account all the performance implications and communication restrictions that accompany the underlying architecture in order to perform the maximum amount of computation at any stage.

Although explicit parallelism is not a new area *per se*, its use in modern processors is beginning to become more important than ever before. Multicore and parallel architectures such as the new Cell processor necessitate a fundamental shift in system design, requiring applications to be developed with new approaches to system usage. Just as cache access behaviour is a concern for the efficient usage of sequential processors, factors such as data distribution, synchronisation and memory access latency become an important consideration for the efficient use of parallel systems.

1.2 Context and Scope

The work contained in this thesis began as part of a HEA-funded project: the Institute for Information Technology and Advanced Computation (IITAC), with the author's particular remit being to study cluster-based rendering frameworks (such as Chromium - see Section 2.2.3) and their use for scientific visualisation. Specifically, the study of the protozoa Trypanosoma Brucei was to be the subject of visualisation in association with the Cell Membrane Group in Trinity College Dublin. Interactive visualisation necessitated the investigation of programmable graphics hardware in order to allow real-time rendering of the datasets being captured by the confocal microscopes used to study this organism.

While this research was being carried out, it was recognised that work being carried out by others on the same project, on crowds and virtual human rendering, could benefit greatly from the knowledge gained of graphics hardware. This was applied to accelerate crowd rendering and enable large numbers of humans to be simulated in real-time.

From this earlier work, a new project arose in association with the Computer Architecture Group (CAG) and funded by Science Foundation Ireland (SFI). This project aims to build upon the knowledge of clusters and commodity graphics hardware gained in the previous research to produce a new hardware framework that incorporates the advantages of both architectures. It also encompasses knowledge gained by researchers in the CAG in relevant areas such as Field Programmable Gate Arrays and the Scalable Coherent Interconnect in order to produce a new parallel cluster architecture for distributed rendering and simulation. The author's particular remit in this project was the investigation of the required software infrastructure and the exploitation of the framework for different types of applications, in particular, entertainment and scientific visualisation.

Most recently, an opportunity arose to investigate the application of the new Cell processor for accelerating graphics algorithms. From the experience gained in both volume visualisation of microscopy data and the use of parallel graphics hardware, it was recognised that this new parallel architecture could be used to substantially accelerate the area of surface extraction, necessary for isolating particular structures inside a dataset.

Therefore, the following chapters present the research performed in these projects. Specifically, the parallel architectures that were used and their application to the areas of scientific visualisation (volume rendering and surface extraction) and entertainment (crowd and virtual human rendering).

1.3 Contributions

In this thesis, we will look at how both implicit and explicit parallel systems can be employed in order to accelerate existing graphics algorithms. The major contributions are as follows:

- An overview of a variety of parallel architectures including both distributed systems and parallel processors. We investigate their exploitation for accelerating graphics algorithms and discuss their use in previous research.
- Details of a new hardware cluster for accelerating distributed rendering. We describe the low-level mechanism of existing graphics drivers and outline a **software infrastructure** to exploit the unique assortment of heterogeneous parallel hardware in the cluster to provide acceleration of both simulation and rendering algorithms.
- The use of the programmable pipeline for **direct volume rendering** of datasets captured by confocal fluorescence microscopy. We also introduce a simple method for quick **volumetric simplification** which allows broad feature preservation while allowing faster isosurface extraction and noise reduction when applied to confocal datasets.
- A description of a novel algorithm for performing **isosurface extraction on Cell**, the recently developed high-profile multicore processor from IBM, Sony and Toshiba. To our knowledge this is one of the first applications to be published which is aimed specifically at this new parallel architecture. We also give an overview of the processor and detail how to exploit it for algorithmic acceleration.

- The use of the programmable graphics pipeline to accelerate and improve the rendering of **impostor-based crowds** made up of a large number of virtual humans. We discuss the shortcomings of previous methods when applied to state of the art graphics hardware and detail a new algorithm that can be applied to achieve superior results.

1.4 Summary of Chapters

The rest of this thesis is divided up into the following chapters:

Chapter 2 describes the different parallel architectures we will be exploring throughout the rest of the thesis; GPUs, commodity clusters, FPGAs and the Cell Broadband Engine. Related research in each area is also presented, with a particular focus on their applications to graphical algorithms.

Chapter 3 presents a new hardware framework built at Trinity College Dublin in association with the Computer Architecture Group. We describe the underlying architecture, composed of custom-built FPGA-based boards attached to commodity graphics cards and connected by an SCI interconnect with distributed shared memory. We also discuss the inner workings of a graphics driver based on technical specifications supplied by ATI, and propose a suitable software infrastructure for exploiting such an architecture.

Chapter 4 explores the area of scientific visualisation. We give an overview of volume rendering, both direct and indirect, before describing the contributing work of this thesis in the areas of volume rendering, volumetric simplification and isosurface extraction on commodity parallel hardware.

Chapter 5 details the use of commodity parallel hardware in the field of entertainment applications. We concentrate on crowd rendering, exploring previous methods and describing an algorithm for producing hardware-optimal lit and varied humans that can be used in a hybrid impostor/geometry crowd system.

Chapter 6 provides a summary of these contributions, as well as a discussion of future work and the direction of commodity parallel hardware in general.

Chapter 2

Background and Related Work

There are many different levels of parallel architectures. Some exhibit parallelism by distributing work to discrete components of the system. Others employ internal parallelism; the job is broken up into steps to be processed simultaneously in a pipelined fashion. However, all architectures are advancing rapidly due to improvements in design and manufacturing methods.

This chapter looks at four distinct parallel hardware architectures, including distributed cluster systems and parallel chips. It gives an overview of each and details previous research performed with each architecture in relation to graphics algorithms.

2.1 Graphics Hardware

In the early days of real-time 3D graphics, the CPU was required to handle all transformations and rasterisation required to produce a final rendered image. Given the limited amount of computational power available, much of the processing time was consumed by these costly operations, leaving less time for simulation and other processing required to generate the 3D data in the first place. Additionally, having the CPU perform rasterisation placed an extra burden on memory bandwidth, which was required to access and update a software frame buffer and depth buffer. The need for specific dedicated hardware was obvious; a Graphics Processing Unit (GPU) co-processor could offload the cost of these operations from the CPU.

2.1.1 Hardware Acceleration

3D graphics acceleration through dedicated commodity hardware is not a new topic. The first generation of affordable commodity 3D accelerator cards arrived in 1996 with the widespread adoption of 3Dfx's Voodoo range of expansion cards. When the price of memory dropped substantially in the same year, these cards could finally be manufactured and sold at affordable prices. Soon competitors such as ATI and NVIDIA were producing similar products. With upwards of 4MB of video memory, these cards were able to take over rasterisation from the CPU, allowing 16-bit frame buffers and depth buffers. Soon afterwards they were also performing hardware primitive assembly, given transformed vertices by the CPU.

More importantly, they performed texture mapping and texture filtering. These operations were severely limited in software implementations that required real-time frame rates, because of the large amount of processing and bandwidth required to filter and project a texture of reasonable size in 3D. The improvements in image quality and rendering speed given by these cards were immediately apparent and 3D graphics hardware was soon a prerequisite for many games and other 3D applications.

After texturing and rasterisation, the next step in hardware acceleration was to perform vertex transformation, clipping and lighting (TCL). NVIDIA's GeForce range was the first to introduce this feature, thus moving all of the graphics pipeline (see Section 2.1.3) processing into hardware. This relegated the CPU to the role of submitting vertex and texture data to the GPU, freeing more CPU cycles to devote to application areas that are non-specific to graphics, such as physics, artificial intelligence, scene graph management etc. Initially there was some concern over the advantages of hardware TCL, due to benchmark performances being well below that of software CPU performance. However, as hardware improved and mesh sizes correspondingly increased, full utilisation of hardware TCL yielded better performance than equivalent software implementations ever could.

2.1.2 Embarrassingly Parallel

An often quoted metric of the recent rate of improvement in GPU power is "Moore's Law cubed". Moore's Law [93] is based upon the empirical observation that, due

to the rate of increase in circuit complexity versus size and cost, computing power doubles roughly every 18 months[1]. Correspondingly, GPU speeds double roughly every six months. This is due to a combination of increasingly better manufacturing processes, 3D-specific algorithmic advances, but most importantly the nature of the computations taking place.

In parallel computing terms, 3D graphics rendering is an embarrassingly parallel problem. In other words, it is a problem that can be easily divided into many steps, each step having little or no effect on the computation of other steps. These steps can therefore be worked on in parallel, the results being combined to form the solution. The GPU exploits this parallelism in acting as a stream processor [139]. A stream processor operates by scatter/gather; data is gathered from disparate sources (usually random or sequential blocks of memory), fed through one or more computational kernels and then scattered back to memory. Each kernel performs the same operation on every part of the stream that passes through it. In the case of the GPU, the kernels are the vertex processor and the pixel processor, as described in Section 2.1.3.

In comparing this to a CPU's need to perform every instruction in sequential order, we get an insight into why the GPU can greatly out-perform the general purpose processor - even ignoring the obvious speed advantages of parallel processing over serial processing. Pushing CPU speeds higher and higher places a strain on the speed at which memory can be accessed, as advances in memory latency have not been in keeping with those of the CPU for a long time [145]. The fact that every instruction and piece of data must share the same path to memory exacerbates the problem. To alleviate this growing speed/latency gap, the CPU needs to devote larger amounts of chip area to cache, to the point that the Pentium 4 actually contains more cache than it does logic.

On the other hand, the nature of the GPU allows many operations to happen in parallel, each coming from a dedicated path in the stream and leaving via another dedicated path - in many cases (texture lookups being the obvious exception) these operations need little or no memory access. Cache can therefore be kept to a mini-

[1]It should be noted that the original article to which the law is attributed never mentions the commonly accepted timespan of 18 months. Instead, Moore estimated that transistor counts would double approximately every two years, a figure which has proven in retrospect to be much more accurate. Nevertheless, the most commonly accepted usage of the Law refers to an 18-month cycle.

mum; even on the latest GPUs, the texture cache is no more than a few kilobytes. Thus with less cache, more chip area can be devoted to processing and the efficiency of the entire chip increases as a result.

The latest graphics chip currently on offer is ATI's R580, which powers the X1900 XTX boards and is comprised of 8 vertex pipelines and 48 pixel pipelines. The general trend is for games and applications to perform much more computation on fragments than on vertices, hence the imbalance. This allows unprecedented rendering parallelism and results in a theoretical peak of 10.4 gigapixels per second.

2.1.3 The Graphics Pipeline

The graphics pipeline is the process that every polygon goes through to become a pixel or group of pixels in the frame buffer (see Figure 2.1). It is broken down into the following categories:

Vertex Processing: Given a model-space vertex as an input, the vertex processor applies matrix transformations in order to output a screen-space vertex. If performing per-vertex lighting, the vertex processor is responsible for evaluating the lighting equation at each vertex. It also performs per-vertex colour application and any necessary normal transformation or texture coordinate generation.

Primitive Assembly: Each vertex sent through the vertex processor also has an edge flag associated with it. These edge flags specify which polygon the vertex belongs to and describe its connectivity with the other vertices of that polygon. From these parameters a polygon is constructed and forwarded for rasterisation.

Rasterisation: Rasterisation converts the continuous polygons from the primitive assembly stage into discrete fragments. A fragment can be thought of as a 'potential pixel', with the attributes of colour and depth - it has made it as far as being rasterised, but still may be removed from the pipeline by pixel tests as detailed below. This stage is also responsible for interpolating per-vertex attributes (such as texture coordinates, colour etc.) across the polygon assembled by the previous stage.

Fragment Processing: The fragment processor's purpose is to calculate the final colour of the fragment. It usually does this by combining texture lookups with the interpolated vertex colours.

Pixel tests: Finally, the fragment is subject to a series of pixel tests to determine if it should end up in the frame buffer. These are tests such as the alpha test, depth test, stencil test etc.

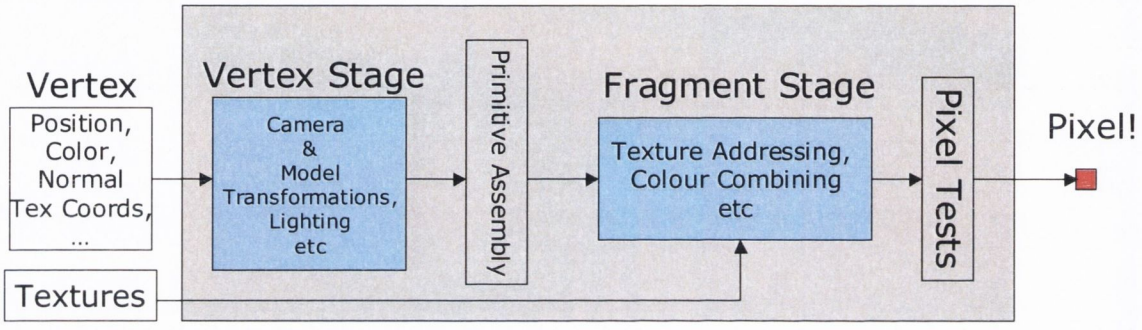


Figure 2.1: The Graphics Pipeline

Fixed Function Graphics Pipeline

The system of processing every vertex and pixel in hardware via a non-configurable pipeline implementation is referred to as the *fixed function pipeline*. In hardware it is more efficient to implement and easier to optimise a pipeline of specific operations in a specific order than it is to implement generalised logic. This was originally the way graphics hardware was able to achieve sufficient 3D acceleration.

There are two main APIs that the graphics card vendors support for interfacing with the hardware through their drivers - OpenGL and Direct3D. While the underlying hardware uses the same implementation for both APIs, the feature set and language semantics differ from version to version. However, the most noticeable difference between the two is their release dates. OpenGL tends to update its core functionality relatively infrequently, preferring a hierarchical extension mechanism that puts new additions to the API through a series of architectural reviews before they get promoted to the core. This more stringent process is a result of the differing interests expressed by the multiple members that make up the OpenGL

Architecture Review Board (ARB). On the other hand, Microsoft has complete autonomy in deciding which features get included in Direct3D. This results in more regular Direct3D releases, usually accompanying the release of a new graphics card generation. In any case the underlying hardware functionality is always the same, it is just exposed differently by the two APIs.

In the case of the core OpenGL graphics API up until version 1.5 [119], the fixed function pipeline was the only option for rendering with hardware acceleration. It was essentially a black box - it had various parameters to alter properties like transformations, materials and texture attributes, but how these properties were applied to the scene for rasterisation was set in hardware and could not be altered.

In the fixed function vertex processor, the only option for lighting was to use Blinn-Phong lighting [10] combined with Gouraud shading [44]. This model played to the strengths of the hardware - only evaluating the lighting equation at each vertex and using the fast hardware interpolation needed to colour each point on the surface. Despite this method missing specular highlights on low-tesselated models, this was an acceptable model for most 3D applications. Similarly, the fixed function fragment processor only allowed limited methods of texture lookups, a fixed set of blending modes and no interaction with the depth buffer.

The result of a single unalterable rendering equation was that every rendered scene always had the same lighting and the same feel to it. It did not allow for variation in the projection of vertices, nor did it give the developer much control of texture addressing or material application, beyond the limited parameters of the OpenGL rendering equation. For this reason, the fixed function processors were discarded and replaced with new programmable engines once GPUs became powerful enough.

Programmable Pipeline

The switch to the programmable graphics pipeline was a gradual one. Cards did not suddenly change from having fixed function processors to programmable ones, but rather the changes came in increments.

The first manufacturer to take a step toward general programmability was again NVIDIA with their GeForce cards. As well as introducing hardware TCL in 1999,

the GeForce 256 also had the first implementation of what NVIDIA called *register combiners* [123]. Register combiners were to be the first step towards generalising the fragment processing stage, allowing a programmable process to decide the final colour of a pixel. They consisted of a chain of 4-input combiner units, each combiner being able to perform operations such as multiplication, addition or dot products on the data being passed through. Additionally, a final combiner could perform interpolation on the outputs of the previous stages. The power of register combiners came from their configurability - the input of any combiner could be the output of any earlier combiner in the chain. While the number and nature of the operations were restrictive, it still allowed a great flexibility on how colours and textures could be combined to produce a fragment.

Soon afterwards NVIDIA produced *texture shaders* [33], a superset of conventional OpenGL texture operations which were to supplement the register combiners by adding extra texture addressing operations. In addition to the regular 1D, 2D and 3D texture addressing available in OpenGL 1.2.1 (the latest version at the time), texture shaders added cube mapping, dependent texturing, offset texturing and dot product texturing. Together with the register combiners, texture shaders allowed the fragment processing stage to be much more flexible than the core OpenGL specification allowed.

The following year saw the introduction of the vertex program [146]. Vertex programs introduced complete programmability of the vertex processing stage through an assembly language interface composed of special GPU instructions. This was the first truly programmable part of the graphics pipeline, completely bypassing the fixed function vertex processing stage. However, some restrictions were still in place. As in the fixed function pipeline, for every untransformed vertex submitted to the vertex processor, one transformed vertex was output. This was in keeping with the stream processing model, where the processors only affect the stream data being passed through and do not actually insert new data into the stream. Vertex creation and deletion therefore was not possible, and the vertex program written was required to do the vertex transformation manually and output a transformed vertex. Optionally, it could perform any other arbitrary calculations to generate texture coordinates, perform vertex lighting, and output any other attributes to be interpolated across the polygon during rasterisation.

The corresponding extension for the fragment processor was the fragment program [56]. In a similar fashion to the vertex program, the fragment program was a set of assembly language instructions which was given complete control of the fragment processing stage in order to write the final colour of the fragment being processed. Attributes from the vertex processor which were interpolated by the rasteriser are passed into the fragment program as parameters.

Shader development was not easy given the assembly language interface to vertex and fragment programs. Any shader of even moderate complexity was hard to read and even harder to debug, and any modularity that might help code reuse was difficult to maintain. For these reasons it was not long before high level shading languages were designed; the OpenGL Shading Language (GLSL) [68] and for Direct3D the High Level Shading Language (HLSL). These are high level shading languages, both loosely based on the syntax of the C programming language with extra vector types to take into account the vector-based nature of the underlying hardware. The driver then compiles these shading languages into hardware calls suitable to the hardware that the application is being run on. NVIDIA has also developed another shading language called Cg [98, 40]. Cg aims to overcome the API differences by building a shading language on top of both OpenGL and Direct3D. A program written in Cg can be compiled to target a specific platform, such as the combination of a particular GPU with a particular API. It then produces code that will work correctly on that platform. Cg was particularly useful when it was the only other option to assembly language shader programs. Although this aspect of Cg's utility has been overshadowed by the appearance of GLSL and HLSL, it is still highly successful for cross-API shader programming, and keeps up with the latest GPU developments through support from NVIDIA. It should also be noted that the API for the Playstation 3's graphics chip (the *RSX*, produced by NVIDIA) is PSGL, a conglomeration of extended OpenGL-ES 1.0 and Cg for shader functionality.

In more recent GPUs, the hardware implementation of the fixed function pipeline has been replaced completely by the programmable pipeline. Any fixed function calls made by an application will be emulated in shader hardware by the programmable processors.

The next major release of Direct3D, version 10, will also include an extra programmable unit called the *geometry shader*. This processor will be placed between

the vertex processor and fragment processor, and will run a shader program on a per-primitive basis. Unlike the vertex processor it will allow the creation of new vertices and have access to both primitive type and vertex adjacency information. Additionally, it will be possible to recirculate the newly created primitives to the beginning of the pipeline so that they can be operated on by the vertex processor. This will allow a new class of GPU-based algorithm to be implemented, such as procedurally created geometry which can be transformed by the vertex shader and lit - all while never having to pass across the bus to the CPU. When the underlying hardware implementation of this new concept is produced, it will also be exposed via the OpenGL extension mechanism.

2.1.4 Exploiting Graphics Hardware

The impressive parallel processing power offered by recent GPUs, combined with this general level of programmability, has resulted in much research into using the GPU not just as a 3D accelerator but as a general processor for suitably parallelisable algorithms. Termed GPGPU [47] by Mark Harris [55], this model expands upon the stream-based nature of the GPUs to replace the 3D processing kernel usually implemented in the programmable processors with a kernel for general computation, specific to the problem domain to which it is applied.

These kernels use texture maps as gather memory and the frame buffer as scatter memory. By employing render-to-texture methods, this memory can be used as a feedback input to the same or different kernels. GPGPU applications typically perform much of their computation in the fragment processor, as this is where the hardware affords the most parallelism - the computational kernel will be executed for every pixel in the output frame buffer. These output values might be the desired final result, or they may be stored and reused for additional computation.

Therefore the ideal application areas for GPGPU are those that have high computational costs with large datasets, exhibit high data-parallelism, and have low dependency on those parts of the dataset not being processed. Many data-parallel domains have benefitted from this work; research has been published in such varied areas as sorting [46], collision detection [45], bioinformatics [15] and computational geometry [11]. Additionally, the middleware physics company Havok [57] have re-

cently announced the implementation of a physics system implemented entirely on the GPU, named *Havok FX*. This framework allows for thousands of particles and rigid bodies to be simulated in real-time for special effects purposes.

2.1.5 Bottlenecks

The pipelined nature of graphics hardware has the inherent implication that the entire system can only proceed at the speed of its slowest section. The vertex transformation stage of a particular application may be very fast, but if an extremely long shader program is causing the fragment processor to struggle, any speed-ups gained in the vertex stage is lost. In optimising GPU-accelerated applications, the biggest gains will be made by identifying and increasing the speed of the slowest stage. Any other optimisations will not move the bottleneck and consequently will not make a difference to the application's performance.

Pipeline Bottlenecks

The most common bottleneck in the graphics pipeline is either the vertex stage or the fragment stage; the primitive assembly and rasterisation stages are rarely the cause of a drop in frame rates. The nature of a pipeline bottleneck is entirely application-dependent. In the case of a simple model viewer that applies a single texture to a very highly tessellated model, the pipeline will be slowest in the vertex transformation stage. Similarly, a model viewer that views only low-polygon models but applies many complex per-pixel effects to the model's surface will be bound by the speed of the fragment processor.

Some applications are not limited by any pipeline stage; instead it is the submission of data in the first place that cannot keep up with the GPU's speed.

Transfer Bottlenecks

The complexity of the operations performed by current graphics cards necessitates the transfer of a large amount of data to the GPU for both geometry and textures.

Up until recently, one major drawback to using the GPU as a general purpose co-processor was its inability to read back the computed results at speeds sufficient to keep up with the rate of processing. This was due to the prevalent graphics card

bus being the Accelerated Graphics Port (AGP). AGP is a dedicated bus based upon the PCI specifications and specifically designed for interfacing the graphics card with the rest of the system. It does so by interfacing with the motherboard's north bridge, thereby having a dedicated link to both system memory and the CPU. This was a major improvement over PCI, the previous bus used by graphics cards. PCI cards are connected to the rest of the system through the south bridge, and as such have to share the PCI bus with every other PCI card such as network cards and sound cards, as well as other I/O devices such as USB and hard drives. Any one of these devices could easily overwhelm the PCI peak transfer rate of 133MB/s, leaving little for the graphics card to use.

At its highest speed (termed AGP 8x), the AGP bus can achieve peak transfer rates of 2GB/s from system memory to video memory. However, due to the design of AGP as a dedicated bus for writing data to the GPU, reading data back occurs at a much lower transfer rate. This rate was improved upon by later graphics cards, but it was still a limiting factor for GPGPU.

However, recently a new bus has been developed named PCI Express (PCI-E [106] - not to be mistaken with PCI-X [107], a variation on the original PCI specification). AGP is being phased out in favor of this new bus, and all new graphics cards are being developed with PCI-E exclusively. The current standard version for graphics cards, PCI-E 16x, has a peak transfer rate of 4GB/s - twice that of the highest AGP version. However, more importantly PCI-E allows for the bandwidth to be split between reads and writes at the same time; AGP could either read or write, but not both simultaneously and switching between the two was non-trivial. PCI-E also allows for more than one graphics card to be present at the same time, allowing the possibility of linking two GPUs together to double the processing power. Both ATI and NVIDIA are already producing cards with this capability, named *Crossfire* [3] and *SLI* [99] respectively.

This new bus removes any potential data transfer bottlenecks for the foreseeable future, allowing ever larger amounts of processing to occur on the GPU, general purpose or otherwise.

2.2 Commodity Clusters

A commodity cluster is a group of off-the-shelf computers that are networked together in order to distribute and therefore lower the computational expense of a suitably partitioned problem. They are usually connected via a high-speed local area network such as fast ethernet (100Mb/s) or gigabit ethernet (1Gb/s). The speed of the connecting network is an important factor, as the ability to transfer data to be processed in a timely manner is usually the limiting factor of the overall computational ability of the cluster as a whole. Clusters are seen as a cost-effective alternative to single monolithic machines of comparable power, although they introduce extra difficulties such as load balancing, data coherency and concurrent shared resource access which must be addressed by developers wishing to use the cluster.

2.2.1 Using commodity parts

Using commodity parts for a cluster has many advantages over using a single cluster solution such as the SGI Prism [121]. Costs are kept down by using mass-produced components. Powerful graphics cards capable of rendering large amounts of data take care of the actual rendering work on each node. A heterogeneous cluster is more robust, as any faulty part can be quickly replaced with an approximately equivalent part. Upgrading can be accomplished easily and incrementally be replacing individual components. Finally, the competitiveness of the commodity component market ensures regular increases in component performance compared to cost.

2.2.2 Parallel Rendering on Clusters

Parallelisation per se comes in two broad categories; functional parallelism and data parallelism. Functional parallelism is based on the idea of decomposing the problem into discrete functional blocks. Each parallel process then performs one of these blocks in a pipelined fashion - an example of this is the graphics card pipeline seen in the previous section. On the other hand, data parallelism is the partitioning of data for identical parallel processing on separate processors.

Data parallelism is often the preferred option for parallel rendering on clusters, as it generally requires less communication overhead to implement. Molnar et al. [92]

describe several taxonomies for data parallelism in rendering; sort-first, sort-middle and sort-last. These approaches are classed according to where in the graphics pipeline (see Figure 2.1) the distribution of rendering work occurs. In the following descriptions, *transformation* is considered to be composed of both the vertex processing and primitive assembly stages, and *rasterisation* represents the other stages of the pipeline - rasterisation, fragment processing and pixel testing.

Sort-first distributes the work before any geometry is transformed or rasterised. Each processor is entirely responsible for a section of the final frame and the processing of all geometry that falls within that section. Therefore before the distribution can occur, the 'pre-transformation' of geometry is required to decide which processor to assign each piece of data to. This is usually done coarsely by a simple method such as the bounding box of the object to which the geometry belongs. Each processor then carries out the entire transformation and rasterisation process for all its geometry and displays the result, typically as part of a large tiled display. Load balancing can be a problem in sort-first parallelisation - if all scene geometry ends up in one section of the screen, that processor must handle all the rendering while leaving the other processors idle. The extra work of pre-transformation also adds to the overall processing costs of sort-first.

Sort-middle occurs between transformation and rasterisation. All scene geometry is arbitrarily assigned to a processor, where it is fully transformed. The resulting screen-space primitives are then reassigned to rasterisation processors, which again are completely responsible for a section of the final frame. In this respect, load balancing can be a problem for rasterisation, although the geometry can be spread evenly across all processors for the transformation stage.

Sort-last completely transforms and rasterises every polygon on an arbitrary processor, only distributing the final fragments for compositing. This method is the easiest to distribute evenly, although the resulting amount of pixel data which needs to be transferred for each frame can be prohibitive. Additionally, extra processing needs to be performed in order to composite all fragments properly in order of depth for the final frame.

2.2.3 Related Work

Perhaps the most researched architecture for rendering on commodity clusters is Chromium, developed by Humphreys et al. [60]. Based on the earlier work of WireGL [59], Chromium is an extensible architecture for interactive rendering on workstation clusters, supporting both sort-first and sort-last techniques. On running an interactive application on a workstation, Chromium replaces the existing graphics card 3D driver with its own driver, intercepting all OpenGL API commands. It then distributes these rendering calls to rendering nodes on the cluster, where the calls are decoded and dispatched to the normal graphics drivers. In this way applications can be run unaware of Chromium, but all rendering will instead occur on separate nodes.

A by-product of this architecture is that the intercepted rendering calls can first be manipulated by *Stream Processing Units* or SPUs. These SPUs can affect any or all rendering calls without needing to alter the application itself. For example, by intercepting all glPolygonMode calls, an SPU can force any application to render in wireframe regardless of the application's original programming. Niederauer et al. [96] used this functionality to partition and visualise the architecture of a game level without modification.

Related custom architectures such as SGI's VizServer [122] perform similar transparent API interception in order to render on a remote dedicated server, returning the rendered image for display on the client machine. Lightning-2 [125] is a dedicated image compositing hardware system, aimed at accelerating the final stage of sort-last cluster architectures. On a larger scale, Pomegranate [37] aims to replace the cluster completely, instead containing up to 64 complete rendering pipelines and implementing a novel scalable "sort-everywhere" architecture which keeps the load balanced at every stage of the pipeline.

Much other research has been carried out on parallel rendering, such as the work on parallel ray tracing by Green et al. [50] and Menzel et al. [89], or the implementation of a parallel volume renderer by Giertsen et al. [42] among others.

2.3 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) are programmable digital logic chips. A programmable logic device (PLD) is one that can be programmed after manufacturing, in order to perform a specific task in hardware much like an application-specific integrated circuit (ASIC). However, the difference is that once an ASIC is manufactured, its functionality is set and cannot be altered further. On the other hand, an FPGA can be updated after manufacturing, having its functionality updated or completely replaced - hence the term 'Field Programmable'.

2.3.1 Background

An FPGA is based on the idea of a 'logic cell'. Logic cells are the basic component of an FPGA, and are composed of a memory element, a lookup table and some logic gates. Each of these logic gates can be reconfigured to duplicate the functionality of either simple logic gates (AND, OR, XOR, etc.) or more complex functionality such as a memory block or a mathematical function. Individually these cells are not able to perform much computation, but an FPGA can contain hundreds of thousands of logic cells, and each cell can be connected to other cells through interconnect wiring. With the right configuration, an FPGA can be made to perform thousands of parallel calculations at every clock cycle. Some more modern FPGAs have the additional ability of partial reconfiguration, where one part of the FPGA can be configured while another part is still running. This has had a great impact in the area of reconfigurable computing.

FPGAs are popular for use in embedded systems, where their design and large number of gates allow for their use as a 'system-on-a-chip' [38]. Their reconfigurability and quick turnaround time from design to implementation mean that they are popular for use in prototyping ASICs - small changes can be made without having to remanufacture a static design. FPGAs are used widely in areas such as Digital Signal Processing [27], telecommunications [39], military and aerospace hardware [18], computer vision [24], encryption [129] and many others. While originally intended as a simple chip for implementing system board component interface logic, they became more popular for implementing full systems as they grew in size, complexity and speed.

2.3.2 Advantages

There are a number of advantages for using FPGAs in application-specific areas. These advantages center around the chip's parallel nature and low design and implementation costs.

The FPGA's ability to be reconfigured quickly and easily is its biggest advantage. When designing an ASIC, a costly procedure of design, development and manufacturing must take place. This reduces the time between incremental versions of the hardware and therefore reduces time to market. Using an FPGA for prototyping and testing greatly reduces this lead time, resulting in faster chip production and greater profits.

The FPGA's reconfigurable nature allows a single chip to be used for widely varying applications, while a corresponding ASIC can only be used for its intended purpose. FPGAs are also preferable to designing a system board to perform the same task, as all operations happen inside the actual chip, meaning faster communication and processing.

FPGAs are capable of a large amount of simultaneous parallel calculations. By building a functional unit that performs some specific computation out of a number of logic gates, that unit can then be replicated across the chip and each one can perform the same calculation in parallel.

In lower volumes, the production of FPGAs is more cost effective than ASICs, which need the non-recurring engineering cost of setting up a manufacturing plant to produce the ASIC. Additionally, bugs or updates to the chip design can be issued after the FPGA is deployed, which is something that is simply not possible with ASICs.

2.3.3 Disadvantages

While more flexible, overall FPGAs are slower than ASICs, capable of less complex designs, and consume more power. Even taking into account the added cost of design, development and manufacturing, ASICs are still the preferred choice for large-scale production of custom chips due to their lower per-unit cost.

From the point of view of a software developer looking to parallelise an algorithm in hardware, FPGAs are far from an ideal choice. The approach needed to design

an FPGA is considerably different to that of software design. The most common language used for programming FPGAs is VHDL - VHSIC (Very High Speed Integrated Circuit) Hardware Description Language, and other popular choices include HandelC and SystemC. However, the use of these latter languages requires understanding various concepts such as clock signals and channels, and their implications on the design and efficiency of the chip. Any hardware description language can make it easy for the uninitiated to produce a design that is grossly inefficient, or simply cannot be implemented on the target hardware. This is especially true for HandelC and SystemC, as their syntactic similarities to ANSI-C can lead to the use of programming methods that are entirely inappropriate for hardware design.

2.3.4 FPGAs and Graphics

Research into the use of FPGAs for accelerating graphics applications has intensified recently due to the advances being made in the hardware's speed and capabilities. This research is largely based around the area of rendering due to the inherent parallelism as discussed in Section 2.1.2.

Woop et al. [128] have introduced an FPGA-based implementation of a fully programmable ray tracing hardware architecture. With an FPGA prototype running at 66Mhz they demonstrate results comparable to a software ray tracer implemented on a 2.6GHz Pentium 4, despite the comparatively small amount of memory bandwidth available to the FPGA. They also demonstrate the scalability of their design to multiple FPGAs working in parallel. Given the speed and power advantages a full ASIC implementation would have over the current FPGA one, they envisage the future widespread availability of ray tracing GPUs similar to today's rasterising GPUs.

Other rendering methods have also been explored. Beeckler et al. [9] have demonstrated a particle system (including the application of a set of forces to each particle) running on an FPGA that is capable of simulating over 2 million particles per frame. Herout et al. [58] have implemented a 3D point cloud rendering system capable of rendering 5 million points per second. Meißner et al. [88] have produced a PCI FPGA-based card that performs shaded and classified volume rendering of large datasets in real-time. Stewart et al. [124] implement a view-independent rendering

system on an FPGA that generates all possible views of a scene and contains a hardware 4D frame buffer.

In other areas, Atay et al. [2] have presented a collision detection chip implemented with an FPGA. They claim speed-ups of up to 36 times that of a 3GHz Pentium 4 for general non-convex rigid bodies. Similarly, Raabe et al. [111, 112] describe an FPGA-optimised collision detection architecture that performs with fixed-point arithmetic. Their results compare well to CPU-based software implementations, and they report speedups of 30 times over a 1.8Ghz Pentium 4 [73].

2.4 The Cell Broadband Engine

The newest parallel architecture developed is the Cell Broadband Engine (CBE). Cell is the result of a collaboration between 3 major media technology companies; Sony, Toshiba and IBM (collectively referred to as STI). Talks of joining together to create a new processor design began in 2000, with the STI Design center formally opening in 2001 at a joint investment of approximately $400m. Each company brought with it a particular special interest - Sony as a content provider, Toshiba as a high-volume manufacturer and IBM as a microprocessor developer. The most high-profile commercial application of the Cell processor is the Playstation 3 games console, due for release at the end of 2006. IBM is already producing Linux-based servers running on Cell, and Toshiba has demonstrated Cell's ability to decode many MPEG-2 streams simultaneously, presumably as a precursor to Cell-powered televisions and multimedia centers.

2.4.1 Design aims

General purpose processor speeds have been improving steadily in recent years, largely due to increases in processor frequencies. However, memory access speeds have not been increasing at the same rate, leading to many applications being limited by memory latency rather than processing speed or bandwidth. This increased memory latency needs to be hidden by the processor with complex chip logic. As a result, more of the chip area has to be devoted to instruction speculation and deeper pipelining, thus reducing available bandwidth and the amount of actual work the

chip is capable of performing. On the other hand, power requirements and heat output are not reduced, so overall power efficiency is reduced. Similarly, deeper pipelines increase the performance penalty of mispredicted branches, leading to diminishing returns as pipeline depth is increased.

The CBE design aims to alleviate these problems by increasing power efficiency and reducing both memory latency and pipeline depths.

2.4.2 Architecture

A single Cell chip consists of nine processors - one main processor called the PowerPC Processor Element (PPE) and eight co-processors called Synergistic Processor Elements (SPEs). All nine processors are connected via the Element Interconnect Bus (EIB), a high-bandwidth memory-coherent bus which is used by the processors to communicate with each other, external memory and I/O devices (see Figure 2.2). It should be noted that a Cell does not necessarily have the complete set of eight functioning SPUs - for manufacturing the Playstation 3, two SPUs have actually been disabled in order to increase the yield.

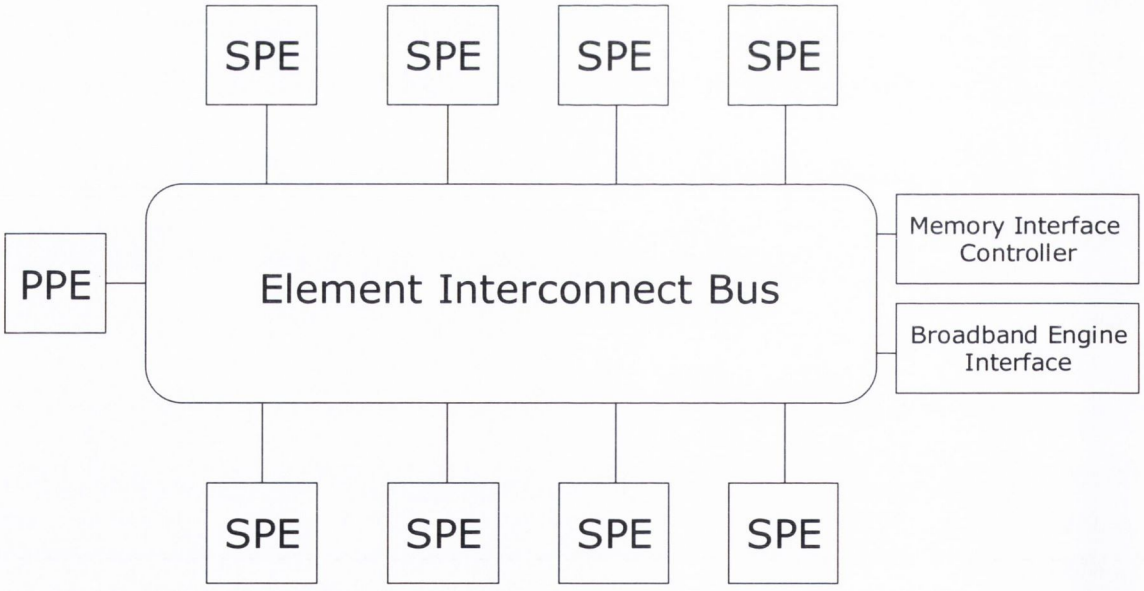


Figure 2.2: Cell Processor Overview

The PowerPC Processor Element

The PPE is the main processor that controls the CBE. It consists of a dual-threaded SIMD 64-bit RISC PowerPC processor and a storage subsystem that governs memory requests from the PPE and external requests to the PPE from other processors (see Figure 2.3). The PPE is a general-purpose processor optimised for running control-intensive software such as an operating system, coordinating all processes running on Cell. The processor itself contains a 32KB level 1 instruction cache, and a 32KB level 1 data cache. It also contains a VMX (Altivec) unit for SIMD computations [28]. The storage subsystem includes a 512KB level 2 unified data and instruction cache.

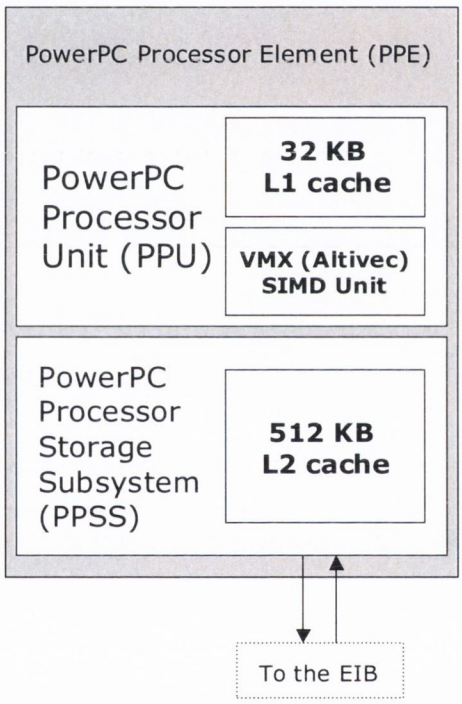


Figure 2.3: The PowerPC Processor Element (PPE)

The Synergistic Processor Elements

The SPEs are where the bulk of Cell's computational work is executed. Each SPE consists of a specialised 128-bit SIMD RISC processor (the Synergistic Processor Unit or SPU) and a Memory Flow Controller (MFC) (see Figure 2.4). The SPUs

are optimised to run compute-intensive code at the expense of branch-prediction and out-of-order-processing hardware, allowing more of the chip to be dedicated to computational work and reducing pipeline depth. Instead of dealing directly with main memory, each SPU contains both a 128-entry register file and 256KB of Local Store SRAM. The SPU uses this to store both data and instructions. Like the PPU, each SPU also contains a VMX vector unit for SIMD operations. These 128-bit SIMD operations can work on a variety of data sizes in parallel; one 128-bit quadword, two 64-bit double words, four 32-bit words, eight 16-bit shorts or sixteen 8-bit chars. However, despite the similarities, the SPU's instruction set is different to that of the PPU, meaning separate compilers must be used for the different processors.

The SPU contains two instruction pipes, and can dispatch two instructions simultaneously to their respective execution units. The first, named the 'even' pipe, issues fixed/floating point and related bitwise operations. The 'odd' pipe covers load/store, branch, and word shuffle instructions. Therefore, maximum SPE execution speeds can be obtained by the careful ordering of instructions to ensure that the pipeline can operate at full dual-issue rates.

Each MFC is responsible for transferring data in and out of the Local Store of its corresponding SPU. It does this through a local DMA controller allowing the SPU, PPU, or another SPU to request a data transfer to or from main memory. In this way the SPE's DMA controller can autonomously transfer data to the Local Store while the SPU is processing other data, thus double buffering and hiding the memory latency behind computation time. Each DMA transfer can be up to 16,384 bytes in size, and an SPU can have up to 16 outstanding DMA requests queued (or 2,048 if using a special DMA-list construct, ideally suited for scatter/gather operations). Theoretical peak bandwidth between the MFC and EIB is 25.6GB/s, with a total EIB peak bandwidth of 204.8GB/s. In practice, approximately 17-20GB/s SPU throughput it typically achievable.

The Element Interconnect Bus

The EIB is a 4-ring structure used for passing data between processors and I/O devices such as main storage. Apart from the PPU and SPUs, the EIB is also

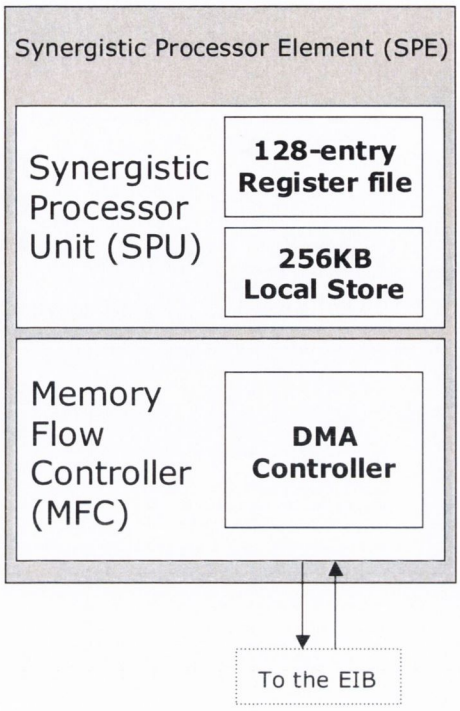


Figure 2.4: The Synergistic Processor Element (SPE)

connected to the Memory Interface Controller (MIC) and the Broadband Engine Interface (BEI).

The MIC supports connections to two Rambus Extreme Data Rate (XDR) memory channels. Compatible devices such as another Cell can be attached through the BEI forming a cluster of Cells - indeed the original patent filing for the Cell showed four cores on a single die. This leads to scalability in two dimensions; the number of processors enabled in any single Cell, and the number of Cells networked together by the BEI.

2.4.3 Programming Cell

IBM released the Cell SDK at the beginning of November 2005 [61]. Included in the SDK is a GNU toolchain which includes everything needed to compile, link and debug a native Cell application. Also included is a full system simulator which replicates the entire functionality of Cell and can be used to emulate a PowerPC-based Linux kernel compatible with Cell. Applications written in C/C++ and

compiled in this simulator using the provided toolchain can then be run on real Cell hardware without alteration.

However, using the GNU compilers require the programmer to have a detailed understanding of Cell's architecture, and to keep in mind every factor that affects performance of the PPE and SPEs. For programmers who have less time to devote to re-implementing existing applications for Cell, IBM are developing the *Octopiler* compiler [36]. While both the GNU compilers produce executables compatible with Cell's different processor instruction sets, Octopiler is capable of compiling and optimising code specifically for execution on the heterogeneous multiprocessor architecture of Cell. Work is partitioned for execution on all nine cores, and communication and memory usage are determined automatically. This is no inconsiderable task, and will be the subject of continuing research on behalf of IBM's compiler designers.

There are three broad programming models for Cell - pipelined, parallel, and service-oriented (see Figure 2.5). The pipelined model has each SPU chained to the next one, using the output of one as the input of another. This allows for high throughput, but is difficult to load-balance. The parallel model runs the same program on each SPU, partitioning and distributing the data to be processed in parallel. The services model is similar to the parallel mode, but each SPU instead processes its data in a different way.

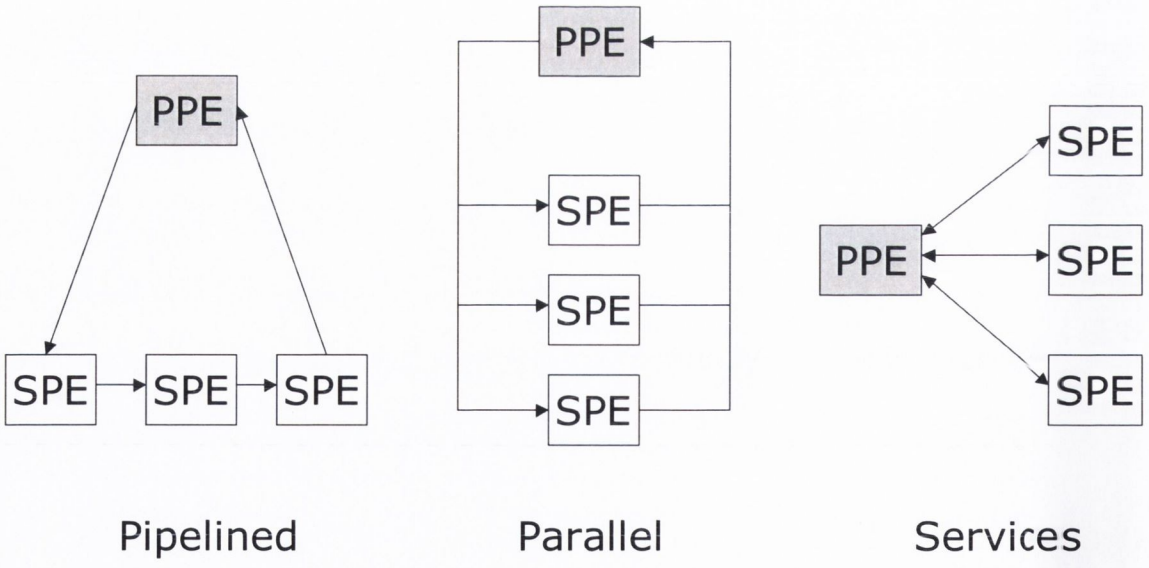


Figure 2.5: Cell programming models

Two sets of program modules are written - one module for the PPU and a separate one for each distinct SPU function (ie., one program for the parallel model but many for the pipelined/services models). All necessary data for the application is loaded and formatted by the PPU, which then distributes the SPU modules to each SPU for execution. The SPUs run, retrieving data from main memory as necessary via DMA requests. The PPU then waits for all SPUs to finish their computation (performing further processing on the results if required) before exiting.

2.4.4 Cell-Related Research

Being a relatively new architecture, the full potential of the Cell processor is still being explored. However, some preliminary work has been published by both IBM and Sony Research and Development.

Masaharu et al. demonstrate a Cell implementation of the Feldkamp algorithm for medical imaging [118]. They compare the performance of the PPE on its own to parallel execution of the PPE and one SPE with SIMD instructions. Compared to the PPE alone, their results indicate an increase in speed of over 20 times for the case of using both PPE and SPE on a 1.8Ghz Cell prototype.

Chow et al. exploit the parallel architecture of Cell to accelerate Fast Fourier Transform (FFT) calculation through the efficient distribution of the computational workload over all available SPEs [17]. On a 3.2Ghz Cell, they demonstrate processing speeds of 46.8 gigaflops per second, over 50 times that of a "leading brand" 2GHz workstation.

Minor et al. also provide an overview of a terrain rendering algorithm on Cell using a ray casting rendering technique [91]. They describe a ray casting technique for height fields and its adaptation for implementation on Cell. An Apple G5 system is used as a client, which interfaces with a Cell-based rendering server over gigabit Ethernet. They claim performance improvements of over one order of magnitude, compared to a single-threaded processor at similar clock speeds.

At a recent Game Developers Conference, Reynolds described the simulation of crowds on the Cell-powered Playstation 3 [115]. He ignores the steering behaviour of the individual units and instead concentrates on their interaction with each other. By using a bucket-based space subdivision algorithm, he parallelises the execution of

update computations and achieves the simulation and rendering of crowds of 10,000 simple polygonal entities (in this case fish) at 60 frames per second. He expects this number to increase as more effective methods are explored.

In a paper published at the same conference, Kokkevis et al. talked about physics simulation on Cell [75]. They gave an overview of Cell-oriented optimisations for various problem domains such as cloth, rigid bodies, fluid dynamics and particle simulations. In each case they gave an example of how best to partition the problem so that individual independent subsystems can process data in parallel, using the Cell architecture to its fullest.

These papers all indicate substantial speedups compared to conventional processors due to the high bandwidth and parallel nature of the system. Many more papers on exploiting Cell for various compute- and bandwidth-intensive problem domains can be expected to appear as availability and popularity of the architecture increases.

Chapter 3

Towards a New Framework

In this chapter we present the design of a new scalable and reconfigurable graphics cluster, built in association with Michael Manzke and Trinity College's Computer Architecture Group. The cluster incorporates many of the advantages of both custom-built hardware and commodity rendering clusters while keeping communication costs down and maintaining a high level of programmability. Additionally, the amount of custom-built hardware is limited in order to keep the resulting manufacturing costs to a minimum.

The design of this cluster encompasses the parallel systems described in the previous chapter, particularly FPGAs and commodity graphics hardware. In addition, many similarities can be drawn between the architecture of the cluster and the Cell processor. The Cell processor contains shared memory that is used by all subprocessors as well as local memory for faster access, just as the cluster does. Where the Cell uses the EIB and DMA engines for data transfer, the cluster uses SCI and distributed shared memory. The SPU co-processors perform the same function as the cluster's FPGAs, albeit with substantially different programming paradigms.

These similarities are not intentional (indeed, details of the Cell architecture were not available when the cluster was being designed), but they do indicate a solid foundation upon which to build. Such a system has many applications in the areas of parallel processing and rendering. As well as being a suitable substitute for expensive all-in-one rendering solutions, it adds the extra advantage of exposing another layer of parallel programmability with FPGAs.

The cluster utilises commodity GPUs to perform all rendering work, and draws upon the research done on graphics hardware and Chromium in order to provide a suitable software framework for rendering. However, as the hardware of the cluster does not contain an operating system that is supported by the GPU vendors, the binary drivers provided by them cannot be used to drive the graphics cards. Therefore new drivers must be implemented that directly interface with the graphics hardware. In order to do this, knowledge of both the underlying hardware and the operating procedure of the driver is needed. Although modern drivers are unified (i.e., a single driver can be installed that will support every generation of GPU released by the vendor), the cluster's drivers must be tailored to the exact GPU chipset that is being employed. This was a major focus of the research done for this project.

The following sections detail the inner workings of each component of the cluster. Section 3.1 gives an overview of the entire system architecture, and describes how the components interact with each other. Section 3.2 details the workings of existing graphics drivers, and describes how this is adapted to supply the cluster's commodity graphics hardware with suitably formatted rendering commands. Section 3.3 details the software infrastructure necessary to run the cluster, Section 3.4 draws comparisons between the cluster and the Cell processor, and finally Section 3.5 reviews some potential applications that could make full use of all the computational resources exposed by such a framework.

3.1 Proposed Cluster

Current solutions for large-scale parallel rendering architectures are generally either custom-built hardware with set specifications and shipped in a singular package, or they are a homogeneous or heterogeneous collection of standard workstations, connected by an ethernet connection as described in Section 2.2.

Both architectures have their respective strengths and weaknesses, and are suitable for different tasks. Packaged architectures such as the SGI Prism [121] are high-quality products that have been thoroughly tested and developed, and tuned for high performance as large-scale visualisation solutions. Using proprietary parts and interfaces can lead to important internal optimizations that enhance overall performance. However, this performance comes at a high price which makes such

products only viable for purchase by large organisations. While scalability of such solutions is possible, it requires even more complex hardware which comes at an even higher price. Another serious limitation is the restriction of upgrade paths; the system is designed and balanced in such a way that upgrading the processing or rendering power requires replacing large parts of the system, leading to further costs. Similarly, the specialisation of the hardware used in these solutions requires a qualified technician for support, and simple problems are not necessarily quickly reparable.

At the other end of the scale, commodity clusters are easily supported by anyone with experience in maintaining commodity computer hardware. While the processing performance attained might not reach that of proprietary solutions due to imbalances in the overall cluster topology, this is compensated for by the flexibility of the system. Parts can be easily replaced at minimal cost, and scalability is usually a case of finding another workstation and plugging it in to the network. Again, upgradability can be performed incrementally and with the minimum of effort, and can be done so as to address any performance bottlenecks that might be limiting the overall cluster's performance.

3.1.1 Cluster Overview

Our cluster aims to draw upon the strengths of both solutions, containing both commodity parts and custom hardware. It has the high bandwidth and processing power of proprietary solutions, which still maintaining the scalability and low cost of commodity clusters.

Broadly speaking, it consists of a tightly coupled cluster of custom-built boards that provide an AGP port for attaching commodity graphics cards. These boards are connected by a high-bandwidth, low-latency Scalable Coherent Interface (SCI) interconnect which implements hardware Distributed Shared Memory (DSM). They are also equipped with two Xilinx FPGAs and a north bridge connecting the chips and SCI to a bank of memory. One of these FPGAs (the "Control FPGA") is used for system control, to govern the SCI services and provide logic to control the board's functions, as well as controlling access to the graphics card. The other (the "Bridge FPGA") is available to be used as a co-processor to augment any algo-

rithms that might be implemented on the cluster, and also manages the distributed shared memory (described below). See Figure 3.1 for an example of the significant components on a single custom board.

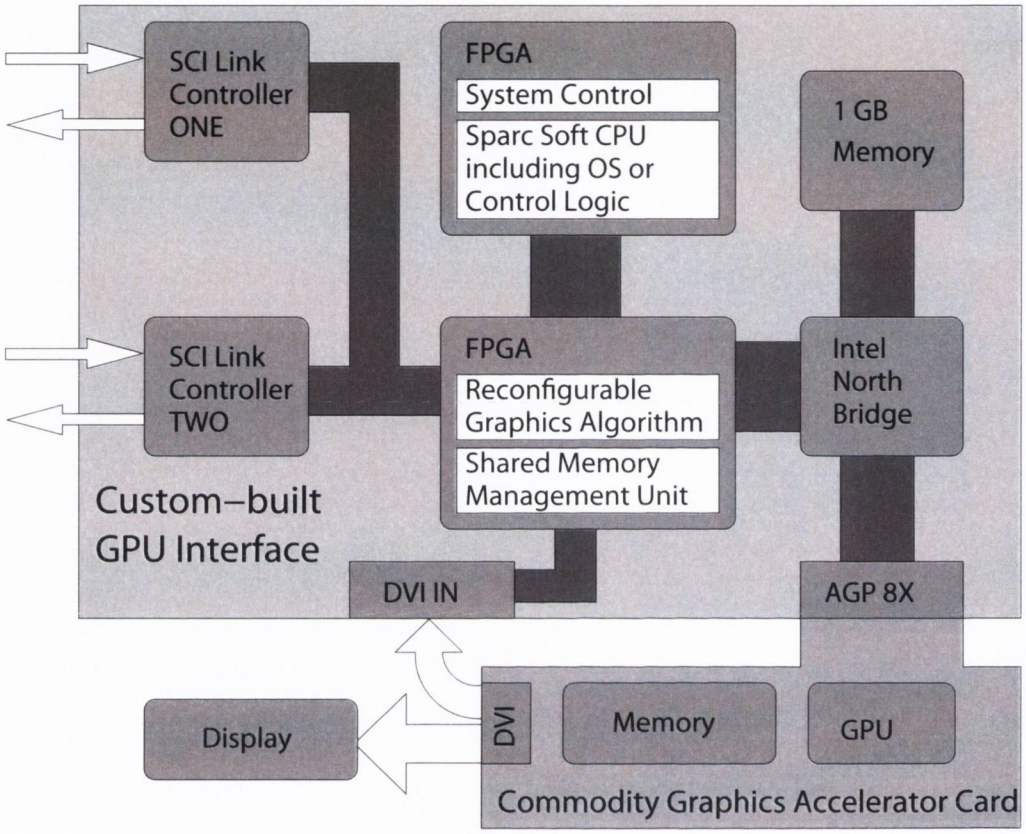


Figure 3.1: A custom board with graphics card and SCI Link Controllers

As well as these boards, the system also utilises commodity SCI subsystems in the form of PCI cards that interface with workstation PCs through a PCI bridge mounted on the card. These workstations are where the actual applications are executed. See Figure 3.2 for an example of an SMP workstation with SCI card attached. The number of workstations can be scaled independently of the number of custom boards being used in the cluster - for example an application may be executed on a single workstation and then have work distributed to many boards for processing and rendering.

In this way, the entire cluster forms a four-level parallel system:

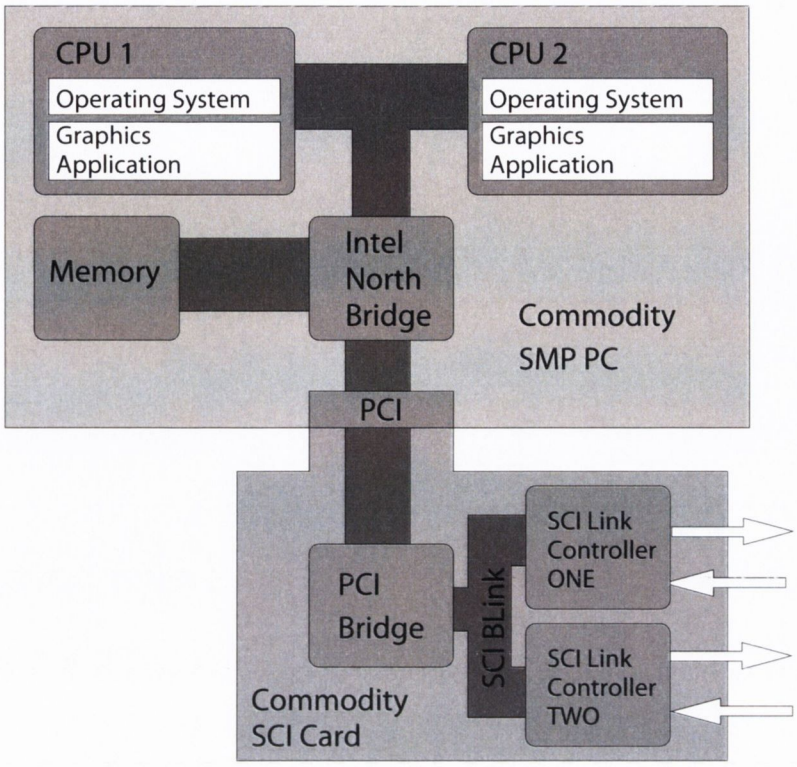


Figure 3.2: A cluster node with attached SCI PCI card

- The connection of workstation PCs through the SCI cards forms a traditional cluster that can be used to work on problems in parallel.
- The connection of the custom boards through the SCI interconnect enables parallel co-processor computation using the on-board FPGAs.
- These FPGAs are themselves inherently parallel, being able to perform a large number of calculations simultaneously.
- The GPUs connected to each board are internally parallel and can be used to augment the computational power of the custom boards by using standard GPGPU techniques, as well as being used for normal rendering work.

A primary advantage of the system is being tightly coupled while still having discrete components. This allows individual parts to be upgraded - most importantly the graphics cards, but also the board memory, the SCI cards and the PC

workstations. Similarly, the reconfigurable nature of the FPGAs allows architectural changes as well as the testing of new computationally intensive algorithms without needing to remanufacture and replace the whole board. It should be noted that while current prototypes use AGP ports for interfacing with the graphics cards, future boards will incorporate PCI Express as standard. Additionally, the PC cluster nodes can also be upgraded to use PCI Express SCI cards, as available commercially from Dolphin ICS Inc. This will considerably improve transfer rates over the PC cluster.

See Figure 3.3 for an illustration of the entire system. Altogether, it incorporates the general computational abilities of the workstation CPUs, the large parallel processing capacity of the FPGAs, and the high throughput stream processing capabilities of the GPUs. This flexibility and large amount of parallel and heterogeneous processing capabilities gives the cluster its edge when compared to traditional parallel architectures.

3.1.2 Scalable Coherent Interface

An integral part of the cluster design is the SCI interconnect which links all boards and PC nodes together. SCI is a high-speed computer bus which is chiefly used in the high performance computing (HPC) sector. Defined in 1992 as IEEE standard 1596-1992 [1], SCI is a system area point-to-point interconnect that has both low latency and high bandwidth, making it very suitable for the implementation of a high performance cluster such as this. In addition, a central part of the standard is its scalability - it supports up to 64,000 nodes on an interconnect. The application of SCI has also been proven in real world critical systems, including those aboard the Mirage F1 and Joint Strike Fighter military jets, the Charles de Gaulle aircraft carrier, and the International Space Station Training Simulator.

Every cluster node (both PC nodes and custom boards) contains two unidirectional SCI Link Controllers (LC). Each LC has an input and output port, and the output of every LC is connected via a cable to the input of the next LC in the ring. Every PC node is connected in a ring, and similarly every custom board is connected in another ring. These two rings are also connected together, forming a 2D torus ring topology, as seen in Figure 3.3, so that any node in the ring (either PC

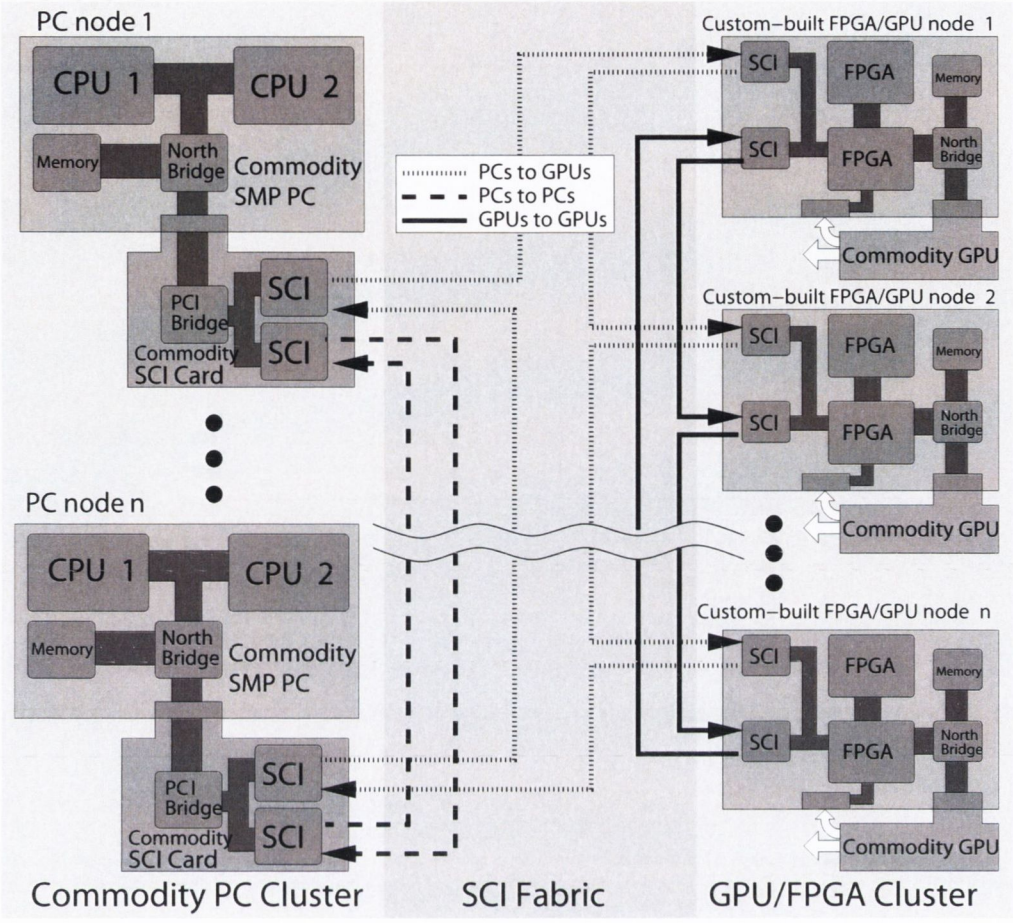


Figure 3.3: An overview of the proposed cluster

or custom board) can talk to any other node. These links are 16-bit parallel connections with a bandwidth of 667MB/s. Packets are routed over the LCs by forwarding them according to local routing tables, which are constructed upon initialisation, thus eliminating the need for an expensive central switch.

Two significant advantages of using SCI as an interconnect are its high bandwidth and low latency due to the fact that inter-node communication can be performed in hardware without having the overhead of software protocol or system calls. Dolphin's current PCI-SCI card offers 326MB/s throughput [32], over twice the speed possible with Gigabit Ethernet solutions. Additionally, they cite 1.4 microsecond application-to-application latency, one of the lowest latencies currently possible according to the HPC Challenge 2005 [23]. This is a marked improvement on other

common cluster interconnects, such as Myrinet's average of 19μs, Gigabit Ethernet's 42.23μs, or Fast Ethernet's 603.15μs. In the case of the custom boards without the overhead of having to go through the PCI bus, internal FPGA-SCI latency would be reduced to the order of nanoseconds.

The SCI standard also offers the option of cache coherency. In this context, cache coherency refers to a local copy of data from a remote piece of memory being up-to-date from the point of view of the local cache. In a distributed system with cache coherency, cached data must be updated to reflect any changes that occur in the original data. Maintaining this consistency adds a performance overhead to the overall memory system, and was not included as part of the cluster.

3.1.3 Distributed Shared Memory

Most importantly, using SCI allows the local memory of each node to be mapped into a shared memory address space. On the PC nodes, this is implemented by the commodity PCI cards using the on-board PCI-SCI bridge which can translate PCI transactions into SCI transactions. Therefore, when a PC makes a memory reference into its own PCI address space, the bridge can translate it into an SCI transaction, transferring it to a remote node. There, the transaction is translated back into a memory access of the remote node's memory, thus implementing distributed shared memory. In this way, distributed Programmed I/O and Direct Memory Access (DMA) can be implemented in hardware without the overhead of system calls, and at very low latencies.

On the custom boards, this shared memory functionality is implemented by the Bridge FPGA. Because this FPGA is connected directly to both local memory and the on-board SCI link controller through the north bridge, it avoids any extra latency or bandwidth restrictions that might be introduced by the PC node's PCI bus operations. See Figure 3.4 for an illustration of the implementation of DSM in the cluster.

The connection of heterogeneous nodes through SCI results in the cluster being a Non-Uniform Memory Access (NUMA) architecture. This means that each node contains its own local memory, and that memory access times differ depending on the location of the memory. Specifically, access by a node to a local memory

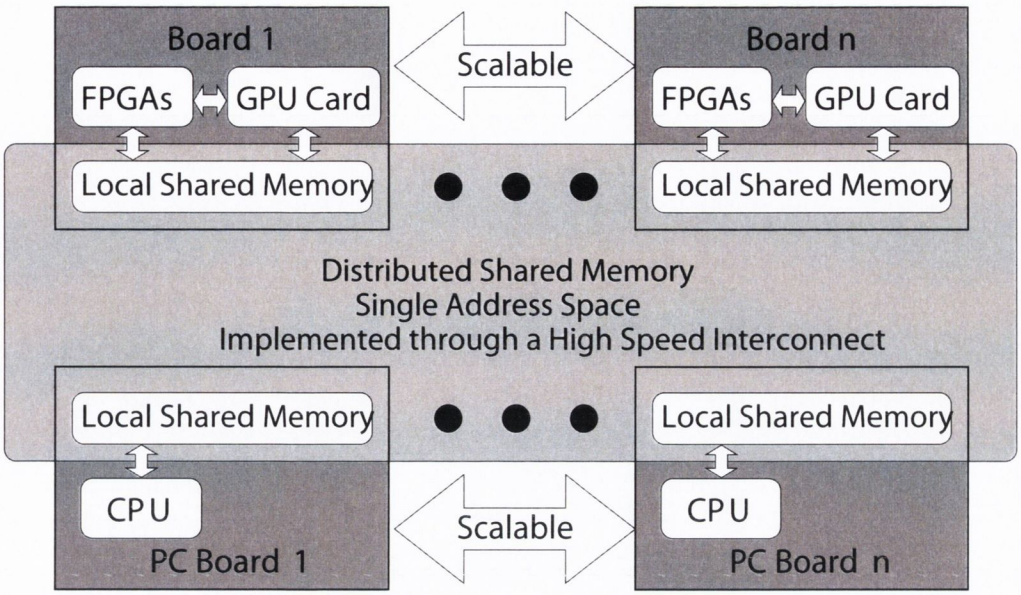


Figure 3.4: Distributed Shared Memory implemented in hardware

location is faster than an access to remote memory. This has consequences for the data access schemes used by the cluster to share data, view remote data, and resolve concurrency problems, all of which must be kept in mind when designing or choosing communication protocols.

3.1.4 Aims

The primary function of the cluster is as a platform for parallel rendering, augmented by the extra computational power of the Bridge FPGAs. The implementation of DSM eliminates the need to replicate data across nodes of the cluster, and the low latency and high bandwidth afforded by SCI allow large amounts of geometry and texture data to be processed and rendered interactively. It also operates as a test bed for the implementation of new parallel hardware algorithms on the Bridge FPGAs. The scalability of such a cluster is addressed by the use of SCI as an interconnect.

It should be noted that the current state of the cluster is a single prototype custom board. It has been manufactured (see photo in Figure 3.5) and is currently being debugged by researchers in the CAG. A second revision of the board is expected soon which will resolve outstanding problems that arose during manufacturing. Once the

board is ready, multiple copies can be produced and implementation of the software infrastructure can proceeed.

The cost of the prototype custom board is currently approximately €3000. However, due to the economics of fabrication, this price can be expected to drop to that of a high-end PC in the case of large scale board manufacturing.

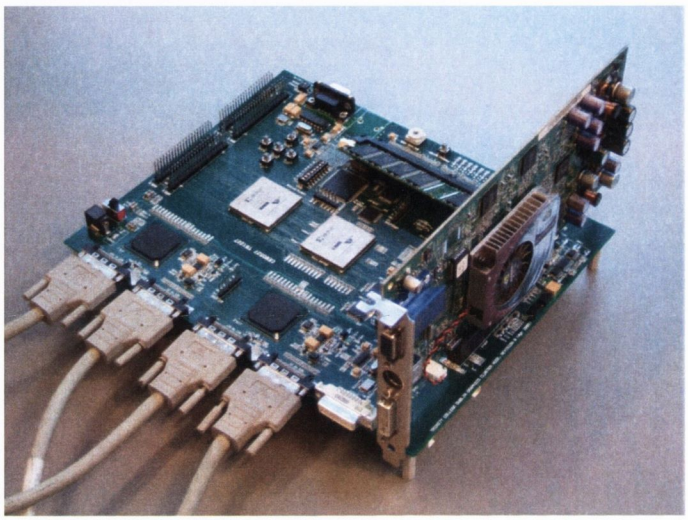


Figure 3.5: The first prototype custom board, with commodity graphics card attached and SCI cables plugged into the Link Controllers.

3.2 Graphics Hardware

Central to the design of the whole cluster are the graphics cards plugged into the custom boards. These cards are used as the rendering engines of any graphics algorithms that are executed on the cluster. This section details the traditional relationship between operating system and graphics hardware, and describes how this mechanism is adapted to work on our custom-built nodes.

3.2.1 Graphics Drivers

Modern operating systems such as Windows, Linux and Mac OS are only able to interface with commodity graphics cards through vendor-released device drivers. These drivers abstract away the proprietary hardware interface of the graphics card

and allow developers to use common APIs such as Direct3D and OpenGL to perform hardware-accelerated 3D operations. Any new functionality added to these APIs (such as a new OpenGL extension) will not be available for use until the video driver exposes this functionality to the application. Similarly, new capabilities of the GPU are not available unless exposed by the driver. If a particular feature is not available in hardware but is required for conformance to API specifications, the video driver may choose to implement it in software instead. In this way, the driver is an extra layer between the hardware and operating system, and as such can incorporate newly-discovered software optimisations (such as vertex submission optimisations or texture packing methods for example) without needing to change either the hardware or the API interface. By regularly releasing updated drivers, the apparent performance of a card can be increased significantly over its lifetime.

The reason for proprietary drivers is that the internal workings of graphics cards are confidential details which GPU vendors do not freely distribute. Through contacts in ATI we were able to obtain the Technical Reference Manual [6] and Register Reference [7] for the R200 series of chips. This chipset forms the basis of GPUs from the ATI Radeon 8500 (the R200) to the Radeon 9200 (the RV280), used in cards released in 2002. We have chosen to use 9200-based cards, being the most advanced chip in this range and because it is one of the only versions to support the highest AGP transfer rate (8x). See Table 3.1 for further specifications. While this is not cutting edge graphics hardware by today's standards, it is recent enough to support a programmable pipeline exposed in the form of the OpenGL extensions ARB_vertex_program and ATI_fragment_shader. Thus it is advanced enough to be used as proof of concept in the cluster prototype, with the hope that more recent technical specifications would be released to us by ATI once the feasibility of the cluster as a whole is demonstrated.

Based upon comparison to a similar technical reference of the earlier Rage 128 chipset [5] also developed by ATI, we are confident that the fundamental ideas behind the hardware interface will remain largely unchanged in the future, and that the methods described here will still be applicable upon migration to a newer generation of ATI GPUs. Therefore, while the rest of this section is based upon information pertaining to the R200 series specifically, it is reasonable to assume that the inner workings of more recent chipsets (and by extension their drivers) are

ATI Radeon 9200 (RV280)	
Bus Type	AGP 8x
Clock Speed	250Mhz
Memory	128MB DDR @ 400MHz
Memory Bandwidth	6.4GB/s
Vertex Pipelines	1
Pixel Pipelines	4
Texture Units	6
Fill Rate	1 Gpixel/sec
Geometry Rate	62.5 Mtriangles/sec

Table 3.1: Specifications of the Radeon 9200 (RV280 chipset) [4].

not significantly different.

Although proprietary graphics drivers are the only way to achieve hardware acceleration of 2D and 3D operations, cards also provide a *VGA mode* for performing simple non-accelerated 2D drawing. This is done via the industry-standard Video Electronics Standards Association (VESA) Video BIOS Extension (VBE) programming interface. Providing support for this interface allows the graphics card to be used as a simple video adapter without the system needing to know any specific details about the underlying hardware, and can therefore use a generic VBE driver.

A certain amount of related work has been done by the open source Direct Rendering Infrastructure (DRI) project [105]. This project aims at providing an open source Linux implementation of drivers for ATI GPUs up to and including the RV280, again based upon technical references provided by ATI to certain members of this project. While a limited amount has been learned from the source code, it is poorly documented and mainly aimed at separating kernel-space functionality from user-space in order to provide a secure method of accessing hardware, as well as concentrating on allowing multiple clients on a single machine to have simultaneous access to the graphics hardware. However, neither concurrent access nor kernel integrity are a concern with respect to the cluster, and access to the original documentation from ATI precludes the need to rely on information gleaned from this project.

3.2.2 Hardware Registers

A hardware register is a limited storage area (typically the most efficient size for the processor to interface with - in this case 32 bits) located on a peripheral that allows high-speed I/O, control and configuration. Rather than being accessed directly by the CPU, these registers are typically mapped into the memory space of the host system, and accessed via reads and writes to memory. This is referred to as Memory Mapped I/O (MMIO).

Upon initialisation of a PCI or AGP peripheral, either the system BIOS or the operating system assigns an area in the system's memory address space (which is a total of 4GB for 32-bit architectures) for each of the peripheral's I/O regions, ensuring that the allocated memory regions do not conflict with the areas of physical memory. This typically happens at boot time, although this is not necessarily the case for hot-pluggable devices which can be inserted and removed while the system is running. The peripheral is unable to be accessed until these memory regions are allocated. In the case of a PCI or AGP video card, there are two regions to be mapped - the I/O registers for MMIO and the card's video memory (also referred to as the frame buffer - not to be confused with the frame buffer of the graphics pipeline that only represents the area of video memory that contains the final image to be drawn on-screen). The addresses of these mapped memory regions, referred to as *Base Address Registers* (BARs), are then assigned to the peripheral via PCI Configuration Space. This is a standardised area of 256 bytes located on the peripheral that describes it with IDs such as Vendor ID and Device ID. In this way, the system can iterate through every PCI device and assign the required BARs for each device without needing to probe the devices, which can cause unwanted side-effects if the wrong address is probed.

Every attribute and function of the RV280 is accessible by reading from or writing to the relevant register. These registers are listed in the Register Reference [7] and describe each register, its purpose and its MMIO offset. For an example see Table 3.2; this particular register describes the type of memory installed on the card. By reading the 32 bits (one word) at offset 0x158 from the MMIO base address in system memory and isolating bit 30, the RAM type can be determined. Similar registers are available for every aspect of the card. Some are just for reading

attributes such as memory type and size, monitor type etc., and others are for writing to and have side effects such as a change in video mode, causing something to be drawn etc.

Similarly, writing values to the correct portion of the mapped frame buffer causes on-screen pixels to change color. We have already implemented and demonstrated the initialisation of a 9200 with a custom driver written from scratch for Linux Fedora Core 4 using an adapted 2.6.11 kernel, based upon the documents supplied by ATI. Writing the appropriate values to the correct registers causes the adapter to change video mode, and then any writes to the frame buffer draw a solid colour of that value directly to the screen. However, while useful for diagnostic testing, direct frame buffer writing does not take advantage of any 2D or 3D hardware acceleration available in the card. Section 3.2.4 below describes how to employ hardware acceleration properly.

MEM_SDRAM_MODE_REG - RW - 32 bits - [MMReg:0x158]			
Field Name	**Bits**	**Default**	**Description**
MEM_CFG_TYPE	30	0x0	0=SDR 1=DDR

Table 3.2: An example of an entry in the R200 Register Reference.

3.2.3 The AGP Aperture

As well as providing a direct point-to-point connection from the graphics card to system memory and the CPU, AGP also allows system memory to be used to augment the video memory local to the card. When video memory is full, this *AGP memory* can be used for caching textures and geometry. Although access to AGP memory is many times slower than video memory, its use is preferable to putting a hard limit on the amount of data that is available to the GPU.

AGP memory requires a significant amount of system memory to be re-mapped. Data that is stored in AGP memory is not swapped into video memory before being used, it is referenced directly. Therefore it is essential that the data be present in a contiguous area of memory, in order to allow the most efficient access possible without costly software reordering. However, this is incompatible with the memory

allocation procedures of operating systems, which keep a pool of free memory pages (each page usually being 4k in size) to allocate dynamically. This can lead to large fragmented regions of memory which are unusable as AGP memory.

As a result, an intermediate step is needed in order to present a single large contiguous area of physical memory to the GPU. An area of contiguous memory is reserved in the system address space and forms the *AGP Aperture*. Each page in this aperture corresponds to a (possibly discontiguous) page of allocated system memory. The mapping from the AGP aperture to system memory is governed by the Graphics Address Remapping Table (GART). When a page inside the aperture is addressed by the graphics card, it is converted to a real system address by looking up the GART and returning the actual physical address represented by that aperture page. This remapping is done by address translation logic in the north bridge.

With respect to the graphics cards attached to the custom boards of the cluster, the AGP aperture can be created inside an area of shared memory so that it can be seen and accessed by any other cluster node. This allows a texture resident in system memory of one node to be directly accessed by the graphics card of another node, eliminating the need to replicate the texture on every node that requires it. This also applies to geometric data. The SCI interconnect ensures that the latency of this shared memory access will remain extremely low, and that the data will be transferred directly to the GPU in a timely manner due to the high bandwidth. By including the system memory of the PC nodes in the shared address space, data can also be accessed from there by the GPUs.

3.2.4 Employing Hardware Acceleration

In order to achieve hardware acceleration of both 2D and 3D draw calls, the RV280 allows two separate methods of instructing the GPU - Programmed Input/Output (PIO) mode and Command Processor (CP) mode. Both methods are used to fill a FIFO command buffer internal to the GPU, the entries of which are processed by the rendering engine to draw into the frame buffer. However, they differ in the processes used to transfer commands to this buffer.

Programmed Input/Output Mode

PIO mode is the more direct and simpler method of the two. In this mode, the driver fills the FIFO command buffer by directly writing to the MMIO registers that control 2D and 3D drawing. This is referred to as the *Push Model* because commands are 'pushed' into the buffer by the driver. Although it would be possible to implement PIO mode access to the cluster's GPUs over DSM, doing so would raise concurrency issues that are not easily resolved if two or more nodes tried to write to the same register simultaneously. This could be overcome by using an intermediate queue, but this problem is already resolved by CP mode as described below. Being less efficient, PIO is most useful as a method of debugging. Additionally, the size of the FIFO command buffer is limited by on-board storage, something that is not a problem for CP mode.

Command Processor Mode

CP mode does not deal directly with the command FIFO, but instead sends commands via *command packets* that are interpreted by an on-board *microengine*. The command packets comprise a 32-bit header which describes the packet type, followed by a payload of data, the size of which is specified in the header. A single packet can represent the same effect as multiple register writes, which simplifies many common drawing operations. Upon activation of the card, the microengine is initialised by loading in 256 quadwords of microcode data which are supplied by ATI. It can then parse the command packets, filling the internal FIFO buffer with commands. As in PIO mode, this buffer is then processed to perform hardware accelerated rendering.

There are two ways to transfer packets in CP mode. The first is similar to PIO mode - the packets are written directly in through the MMIO registers. However, it is much more efficient to queue the command packets in a buffer in system memory, and then initiate a transfer of all packets into the GPU using bus mastering. As opposed to the push model of PIO mode, this mode is referred to as the *Pull Model* because the GPU 'pulls' data in from AGP memory. This allows for CPU/GPU concurrency and lets the CPU continue with other processing while the graphics card transfers the packets for parsing and rendering. Bus mastering in this way is done via two separate but related buffers; the *Ring Buffer* and the *Indirect Buffer*.

These buffers are used to store the command packets that are to be fed into the command FIFO.

The primary buffer used is the ring buffer. This is a contiguous block of memory which is stored in AGP memory, and is seen as a circular buffer by both the GPU and the driver. Every time the driver writes a packet to the end of the ring buffer, it increments a 'write' pointer which is visible to both driver and GPU. Similarly, whenever the GPU reads a packet from the front of the ring buffer, it increments a 'read' pointer to point at the next available packet. When one end of the buffer is reached, it wraps around to the beginning in a circular fashion. Once the write pointer equals the read pointer, the ring buffer is considered to be empty. Initialisation occurs upon start-up, when the driver allocates the buffer and writes the AGP memory location and size of the ring buffer to the relevant GPU registers, along with the address of the read and write pointers.

While using a ring buffer allows much more efficient transferring of data to the GPU, the packets placed into the buffer are consumed and discarded as soon as they are used. This means that every time a particular command packet is needed to perform some function, it must be constructed and placed into the ring buffer. If this packet is used very frequently, the overhead of constructing the packet and placing it in the buffer each time goes some way to negating the advantages of using a ring buffer in the first place. For this reason, the command processor is also capable of reading from the indirect buffer. The indirect buffer is a linear buffer, also a contiguous block of AGP memory, which does not employ any wrapping mechanism. Command packets can be placed into this buffer in an arbitrary order, and the command processor can be instructed to process them by writing the location and size of the packets to the relevant registers. Alternatively, command packets with this information can be inserted into the ring buffer to be processed. The important distinction between the ring buffer and the indirect buffer is that packets in the indirect buffer are not replaced once they are used, allowing them to be used multiple times. For example, a packet pointing to a collection of vertices that describe a model can be inserted into the indirect buffer, and when the model is to be drawn, a packet is inserted into the ring buffer describing the location and size of the packets in the indirect buffer. The command processor then switches to retrieving the packets from the memory pointed to in the indirect buffer packet, processing them and drawing

the model until they have all been accounted for. It then returns to processing the next packet from the ring buffer. This means that no copying needs to be done from system memory to the ring buffer, since the packets describing the location and properties of the model will not be overwritten and do not need to be refreshed.

Distributing the Command Processor

The fact that these buffers are located in AGP memory instead of video memory is an important one. It means that we can choose to allocate a node's buffers in the global shared memory address space, and place packets into them from any node. One node can decide to make a remote node draw a model by placing the relevant packets into its indirect buffer, and then placing a packet into the remote ring buffer describing where to find the model. Alternatively, all nodes could be made to share one big indirect buffer located in the shared memory address space, the physical location of which is spread over all nodes equally. Thus the data (whether it is geometric or texture data) only needs to be stored once and pointed to once in the cluster, and no replication is necessary.

However, for this scenario to be feasible, an arbitration method of accessing a node's ring buffer is necessary in order to avoid the situation of two different nodes trying to write to the same location, or updating the write pointer simultaneously. Similarly, two nodes could attempt to place packets in the same location of the indirect buffer, one overwriting the other. This can be avoided with the use of mutual exclusion algorithms from the area of concurrent programming. However, as mentioned in Section 3.1.3, the cluster's NUMA architecture means that the method of arbitration must be carefully chosen in order not to generate more interconnect traffic than necessary. The use of semaphores to synchronize access could potentially solve the problem, with nodes continually checking to see if the semaphore is free to write to the buffer. Ring buffer and indirect buffer writes will generally be very short, so no single node would be made to wait an unacceptable amount of time before. However, the traffic generated by nodes continually checking the semaphore could become unacceptable. This could be solved by using a queue mechanism for allowing nodes to add their packets to the end of a queue local to the ring buffer's node, which is then fed into the ring buffer in a timely fashion.

The most common and efficient method for a GPU to access texture and geometry data is for that data to be located in on-board video memory. It is transferred there by a command packet that instructs the graphics card to fetch the data from AGP memory. By exploiting this, we can have one copy of any texture or geometry data located in shared memory and then instruct any graphics card that requires that data to fetch it remotely for storage in local video memory. This retains the performance gains achieved by having the data in high-speed local memory, while removing the need to also keep a copy in the node's system memory.

3.3 Software Infrastructure

In order to drive the entire cluster, rendering must be initiated by a graphics application running on one or more workstation nodes. To do this, a software infrastructure must be in place that converts regular OpenGL rendering commands into distributed, hardware-specific command packets and routes them to the right node for timely rendering. The details of how this is done depends on the nature of the rendering and the setup of the cluster.

3.3.1 Molnar's Taxonomies Revisited

For the system to fulfill its primary role as a parallel rendering cluster, we must take another look at the taxonomies of Molnar et al. as described in Section 2.2.2 and explore how they can be applied to take maximum advantage of the unique mixture of parallel hardware available to us.

Sort-first: In a traditional parallel cluster rendering system such as Chromium [60], operating in sort-first mode, pre-transformation must be performed by the application in order to determine where the geometry needs to be sent to be transformed and rasterised. While this pre-transformation can be as simple as calculating a screen-space bounding box for each model (at the cost of 6 conditional assignments per vertex of the model, plus matrix transformation of the bounding box into screen-space), it still adds an extra burden on the application node since all pre-transformation must be done before distribution. We can improve upon this by assigning each model to an arbitrary node and

performing the pre-transformation in the Bridge FPGA, either in-situ through shared memory or by actually transferring the model into the remote node's local memory for faster access. The model can then either be redistributed to the proper node's indirect buffer for rendering, or placed in a shared indirect buffer for remote access by the rendering node. This allows for a more even load distribution, and reduces the strain on the application node. Sort-first is primarily used for 'tile-rendering' systems, where each rendering node is attached to a projector which draws one tile of the complete image.

Sort-middle: While sort-middle allows even load-balancing of the transformation stage, the distribution here occurs with the intermediate screen-space primitives. By employing the GPUs to perform the full transformation stage, these screen-space primitives are located inside the graphics hardware pipeline and so are not directly available outside the chip. To implement a sort-middle architecture, we would have to either use the programmable pipeline to provide the screen-space primitives (by using a vertex shader to calculate them and then encoding them in an image output by the fragment shader), or else perform the entire transformation stage in the Bridge FPGAs, ignoring the GPU's vertex processing capabilities completely. Neither of these solutions are acceptable, as they do not make full and proper use of the available parallel hardware that is dedicated to performing primitive transformation, i.e., the GPUs. Therefore, despite being the natural place to distribute the rendering work for optimal load-balancing, a sort-middle architecture is not feasible on the cluster.

Sort-last: Another promising architecture for the cluster would be sort-last. In a similar fashion to the Lighting-2 system [125], the FPGAs can be made to perform efficient depth compositing of the images produced by the GPUs. Instead of a costly read-back over the AGP bus, the DVI inputs attached to each custom board (see Figure 3.1) can be used to read back the rendered image produced at each frame at a latency of exactly one frame. The pixel data can then be transferred across the cluster due to the high bandwidth of the SCI interconnect, the contributions of all nodes being composited together to form the final image. In this way, the system as a whole can be used for all

simulation, rendering, and compositing procedures without needing additional dedicated hardware for any single step.

There are also other factors that must be considered when building a software infrastructure for a parallel rendering cluster. When the application changes the graphics state, for example by disabling texturing or changing the current diffuse colour, this change must be tracked and distributed to each rendering node. Buck et al. [12] describe a system for updating the graphics state over a cluster by using 'lazy updates'. This efficiently calculates the difference between two graphics states and only communicates updated attributes when absolutely necessary in order to keep transmission costs down. A similar mechanism would certainly be applicable to our cluster, although it may not be necessary if all rendering nodes use the same graphics state via a shared indirect buffer. In this case, any state changes that have a global effect could be queued in the shared indirect buffer and referred to by each node's local ring buffer.

3.3.2 Workstation Parallelism

As well as investigating the infrastructure necessary to perform parallel rendering on the cluster, we must also consider the software necessary for performing parallel computation over the attached workstation nodes (if there are more than one). This is an area of active research, and many methods exist to distribute work between computers.

Of course, regular methods of communicating over ethernet via TCP/IP are possible on any machine with a standard operating system and a network adapter. However, by using these we would be bypassing the available SCI interconnect and the significant improvements in bandwidth and latency associated with it. Therefore we concentrate on the available methods of communicating over SCI.

Traditional parallel computing tasks on heterogeneous clusters often use software layers to abstract and standardise communication between workstations. The two most popular standards are Parallel Virtual Machine (PVM) [127] and Message Passing Interface (MPI) [94], with MPI being arguably the more widespread and popular of the two. Additionally, while work has been done on implementing a

PVM-based system over SCI (such as the work by Zoraja et al. [148]), the most recently released SCI communication software packages are MPI-based.

As the name implies, MPI is a method for parallel processes to communicate through the passing of messages. It is an open standard which is easy to use, but also provides significant functionality if required. There are many implementations of the standard, the most common being the open source MPICH [51] which is available for many platforms including Windows and Linux. Additionally, Worringen et al. [144] have implemented MPICH over SCI. Therefore the use of MPI as a protocol for communication between workstation nodes would allow compatibility with the many existing MPI-based parallel programs while also taking advantage of the SCI interconnect.

Another alternative is to use the open source SuperSockets software supplied by Dolphin [30]. This is a layer which allows an application to use regular Berkeley sockets for communication over SCI without SCI-specific code. Again, sockets are in widespread use for network data transmission, meaning this method would provide support for many applications while providing much lower latency and higher bandwidth than regular sockets due to the underlying interconnect. The fact that it is developed by Dolphin also ensures that interoperability with the SCI hardware is as good as possible and that the software is updated regularly.

For applications that are aware of the underlying cluster architecture and the SCI interconnect in particular, Dolphin also supplies a lower-level library called SISCI [31]. This lets the application developer include API calls that will interface with the hardware directly, reducing the amount of system and library calls that introduce overhead and latency in other higher level systems.

3.3.3 Communicating with the Driver

When a regular Windows OpenGL application makes a call to the API, it is the driver's OpenGL Dynamic Link Library (DLL) that takes this API call and produces the command packets as described above. In the case of Linux, it is a Shared Object (SO) that implements the interface. In order to change the behaviour of the interface, this library must be replaced and all exposed API functions that the application uses must be either re-implemented or forwarded to the original

library. The nature of this re-implementation is entirely up to the replacement library, although usually it will perform some sort of non-invasive monitoring of API calls before passing them onto the original driver for regular rendering. Applications such as gDEBugger [48] and GLIntercept [133] take advantage of this in order to intercept OpenGL calls and allow the developer to view statistics and diagnostic output. They also permit influencing the actual OpenGL functionality through methods such as replacing textures and changing state variables etc. This can prove very useful to developers for optimisation and debugging. Chromium also uses this method to intercept API calls for distribution over the cluster, handing the calls to a regular vendor-released driver for rendering at each node.

In a similar way, we can replace the system's OpenGL driver with our own library. This allows a regular application to run normally, even though it is unaware of the underlying architecture, while still taking advantage of the distributed rendering power of the cluster - much like Chromium. However, the processing required by the cluster's replacement library is much more involved than the simple pack-and-forward operations of Chromium's node library. In our case, we need to do the work of both the distribution library and the vendor's driver. This involves deciding which custom board node to distribute the rendering calls to (depending on rendering method, i.e., sort-first etc.), and constructing the corresponding command packets to feed to the destination node's GPU.

It makes sense to condense these two steps into a single library call in order to minimise the latency caused by multiple shared library calls. Therefore, instead of simply packing the OpenGL API calls into a network stream for transmission, the replacement driver will translate these calls into command packets and submit them to the relevant ring buffer or indirect buffer. This eliminates the need for command packet construction on the FPGAs, which is beneficial as details of both the OpenGL API and the logic used to interface with the hardware are subject to change. A software driver is easier to upgrade and encapsulate than hardware changes, meaning the view from outside the driver is essentially remains the same throughout upgrades.

In order to avoid driver bottlenecks due to the extra overhead of command packet construction and distribution, care will have to be taken to ensure that API calls make as efficient use of the hardware as possible. However, this should already

be the case for ordinary performance-oriented 3D applications. Additionally, the available processing power of the FPGAs offloads some of the work from the CPU, giving it more cycles to devote to the driver.

3.3.4 Communicating with the FPGA Co-processors

In the case of applications that have been designed specifically for the architecture of the cluster, the extra processing power available on the custom boards must be exposed to the developer in order to be usable.

The underlying logic for any co-processing functionality will first need to be implemented in the FPGAs, through VHDL or another hardware description language. It would not generally be possible to allow the developer to implement arbitrary logic programmatically, due to the fundamental differences in architecture between a regular 3D graphics application and an algorithmic implementation on FPGA. Therefore, a library of modules will have to be built up as different requirements are realised. This may initially be something as fundamental as hardware accelerated linear algebra calculation, leading to more specialised implementations later on as new and existing applications are implemented on or ported to the cluster.

Communication with the FPGAs is achieved through the SCI fabric and the single shared address space. Data structures specific to the algorithm being processed are written into shared memory, and computation is initiated by writing to special FPGA registers or *command ports* which are also accessible via shared memory. The results of these calculations are then placed back into memory, where they are available to be read back into the application for further processing. In special cases where further computation is unnecessary, these results could be formatted by the FPGA to be directly accessible to the GPU as texture or geometry data, avoiding further latency that would be introduced by passing data back over the workstation's I/O bus.

Although this work would initially be part of the replacement system driver, eventually a new driver and API will be designed to abstract away any hardware access. This would make the FPGA's capabilities available to developers independently of the GPU functionality, leading to the increased modularity and flexibility of the overall system.

3.4 Comparison with the Cell Processor

There are many aspects of the cluster that invite comparisons to the Cell Broadband Engine described in Section 2.4. Both are parallel systems on multiple levels and both have similar methods of transferring, storing and processing data.

Memory access: The memory access methods used by both the cluster and the CBE are very similar. The NUMA architecture implemented by both leads to a two-level concept of fast local memory and slower remote memory accesses. On the CBE, the local memory is restricted to the SPU's 256kb local store, which is much more restrictive than the cluster's local memory. However the design of the SPU's local store as on-chip Static RAM (SRAM) means that it can be accessed and written to faster than the cluster's DRAM. At the same time, remote memory accesses on the CBE must take place through the DMA transfer mechanism, whereas the cluster can access memory more transparently through the single global address space. In both cases, an application's memory access pattern must take into account the cost of retrieving data from both local and remote locations.

Distributed processing: The closest comparisons between the two systems can be drawn from their distributed processing capabilities. Both are capable of executing multiple programs in parallel, distributing them across discrete processors and allowing inter-process communication. Alternatively, both are also capable of performing in a pipelined fashion or using a service-oriented paradigm. The only difference here is that while the cluster can distribute processing symmetrically across all processing nodes, the asymmetrical nature of the CBE is more suited to data-intensive processing on the SPUs with control-intensive processing on the PPU.

Internal parallelism: As well as the explicit parallelism of multiple distributed processors, both systems are also capable of internal parallelism on each processor. The Cell's PPU can execute two hardware threads simultaneously as well as containing an Altivec unit for SIMD code execution, which the SPUs are also capable of. However, each cluster node has the potential for much higher parallelism due to the FPGA implementation. Neither system can be

said to be superior to the other; the advantages of each are data-dependent and the suitability of the algorithm being executed determines how well each design performs.

Interconnect architecture: Each systems requires an interconnect for communication between processors; the Cell's EIB is comparable to the cluster's SCI interconnect. Both are implemented in a ring topology, with the consequence of adjacent nodes communicating faster than physically more distant nodes. The communication systems differ however - the SCI fabric implements the global shared memory address space, performing address translation for the application. Conversely, applications on the CBE must perform and manage the DMA data transfers explicitly.

Scalability: Again, both systems correspond with their potential for scalability. Multiple CBEs can be connected through the Broadband Engine Interface, increasing the processing power available to an application. The cluster also scales by adding more nodes. In the cluster's case, this also implicitly increases the rendering power available to the system as a whole, due to the graphics cards attached to each node.

Programmability: The programmability of each system is where the designs diverge most significantly. Programming the Cell will be immediately more familiar to software developers, even taking into account the considerations that must be given to the CBE's unique architecture. While applications on the PC nodes of the cluster can execute conventional programs written in languages such as C/C++, more hardware-specific expertise is required to implement the FPGA side of the applications. This has the potential to offset any advantages that might arise from the large amount of parallelism provided by the FPGAs if the application developer is unable to efficiently exploit them.

As mentioned previously, the many coincidental similarities between the cluster and such a highly-developed commercial system suggest a solid foundation for the cluster's architecture and validate many of the design choices made during its inception.

3.5 Example Applications

There are many potential applications that could take advantage of the large amount of parallel processing power offered by the cluster. Although standard sort-first tile rendering is the primary application of the cluster, it is by no means the only one. Here we will look at some of the most popular areas that stand to benefit most from all the available parallel computational power:

- **Video Processing:** Another group involved in this project is Trinity College's Signal Processing and Media Applications Group (SigMedia). They perform research on motion estimation - the study of extracting motion information from visual media processing systems. They have already demonstrated the use of commodity GPUs to accelerate motion estimation [65, 66], and cite poor frame buffer readback speeds as the main hindrance to a complete solution. By employing the cluster in this respect and using the built-in DVI readback port, this problem will be resolved. The parallel computation afforded by the FPGA co-processors can also be used to achieve further acceleration in this respect, as can the fact that multiple GPUs are used by the cluster in the first place.

- **Volume Rendering:** As discussed in the next chapter, volume rendering is an area ripe for parallelisation. Recent work has been done by Strengert et al. [126] on texture-based direct volume rendering of large datasets using GPU clusters, with impressive results; on a Myrinet-based cluster of 16 1.6GHz AMD Athlons with NVIDIA GeForce4s, they achieve 5.7 frames per second for a $2048 \times 1024 \times 1878$ volume in a 1024^2 viewport. Using sort-last compositing, they note that performance is restricted by both the blending computation and the interconnect latency. Both of these problems can be addressed in our cluster - the FPGAs for parallel sort-last compositing and the low latency of SCI for data transfer. Additionally, using the DVI readback would further improve compositing rates compared to readback over the bus.

- **Isosurface Extraction:** The methods described in Section 4.4 for isosurface extraction on Cell can equally be applied to the architecture of the cluster, with

the additional advantage of being able to distribute the resultant geometry to multiple GPUs for sort-first rendering.

- **Ray Tracing:** An obvious graphical application to such an inherently parallel architecture is ray tracing. Each component of the cluster lends itself to ray tracing acceleration in a different way. The FPGAs can be employed to perform ray tracing as demonstrated by the SaarCOR project [128] on a single FPGA - using the many FPGAs of the cluster in parallel would provide many times the performance of that architecture. The fact that a ray can travel anywhere in a scene leads to the problem of needing to replicate all scene data on every node in a classical cluster ray tracing architecture - the shared memory provided by the SCI interconnect would eliminate this need and also allow fast transfer of the data. GPUs are only now becoming powerful enough to perform ray tracing, and research is still ongoing as to the best methods of implementing it on such a specialised architecture [110]. As such, they can still be used to accelerate parts of the ray tracing, or perform hardware accelerated image-based post processing.

- **Collision Detection:** As discussed in Section 2.3.4, Raabe et al. have already demonstrated large improvements in collision detection algorithms implemented on reconfigurable hardware. This can be extrapolated to the cluster's parallel FPGAs being able to accelerate collision detection even further, as part of a larger simulation and rendering system involving the workstation nodes and GPUs.

- **Arbitrary Hardware Shaders:** One potential application of the FPGA co-processors is as another arbitrary shader stage. In this way they could emulate an extra processor stage in the programmable pipeline. An example would be implementing a hardware subdivision surfaces [26] shader, tessellating geometry in the FPGAs and forwarding the produced geometry to the GPUs.

- **Crowd simulation:** The independent nature of individuals in a crowd is very amenable to parallelisation on such a cluster. The crowd could be partitioned, spatially or otherwise, and distributed evenly across the cluster for

advanced behavioural simulation before being rendered on the same node using the methods described in Section 5.2.

- **Other Compute-Bound Applications:** Apart from graphics and image-related algorithms, the large amount of parallel processing power available on the cluster can be applied to any other compute-bound problem domain. The large bandwidth and low latency of the interconnect combined with the distributed shared memory space provides efficient, fast and transparent data transfer between nodes, allowing very large datasets and good scalability. Any algorithms that can be adapted to use the GPUs for GPGPU can expect further increases in speed, without the usual GPGPU drawback of slow readback, thanks to the DVI feedback feature on the custom boards.

In this chapter we have described a tightly-coupled rendering cluster that employs many different levels of parallelism to accelerate simulation and rendering. The remainder of this thesis presents the first potential applications that will be implemented on this cluster framework, scientific visualisation and entertainment algorithms, and the research that has been done on improving these algorithms to take advantage of commodity parallel hardware.

Chapter 4

Scientific Visualisation

The real-time, interactive visualisation of scalar volumetric fields is a desirable goal for many scientific applications. In medical fields, the application of volume rendering and surface extraction for rapid and meaningful visual representation of datasets such as CT (Computerised Tomography), MRI (Magnetic Resonance Imaging), Ultrasound and PET (Positron Emission Tomography) scans can make an important difference in the speed of surgical planning, diagnosis and treatment. It is also a useful tool in surgical simulation and medical education. However, volume visualisation is not only useful for medical imaging, but also for other areas such as rendering the datasets captured by confocal microscopy, and for extracting polygonal structures and surfaces from these volumes.

Section 4.1 gives an overview of the area of scientific visualisation, concentrating on volume rendering and surface extraction on commodity parallel hardware. Section 4.2 details the research done on the visualisation of confocal microscopy data. Section 4.3 presents a method for simplifying volumetric datasets for faster surface extraction and improving the quality of confocal dataset surfaces. Section 4.4 describes a novel algorithm for performing fast surface extraction on the Cell processor in order to take advantage of its parallel architecture. Finally, Section 4.5 presents the results of this research.

4.1 Volume Visualisation

A *volume* can be defined as a 3-dimensional array of point samples. Each sample is called a *voxel*, an abbreviation of Volume Element and analogous to the role of a pixel in a 2D image. A volume may represent any 3D field of discrete scalar data, usually in the form of a rectilinear grid of values that have been sampled from the original continuous domain that the volume represents. For example, a single voxel in a volume captured by a CT scan represents the radiodensity (in Hounsfield units) of the subject sampled at that particular point. Volumes can also be based on other topographical grids, such as curvilinear or unstructured fields of points. Voxels are generally either 8, 16 or 32 bits in size.

In this section we will concentrate on two problems; the actual volume visualisation of collected data in both software and hardware, and the reconstruction of polygonal surfaces from the volume data. The efficient and meaningful rendering of volumes such as these is just as important as the capture itself, as being able to acquire such data is of no use without the ability to extract meaningful information from it. The two most popular methods of achieving this goal are *direct volume rendering* and *isosurface extraction*.

4.1.1 Direct Volume Rendering

Direct Volume Rendering (DVR) is concerned with the rendering of a volume without the intermediate step of extracting explicit polygonal data from the volume. In order to carry out DVR, three items are needed; an optical model of the volume, a method of classification, and a method to project the volume into the frame buffer.

Optical Models

Firstly an optical model of the volume is needed. This model determines how each particle in the volume reacts to light, and how they will be rendered into the frame buffer. The most common optical models used in DVR are as follows, summarised from [86]:

Absorption: Particles are completely black and absorb all incoming light without scattering it to nearby particles, or emitting their own light. In absorbing the

incoming light, they occlude other particles.

Emission: Particles do not absorb incoming light, but only emit their own light.

Scattering: Any incoming light to the particle is scattered to other nearby particles

In general, a combined emission/absorption model is used, together with a lighting model. This means that particles in the volume both absorb and emit light and react to the light's direction and colour, but do not scatter incoming light to nearby particles.

Classification

In order to convert the scalar voxel values of the initial volume data into meaningful optical properties such as colour and opacity, a *transfer function* is needed. Transfer functions are generally implemented as one-dimensional colour lookup tables, mapping the scalar values directly to an RGBA value.

The transfer function can be applied directly into the texture data upon initial generation of the stacks, replacing the scalar values with the results of the transfer function lookup. This is known as *pre-classification*. However, there are a number of disadvantages associated with pre-classification. It is inefficient, as the entire volume needs to be regenerated and updated whenever the transfer function is altered. In addition, it leads to artefacts in interpolation. An interpolated pre-classified colour might not be the same as the colour that would result from evaluation of an interpolated voxel value by the transfer function.

Post-classification is used instead, where the voxel data is stored in the textures directly and only converted into optical properties upon rendering, and after filtering. Post-classification can be accomplished in one of two ways: the first is to use OpenGL colour tables, where an 8-bit texel intensity value is used to look up a one-dimensional colour table. The alternative is to use dependent texture reads, where certain channels of the fragment colour resulting from one texture lookup are used as texture coordinates for a second texture. While somewhat slower, dependent texturing is more flexible than OpenGL colour tables, and can be used for implementing multi-dimensional transfer functions [72].

Volume projection

Without an explicit geometric model to display through the normal rasterisation pipeline, an alternative method is required in order to convert the 3D volume into a projected 2D image in the frame buffer. This is usually done by *ray casting* [135, 79]. Ray casting is the procedure of sending rays through the image plane into the volume, re-sampling the volume at regular intervals. Because the volume of voxels represents a discrete sampling of the original continuous data, this re-sampling is necessary in order to reconstruct the original data as closely as possible. It is performed by using a filter such as trilinear interpolation. Classification occurs either before or after this filtering step, depending on the type of classification being used (pre- or post-classification). The contribution of each voxel encountered along the ray is then integrated to produce the final pixel colour. The nature of this contribution is governed by the optical model chosen for the DVR procedure, as described above. It is possible to achieve very high image quality by increasing the number of interpolated samples along each ray, at the expense of processing speed.

Related Work

After the original work on ray casting was published, there were a number of papers on accelerating and optimising its performance. Levoy [80] proposes two key ideas for reducing rendering costs; adaptive ray termination which terminates rays early when their accumulated opacity reaches a certain threshold, and a hierarchical spatial partitioning scheme for fast traversal of empty areas of the volume. Danskin and Hanrahan [22] compare a number of ray casting acceleration techniques, and adapt two of the techniques in order to produce an improved algorithm that increases performance without degrading image quality.

Another class of volume rendering is *splatting*. This is where the voxels are sorted from back to front, and the projection of each voxel is composited as a *splat* into the frame buffer [142, 78]. However, the image quality produced by splatting is not comparable to that of ray casting. One advantage of splatting is that it is more parallelisable than ray casting; the volume can be partitioned and distributed to many nodes, where each node only needs the information local to the voxels being splatted. Mueller and Yagel [95] propose a hybrid method of splat-based ray

casting, and present an efficient method of addressing and intersecting the splats. This combines the speed of splatting with the acceleration techniques possible for ray casting, such as early ray termination and bounding volumes.

Ray casting can also benefit from parallel processing, much as ray tracing can by distributing rays to different computational nodes in a divide-and-conquer approach. Ma et al. [83] describe a system for parallel processing of ray casting for volumes, and subsequent parallel composition of the resulting partial images using binary-swap compositing. They demonstrate communication costs as being only a small overhead of the overall processing time, indicating successful parallelisation.

A new approach to DVR was presented in 1994 by Lacroute and Levoy, called *shear-warp* [77]. This is a high-performance volume rendering algorithm that eliminates the costly voxel viewing transformation overhead of other methods. It does this by shearing the volume slices in order to be able to easily project them onto an image plane, which is then warped to produce the final image. Although shear-warp has the potential to produce inferior images compared to ray casting due to less accurate sampling, this fact is compensated for by its speed as a volume renderer. Additionally, Levoy notes that the latter steps of the algorithm are very suitable for parallelisation.

With the exception of the shear-warp algorithm, software volume rendering algorithms are generally too slow to use for interactive real-time volume rendering. Therefore it was not long before researchers started to look towards graphics hardware in order to increase the rendering speed and interactivity of volume viewing.

4.1.2 Volume Rendering on Commodity Graphics Hardware

In recent years, research in the area of volume visualisation has advanced rapidly due to the increased availability of high speed, low-cost commodity graphics hardware. Volume data that could previously only be interactively explored using expensive workstation hardware can now be rendered at interactive rates on a common desktop PC by taking advantage of the texture mapping hardware in today's commodity graphics cards.

Much like ray casting, the process of texture mapping is concerned with resampling discrete data. A continuous domain is sampled into the discrete units of a

texture (texels), which approximate the original data. Texture mapping hardware then filters this texture in order to re-sample the data and map it onto a polygonal surface. The similarity between these two procedures can be exploited in order to perform the repetitive re-sampling task of ray marching, but with the advantage of being hardware accelerated.

The basic method of visualising stacks of images such as these is by using 2D-textured axis-aligned slices [141, 71]. The volume is separated into textured slices, each slice being a two-dimensional section of the volume, one voxel in depth. *Proxy geometry* is generated in the form of a quad for each texture slice. For example, a volume of dimensions $128 \times 128 \times 64$ would be separated into 64 textures, each 128×128 in size. These textures are arranged to give the illusion of a solid volume by blending each slice from back to front into the frame buffer using the graphics hardware's support for texture blending. Blending the textures together approximates the ray integration step of ray marching, thereby producing a rendered volume.

However, when the texture stack is rotated past a certain point, gaps between the slices become apparent and the proxy geometry becomes evident (see Figure 4.1). To avoid this, 3 stacks must be generated, one for each principal axis. The original stack is used for the Z-axis, with interpolated stacks generated from this data used for the X- and Y-axes. As the volume is rotated, the stack that is most orthogonal to the viewing direction is displayed, thus preserving the illusion of solidity. The main drawback of this method is that it consumes 3 times the amount of texture memory as the original stack. However, this is unlikely to cause a major problem given the large amount of texture memory on modern graphics cards together with the increasing amount of bandwidth available for mapping AGP and PCI-Express texture memory. Another drawback of using axis-aligned slices is that there is a visible transition between the three different stack representations of the volume.

An alternative that eliminates this memory consumption requirement is the use of 3D textures, as observed by Cabral et al. in their work implementing a real-time volume renderer on the SGI Reality Engine [13]. Graphics hardware supports the binding of the entire volume as a single 3D texture. Geometry can then be texture-mapped using three texture coordinates at each vertex to apply any arbitrarily-oriented section of the volume to the polygon's surface. By taking advantage of this functionality, the axis-aligned proxy geometry can be converted into view-aligned

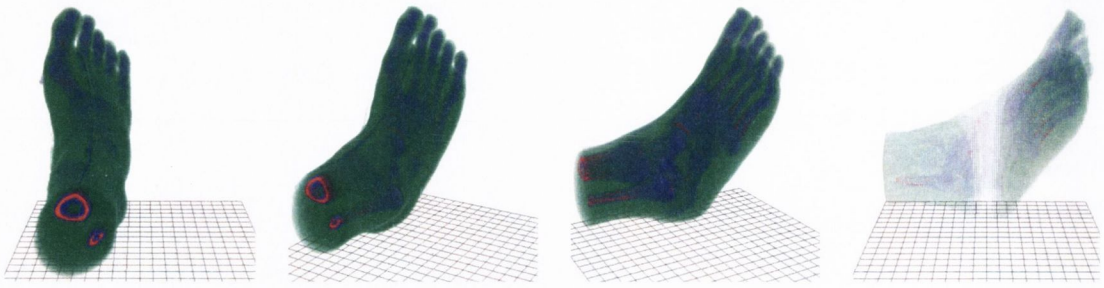


Figure 4.1: A single stack of textures at 0°, 30°, 60° and 90°. Upon each rotation, the underlying proxy geometry becomes more evident.

slices that are generated dynamically as the viewpoint is altered. This eliminates any visible transition that might be apparent using axis-aligned slices. The hardware will then filter the 3D texture using trilinear interpolation in order to sample texture for mapping to the slices, leading to improved image quality. However, 3D texture support on graphics hardware is slower than regular 2D texturing [117], so a trade-off decision must be made between memory usage, image quality and rendering speed, depending on the target platform and application needs.

Custom Volume Rendering Hardware

Given the specific nature of volume rendering, the widespread areas of application, and the inherent parallelism in its implementation, a number of custom-built hardware solutions have been published over the last ten years.

Pfister and Kaufman's work on the Cube-4 [109] builds upon previous iterations of the Cube architecture. Cube-4 is a scalable array of volume rendering pipelines and associated memory modules, which work on local voxels in parallel and process them with a modified ray casting algorithm. The system scales linearly with the number of rendering pipelines, and is aimed at implementation in an ASIC. Osborne et al. continued their work and presented EM-Cube [101], a feasible architecture for using the Cube-4 in a PCI card that could process 16-bit 256^3 volumes at interactive rates.

At around the same time, Kreeger and Kaufman were working on PAVLOV [76], a parallel architecture with SIMD elements. While PAVLOV is capable of volume rendering, it is also designed to be programmable, allowing the use of different

rendering algorithms and properties such as feature extraction and segmentation.

In 1999, Pfister et al. produced the VolumePro [108], a PCI volume rendering board manufactured by Mitsubishi. It was a commercial implementation of the EM-Cube architecture, costing around $3,000 at the time of release. VolumePro was capable of 500 million interpolated, illuminated and composited samples per second, enough to display a shaded 256^3 volume at 30 frames per second. Today, the VolumePro range is sold by TeraRecon Inc., and the latest incarnation (the VolumePro 1000 D) is capable of real-time visualisation of volumes up to 512^3.

Most recently, Meißner et al have published the Vizard II [88], another volume rendering PCI card. However, unlike other custom volume rendering hardware, the Vizard II is implemented on an FPGA, thereby allowing a customisable feature set and quick implementation of upgrades and optimisations. While the Vizard II is only capable of a maximum of 50-100 million samples per second depending on board speed, they claim a superior image quality to the VolumePro, and a more efficient and correct implementation of the ray casting algorithm.

4.1.3 Isosurface Extraction

The alternative to Direct Volume Rendering is Indirect Volume Rendering. This is where an explicit polygonal surface is extracted from the volume as an intermediate step and rendered using the traditional triangle rasterisation pipeline of modern graphics cards. These surfaces are called *isosurfaces* - in the same way that isobars demarcate areas of equal pressure on a weather map, isosurfaces are the representations of areas of equal data values (referred to as *isovalues*) in a volume.

Besides the obvious applications of isosurface extraction for visualising internal organs and bones in a medical dataset, isosurface extraction is also useful in other areas where specific surface visualisation can allow for the more meaningful exploration and illustration of collected data, such as in Pharmacology [134], Chemistry [103] and Heliology [62].

Methods for calculating and extracting isosurfaces from a given scalar volumetric dataset have been the subject of much research. Early work by Keppel et al. [67] involved the reconstruction of surfaces by connecting applicable contours on adjacent slices, but was subject to ambiguities concerning how to connect contours when more

than one exists on a slice. Miller et al. [90] use 'Geometrically Deformed Models' which are grown from a seed placed in the model and deformed according to a set of constraints. They produce closed models, and relate the computational time of their algorithm to the size of the surface produced, not the size of the volume itself.

However, the most popular isosurface extraction algorithm has been in use since Lorensen and Cline introduced Marching Cubes in 1987 [81]. Marching Cubes constructs a cube or '8-cell' from eight voxels, four each from two adjacent slices. It then iterates through every cube in the volume, comparing the voxel values at each corner of the cube and determining whether an isosurface for a desired isovalue would intersect the cube. If all eight of the cube's values are below the desired isovalue, the surface will not intersect it. Likewise if all values are above the desired isovalue. The other cases are where the surface needs to be generated - some values are below and some are above. Each voxel therefore has two states - either above (or equal to) or below the isovalue, and eight voxels in a cube leads to $2^8 = 256$ possible cube states with respect to isosurface intersection for a given isovalue. However, many of these states are identical except for rotational symmetry differences. By eliminating these duplicate cases, the number of possible cube states is reduced to a more manageable 14. These cases can be seen in Figure 4.2. When it is known which edges the surface intersects, the exact location of these intersections can be calculated by performing a linear interpolation between the two voxel values of each edge. A number of interpolated triangles are then constructed according to which case the cube falls into, and all the triangles created form the volume's isosurface for the given isovalue.

While Marching Cubes is a simple algorithm that is easy to implement and produces good surfaces, it suffers from the problem of ambiguous cases where there is more than one way to triangulate a given cube. This can lead to holes in the resulting isosurface. Further research was done by Van Gelder and Wilhelms [138] and Nielson and Hamann [97] on adaptations of Marching Cubes to overcome this problem. An extension of Marching Cubes called Marching Tetrahedra [52, 14] solves the problem by separating each 8-cell into a number of tetrahedra, eliminating potential ambiguities and producing a finer tesselated surface.

A cube can be decomposed into a minimum of five tetrahedra. Each tetrahedron has four vertices, so the number of possible isosurface intersection cases is $2^4 = 16$.

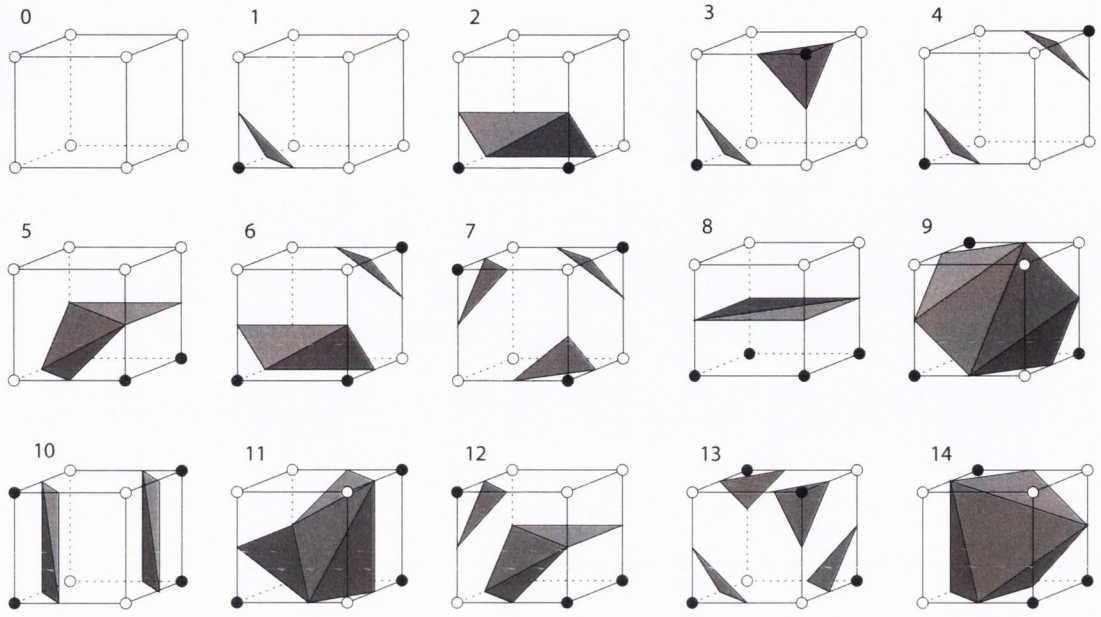


Figure 4.2: The 14 possible cases of Marching Cubes, including the empty case 0.

Again, the majority of these are duplicate symmetric cases which can be eliminated in order to reduce the number of unique intersections to three. See Figure 4.3 for an illustration. These cases do not contain the ambiguities inherent in Marching Cubes, although they do have the potential to produce more triangles. In the worst case, Marching Cubes creates four triangles per cube. Marching Tetrahedra, with a maximum of two triangles per tetrahedron and five tetrahedra per cube, has a worst case of ten triangles per cube, over twice that of Marching Cubes. Another implication of tetrahedral decomposition is that the edges of tetrahedra in adjacent cubes do not match up, so the edges of any surfaces created in these tetrahedra will not match up either, thus producing holes in the surface. This can be overcome by increasing the number of tetrahedra in a cube to six, but at the expense of an even higher triangle count in the resulting isosurface. Instead, the orientation of each alternate 5-tetrahedra cube can be changed so that adjacent tetrahedra always match up.

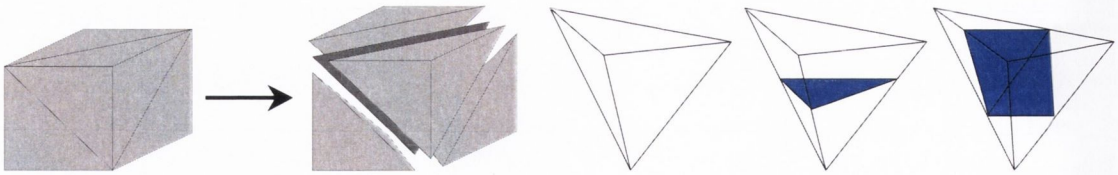


Figure 4.3: The decomposition of a cube into 5 tetrahedra, along with the 3 possible triangulation cases of Marching Tetrahedra.

4.1.4 Accelerating Isosurface Extraction

Given the desire to interactively explore volumetric datasets (i.e., being able to generate new surfaces for a given isovalue parameter at responsive frame-rates), many techniques have been developed in order to accelerate isosurface extraction by Marching Cubes or Marching Tetrahedra.

A popular method of acceleration is by pre-processing the data in order to allow for rapid identification of subsections of the volume known to include a desired isovalue. Wilhelms and Van Gelder propose a modified octree subdivision method [143], keeping storage costs down by using what they call 'Branch-on-need octrees'. Similarly, Cignoni et al. use interval trees [35] to accelerate the identification of 8-cells that contain a desired isosurface [19]. These methods dramatically improve processing speeds at the expense of additional memory usage for the required data structures, thus reducing the memory available for the dataset.

Others achieve acceleration by amortising the computational cost through parallelisation. Hansen and Hinker were one of the first to propose the parallelisation of Marching Cubes, developing an adapted version of the algorithm that takes advantage of SIMD architecture [54]. At the same time, Mackerras implemented Marching Cubes on a MIMD *Fujitsu AP1000* with 128 processors [85]. More recently, Jinzhu and Shen demonstrated a parallel multi-pass view-dependent algorithm for isosurface extraction on a 40-processor *SGI Origin* [41]. They employed octrees to cull large empty areas of the dataset, and used multi-pass occlusion culling to only extract the areas of the isosurface that will be visible in the final frame. Udeshi and Hansen parallelise both isoextraction and isosurface rendering by using the multiple processors and multiple graphics cards of the *SGI Onyx2 Reality Monster*, using a sort-last paradigm for image recomposition [136].

However, these implementations required expensive high-end multi-processor server architectures. Zhang et al. instead use a cluster of commodity machines and exploit the parallel processing and parallel disk accesses available to such a cluster [147]. They also employ interval trees to reduce the amount of I/O traffic necessary. The result is a scalable, out-of-core architecture capable of extracting isosurfaces from arbitrarily large volume datasets.

4.1.5 Isoextraction on Graphics Hardware

The recent trend of exploiting programmable graphics hardware for general computation has also led to isosurface extraction being performed on the GPU itself by taking advantage of the graphics pipeline's inherent parallel nature.

Pascucci published the first example of isoextraction via Marching Tetrahedra on the GPU in 2004 [104]. He uses the vertex processor to offload all isosurface computation from the CPU. In addition, he introduces a streaming mechanism which exploits the persistent nature of the vertex processor's registers to allow the specification of a new tetrahedron by only transferring a single vertex, as opposed to all four. He also demonstrates a tetrahedral stripping scheme in order to optimise data transfer, and view-dependent refinement to reduce the amount of unnecessary computation. The result is a peak processing speed of approximately 2.1 million tetrahedra per second on a GeForce4. Reck et al. also demonstrate an implementation of Marching Tetrahedra in the vertex processor, employing interval trees in order to reduce the amount of data transfer and video memory consumption [113]. Goetz et al. [43] also perform similar isoextraction in the vertex shader, but use the original Marching Cubes algorithm due to the reduced number of vertices in the resulting isosurface, reducing the amount of bandwidth and vertex processing necessary.

The limitation of these methods is that the vertex processor is unable to either create or delete vertices, only process them. Therefore, for every tetrahedron that is potentially part of the final isosurface, two triangles must be submitted for processing whether they are used or not. Vertices that are not needed are rendered as degenerate triangles of zero surface area, and are culled efficiently by the hardware. However, these extraneous vertices must still be transferred, thereby unnecessarily

using bandwidth and video memory space that is already at a premium. As discussed previously, the amount of computation performed in the fragment shader is necessarily more than that of the vertex processor, and this difference will only increase as more fragment pipelines are added to next generation GPUs. Therefore, it makes sense to move as much computation to the fragment processor as possible in order to take advantage of this increased parallelism.

As a result, the most recent papers on isosurface extraction on the GPU do so in the fragment processor. Klein et al. [70] report a peak of 7.2 million tetrahedra per second on an ATI Radeon 9800, using an experimental ATI_super_buffer OpenGL extension which allows rendering directly to a texture (this extension has since been altered and accepted by the ARB, and is currently known as EXT_framebuffer_object). This texture can then be bound as a vertex array and used as the source of a vertex stream, without having to be copied over the bus. Kipfer and Westermann [69] perform a similar procedure, but approach it with the aim of minimising the number of operations and memory accesses. They achieve this by sorting the element vertices and processing them with an edge-based algorithm that allows the entire isosurface to be constructed with less work than previous methods. By using an interval tree, they achieve an isoextraction processing time of 69.4 million tetrahedra per second. An additional step of normal computation and final lit rendering of the isosurface results in a speed of 57.1 million tetrahedra per second.

It should be noted that all of the performance numbers quoted in the above GPU-based isosurface extraction methods are only achievable if the isosurface to be generated fits entirely in video memory. If this is not the case, vertex or texture data will be swapped out to AGP memory, resulting in a severe drop in processing speed. As a result, GPUs can achieve extremely high isosurface processing speeds due to the large amount of internal parallelism, but the size of the volumes that can be processed at this speed is relatively restricted.

4.2 Accelerated Visualisation with Parallel Hardware

This section is concerned with accelerating the visualisation of confocal microscopy datasets. Although much of the current work on volume visualisation is aimed at visualising medical scans acquired by CT or MRI, many of the same methods can be applied to the display of confocal fluorescence microscopy data.

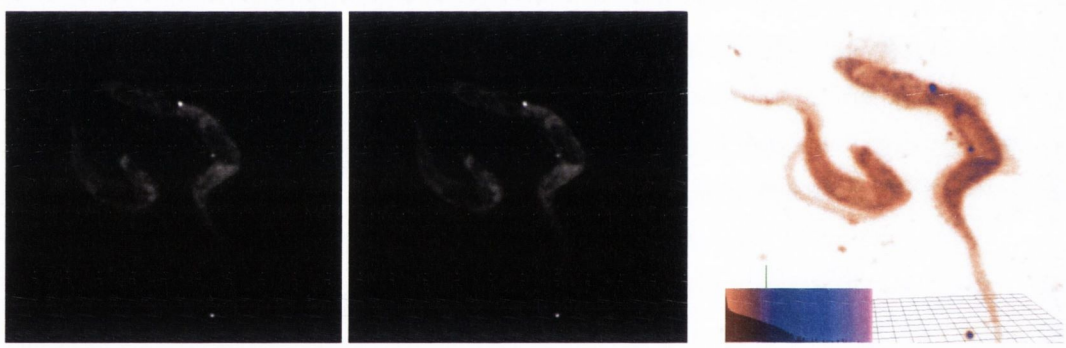
4.2.1 Confocal Fluorescence Microscopy

The essential principle of confocal fluorescence microscopy relies on the fact that there are two pinholes in the optical path. The first intercepts the light beam after leaving the light source and before striking the specimen, and the second intercepts the light after leaving the specimen and before entering the detector. Then, at any fixed distance between pinholes and fixed position of lenses, only one plane normal to the light path will be in focus within the specimen, i.e., will be confocal with respect to both pinhole positions. In the commercial instruments available, the most common way to produce optical sections is to move the specimen in successive steps in a line between the two pinholes, termed the "z" direction, thus bringing into focus successive planes within the specimen. The thickness of the plane in focus is inversely related to the diameter of the pinholes and has a practical limit of about 0.2 of a micron. This optical arrangement effectively removes light from the planes within the specimen that are not in focus, commonly known as stray light and, consequently, sharpens the image at the focal plane recorded by the detector. Most of the image recording methods rely upon a raster scan of each of these optical z-planes using a laser light source and a photomultiplier detector coupled to a digitiser to produce a set of ordered datasets. Each dataset consists of an ordered array of data pairs that specify position and optical intensity at that position.

Various methods of presenting the data have been used in the past. These methods range from displaying each optical section as elements in a planar array of images arranged in step-order of z-direction, to creation of a pair of stereoscopic views by reconstructing the image from two separate angles through the three dimensional dataset. The angles are chosen to approximate those formed by a pair of human eyes

viewing a specimen. Finally, the most common approach has been simply to superimpose all of the z-stack images in one single image. This last technique produces a sharper image than can be collected with ordinary epi-fluorescence microscopy because of the elimination of stray light in each plane, but has the disadvantage of losing the z-directional information. New methods of presenting the data that reconstruct a partially transparent 3D image with perspective along the Z-axis for ease of publication and easy viewing by the reader are highly desirable.

Current commercial methods for visualising confocal data such as stereoscopic cross-eyed images (where the user must cross their eyes in order to view the image in 3d - Figure 4.4(a)) are difficult to use and do not lend themselves well to viewing or publication, nor allow for interactive investigation of the volume. By contrast, the stacks of image data collected by confocal microscopes can be readily used to create an interactive and customisable rendering of the entire volume (Figure 4.4(b)) using hardware-accelerated volume rendering methods.



(a) (b)

Figure 4.4: (a) Cross-eyed stereo depiction of Trypanosoma Brucei. (b) Interactive texture-based Direct Volume Rendering application with the same dataset.

4.2.2 Interpolated slices

Given the sub-micron scale of data generally collected by confocal microscopes, often the resolution limits the number of slices collected to a small amount. Displaying this data while preserving the scale and spacing of slices in the original specimen produces gaps and banding artefacts. While these problems can be remedied by inserting extra interpolated slices upon initial generation of the principal-axis stacks,

this unnecessarily increases the amount of texture memory used by a factor of two or more, depending on the number of slices inserted.

Instead, slices can be inserted during run-time and interpolated without extra memory usage by employing multi-texturing and programmable graphics hardware, as demonstrated by Rezk-Salama et al [116]. We use NVIDIA's Register Combiners [123] to perform this interpolation. As described in Section 2.1.3, the final combiner can be used to perform interpolation on the outputs of previous combiner stages. Possible inputs to any stage include texture lookups, primary/secondary colours and application-set constants. Thus it is possible to use the final combiner to interpolate between two texture slices by assigning the start and end textures to two inputs, and an interpolation factor to a third input signifying the distance of the inserted slice between the two existing slices.

Bilinear image filtering can be performed on 2D textures using graphics hardware. By combining this with the linearly interpolated extra slices, the outcome is trilinear interpolation with similar results to that performed with 3D textures, but at improved frame rates. The number of extra slices inserted can be altered during run-time; in order to preserve interactive frame rates, these interpolated slices are only displayed when user input is not being received. Therefore, a large number of extra slices can be introduced to achieve dramatically increased image quality without hindrance to the user. We can see in Figure 4.5 the significant increase in quality that even one extra slice can give.

4.2.3 Hardware Accelerated Transfer Functions

NVIDIA's Texture Shaders [33] extend OpenGL's standard texture addressing operations, allowing for an additional variety of operations such as dot product operations, offsetting and dependent texturing. There are two different types of dependent texture shaders, Alpha-Red and Blue-Green. These determine which colour channels of the first texture unit are used as texture coordinates for the second unit's addressing operation (as already seen in Figure 5.7).

We implement the transfer function using an Alpha-Red texture shader. When the initial textures are being uploaded, their internal format is set to GL_ALPHA8, so each density value is stored as an 8-bit alpha value. This is bound to the first

Figure 4.5: Left: a rendering of a 128^3 CT scan of a foot. Right: the same rendering with one slice inserted dynamically with programmable hardware.

texture unit, and a one-dimensional RGBA texture representing the transfer function is bound to the second unit. When texture shaders are enabled, the alpha value of the slice texture is converted into the corresponding colour and opacity according to the transfer function texture. This texture can be edited while the application is running and re-uploaded to alter the visualised colours in real-time. An example of the transfer function used for Figure 4.4(b) can be seen in the bottom left corner, upon which is superimposed a logarithmic histogram of density values present in the volume. While opacity information can be included in the transfer function texture and applied to the resulting visualisation, we have found that given the inherent noisy nature of confocal microscopy data, better results can often be obtained by using purely opaque colours. We have also implemented the transfer function using the GLSL high-level shading language.

4.3 Volume Simplification

The Marching Tetrahedra algorithm (as described in Section 4.1.3) was also implemented for indirect volume rendering of extracted isosurfaces, both from medical imaging machines and from confocal microscopes (see Figure 4.6).

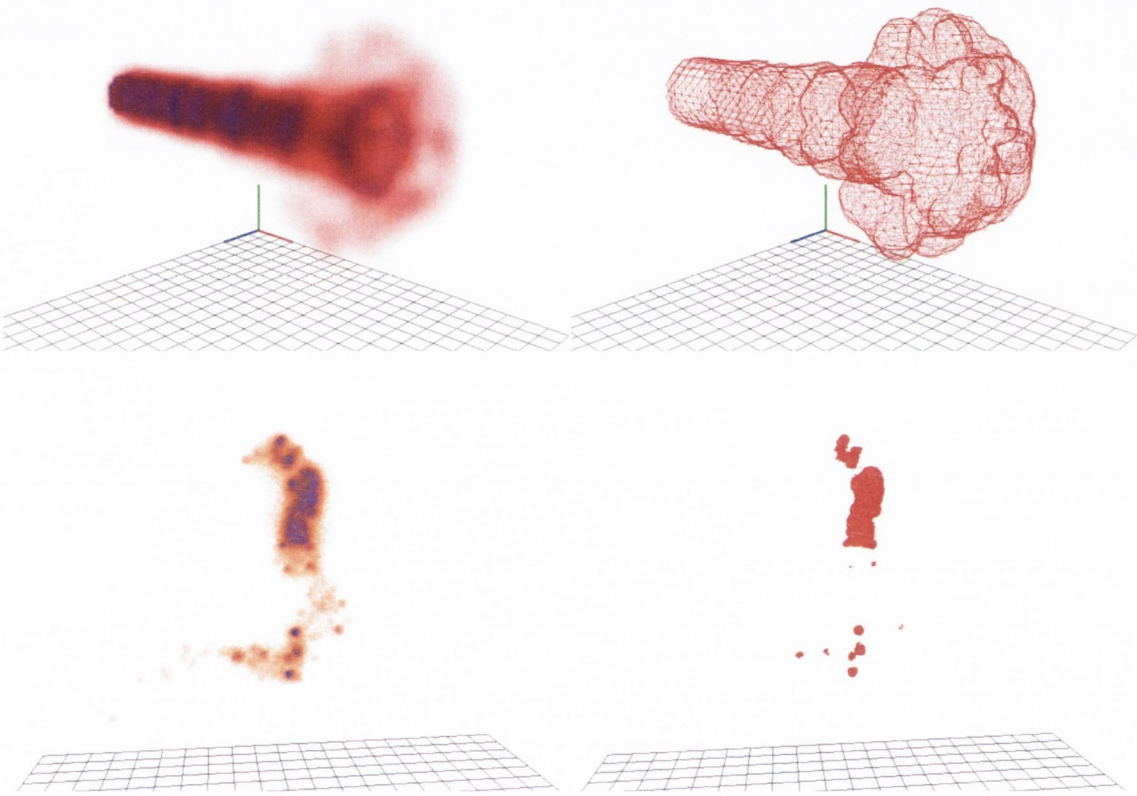
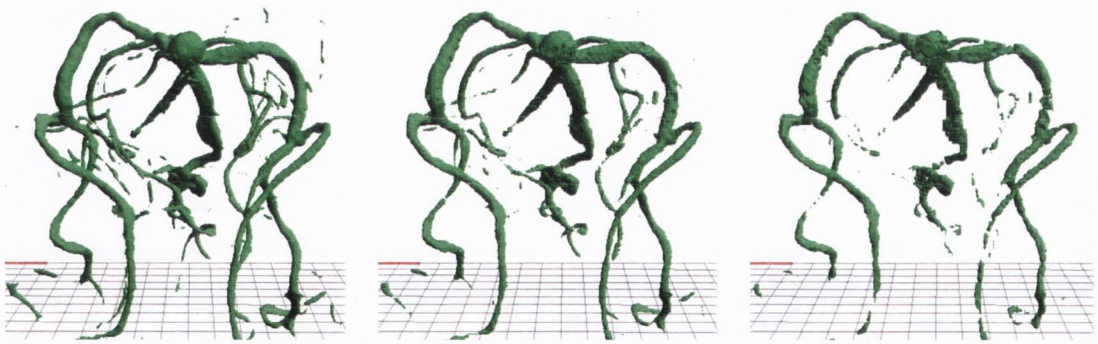


Figure 4.6: Top: a 64^3 simulated volume of fuel injection into a combustion chamber, DVR (left) and isosurface wireframe (right). Bottom: a confocal microscopy scan of Trypanosoma Brucei, DVR (left) and isosurface wireframe (right).

However, the performance of un-optimised Marching Tetrahedra for any reasonably sized volume precludes real-time interaction or investigation of the dataset. Additionally, the noisy images captured by confocal microscopy can produce overly spiky isosurfaces that do not satisfactorily convey the shape of the desired surface (for example see Figure 4.8(b)). To achieve this, we have investigated the simplification of the dataset before generation in order to reduce the amount of processing necessary to produce the surface.

The approach taken was to remove a number of voxels from the volume, but still attempt to keep the overall shape and salient details of the dataset. A naïve attempt of simply skipping every n voxels in the dataset produces an overly blocky surface which quickly loses the details of the original dataset (see Figure 4.8(c)). However, it does dramatically decrease computation time, so an attempt was made to improve upon this method by incorporating the data from the excluded voxels into the remaining voxels. This was done with a 3D averaging filter using a Gaussian kernel. In order to attempt to retain the shape of the surface and prevent degradation, any excluded voxel that has a value of zero is not included in the filter. The value of n can be varied depending on the level of simplification required.

The result is a simplified and reduced version of the volume that can be used for isosurface extraction with the usual Marching Tetrahedra algorithm (see Figure 4.8(d)). The produced surface still retains the rough shape of the represented isosurface up to a point, but excessive simplification results in an isosurface that bears no resemblance to the original dataset. Due to the method of averaging, fine details such as tendrils or other thin or narrow structures are quickly lost in the first couple of simplification levels, but the overall shape of the surface can still be discerned (see Figure 4.7).



(a) (b) (c)

Figure 4.7: Volume simplification of the 512^3 Head Aneurysm dataset for different values of n. (a) $n = 1$, (b) $n = 2$, (c) $n = 3$. At each level, thin structures and fine detail get lost.

For this reason, this method is a perfect candidate for quick isosurface simplification that does not require the entire surface to be fully represented. Typical examples of this are Level Of Detail representations for distant objects, where the

evaluation of an isosurface at every voxel is not necessary and contributes little to the final image. Another use is the progressive transmission of isosurfaces. Very simple coarse representations of the volume can be transmitted first and reconstructed, giving a broad overview of the volume and costing little in terms of bandwidth and transmission time. If a more detailed surface is required, the excluded voxels can be subsequently transmitted and inserted into the already-constructed volume. A more detailed isosurface can then be extracted.

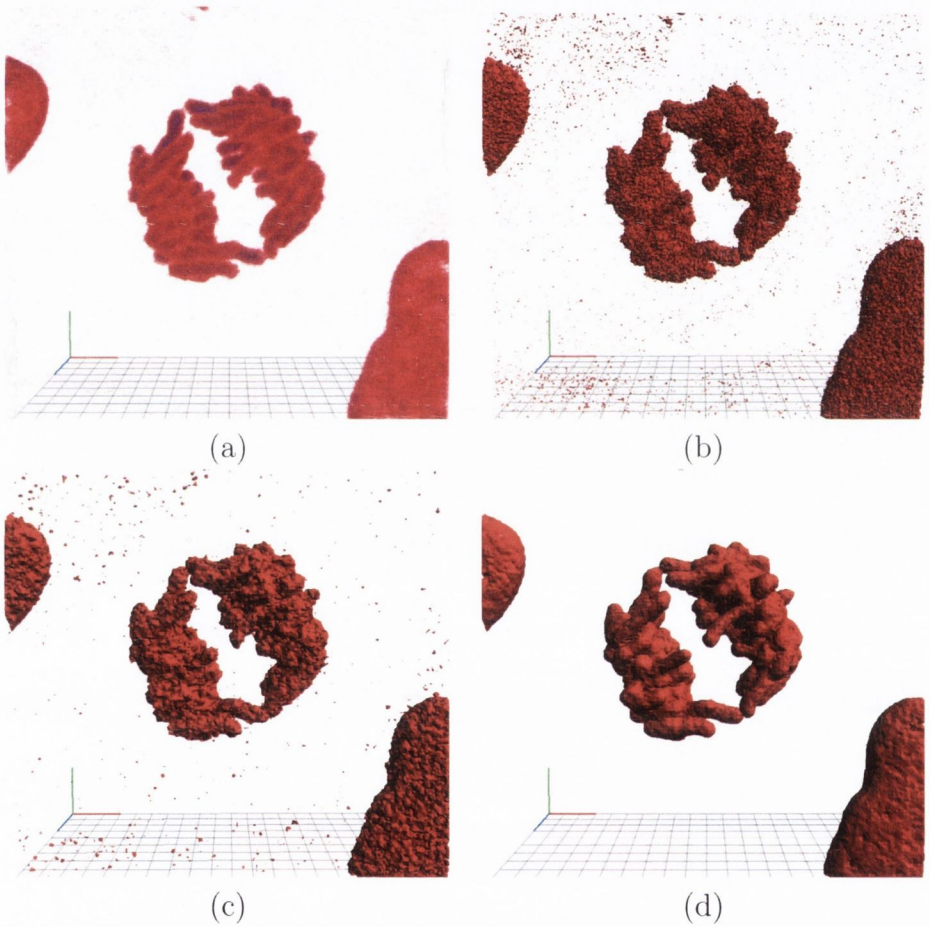


Figure 4.8: Volume simplification. (a) The original 'chromatid separation' dataset of size $256 \times 256 \times 79$, rendered with texture-based DVR. (b) The isosurface extracted from the original unaltered dataset. (c) Naïve simplification by skipping voxels. (d) Improved simplification using 3D Gaussian filtering where $n = 1$.

An additional advantage of this method is that small levels of simplification can actually improve the visual quality of noisy volume polygonal representations, such

as those produced by confocal microscopy. This results in a smoother surface that is a better candidate for illustrative or structural identification purposes (for example, compare Figure 4.8(b) and Figure 4.8(d)).

At the same time, it should be noted that the more the size of the simplification kernel is increased, the more volume data is averaged and as a result high frequency details are lost. This tends to produce holes in the resulting simplified surface, which restricts the usefulness of this algorithm to the first few simplification levels. This is especially true in low resolution datasets, where any over-simplification can cause significant features of the volume to be lost.

This algorithm increases runtime isosurface extraction time by introducing an offline simplification step which can be done as by pre-processing the dataset. While this can be performed once for each simplification level and stored along with the dataset, this step could itself be further accelerated by using the GPU to perform the filtering. This can be accomplished by using GPGPU methodologies to interpret the dataset voxels as texture samples and implementing a filtering kernel in the fragment processor to compute the averages of multiple texels, accumulating the results into intermediate textures. By doing this in slices in both the X-Y and Z-Y dimensions, we can build up a filtered volume in multiple passes and read back the resultant simplified dataset to main memory via the framebuffer.

Indeed, the Cell processor could also be used to perform simplification in much the same way as it can be used for the isoextraction algorithm itself, as described in the next section.

4.4 Isosurface Extraction on the Cell Processor

In this section we describe a new algorithm for performing isoextraction on the new Cell processor architecture. When dealing with large datasets such as those used in medical visualisation, a significant amount of data must be processed in order to extract a desired isosurface. In this section we focus on accelerating the computation of isosurface extraction through the parallelisation of a marching tetrahedra algorithm on the Cell processor. Cell provides significant increases in memory bandwidth and processing speeds while still being available at prices comparable to desktop processors - this allows algorithms such as isoextraction, which were previously the domain

of supercomputers and workstation clusters, to be executed on desktop machines.

4.4.1 Cell Applicability to Marching Tetrahedra

Cell is particularly suitable for isosurface extraction by Marching Tetrahedra (MT) in two areas; parallelisation and data transfer latency/bandwidth.

Parallelisation: Like the graphics pipeline, isosurface extraction is also an embarrassingly parallel problem. In MT, the processing of each tetrahedron is independent from the next, requiring no knowledge of its neighbours in order to determine the location of the intersecting isosurface (if any). This means that any tetrahedron can be distributed to any SPU for independent processing.

Data transfer bandwidth & latency: Any algorithm dealing with volumetric datasets must process large amounts of data, which can quickly become a bottleneck if the system executing the algorithm is incapable of keeping the processor fed with data. The high bandwidth of the CBE, combined with the SPU's DMA mechanism for hiding storage latency, eliminates any potential data transfer bottlenecks.

4.4.2 Implementation

This sections details the adaptation of the MT algorithm for implementation on the Cell processor. Broadly speaking, the process is as follows:

1. The volume is partitioned into slices.
2. The slices are partitioned into chunks.
3. The chunks are assigned to different SPUs.
4. Each SPU iterates over every pair of slices and processes the assigned chunks of those slices.
5. Tetrahedra are constructed by iterating through every 8-cell associated with each voxel in the chunk - four from each adjacent slice.
6. Triangles are produced by evaluating each tetrahedron according to MT.

This involves three steps; volume partitioning, data transfer, and processing.

4.4.3 Volume Partitioning

For each SPU to perform a comparable amount of work, the dataset must be partitioned before it can be distributed. We accomplish this via a two-level partitioning scheme, with the additional aim of minimising both data replication and transfer costs.

First, the volume is logically divided into slices. A 3D volume of dimensions (x, y, z) can be considered as being a collection of z 2D slices, with each slice consisting of y number of rows where every row contains x voxels. We determine these slices according to contiguous areas of data in memory, as it is more efficient to access and transfer a few large contiguous blocks of memory than many small blocks. This format is typically how the data is stored offline on disk.

Then, every slice is split into n *chunks* for distribution to n available SPUs (see Figure 4.9 for an example where $n = 4$). A chunk consists of several rows of data, each chunk overlapping adjacent chunks by one row. The reason for this is that for each tetrahedron, the MT algorithm requires data from two adjacent rows in order to build an isosurface. An SPU works with two chunks at a time for the same reason, these chunks coming from two adjacent slices. For each chunk except the last one, the number of rows is calculated by the formula $round((r + (s - 1))/s)$, where r is the number of rows not yet assigned to an SPU, and s is the remaining number of SPUs including the current one. The inclusion of $(s - 1)$ is to account for the overlapping rows. The final chunk is then assigned all remaining rows.

This method ensures that each SPU receives an approximately equal section of the overall volume. For example, if $n = 8$ and $y = 128$, a chunk size of 16 rows will be assigned to the first SPU, with the other seven SPUs being assigned chunks of 17 rows. Thus a total of 135 rows have been assigned; 128 rows plus the 7 overlapping rows which have been assigned to two SPUs.

4.4.4 Data transfer

As described above, the SPU has 256KB of storage immediately available to it, which puts a limit on its ability to process locally stored data. Additionally, the

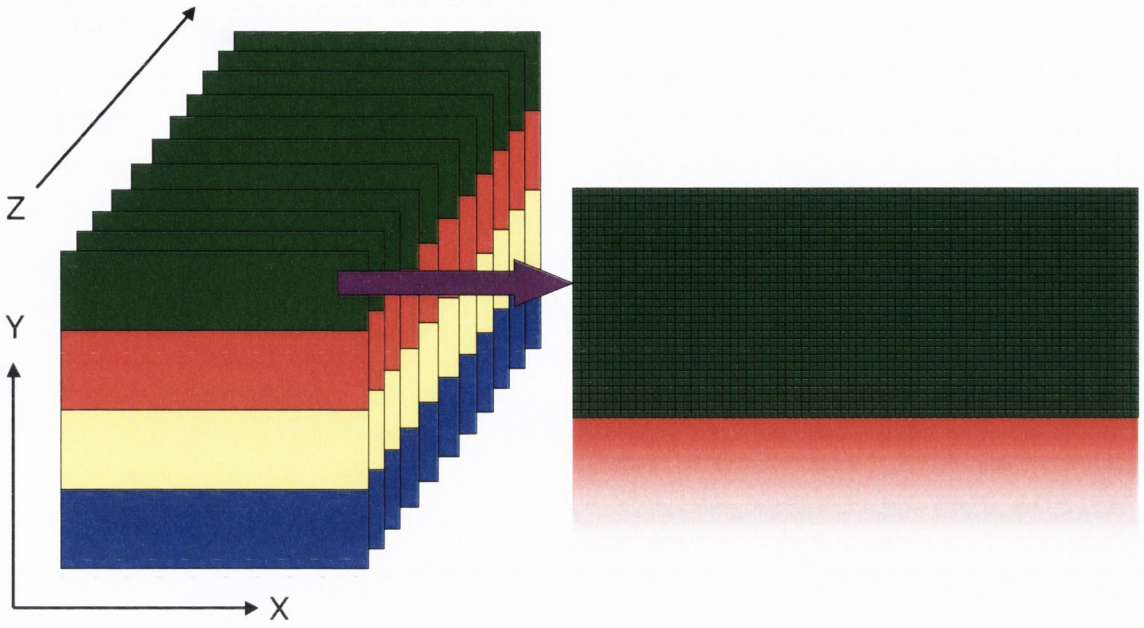


Figure 4.9: Volume slice divided into chunks for distribution to 4 SPUs. Each colour represents a separate SPU.

only way to transfer data to and from the SPU's local store is via DMA transfers, which have a size limit of 16KB. However, the high bandwidth of the EIB and the ability to buffer transfers while still performing computation means that streaming the data becomes an efficient method of processing. Thus, the size of the volume being processed is not limited by SPU storage space.

As a result, a third level of data partitioning is needed in order to enable the SPU to process chunks of any size. If the size of a chunk is bigger than 16KB, it needs to be broken into sub-chunks of below 16KB for transferring. The SPU therefore decides how many complete rows can fit into a single DMA transfer, and iterates through the slices, processing adjacent sub-chunks to create the isosurface.

This has a direct influence on the amount of data replication necessary. Normally, if a chunk fits entirely in one DMA transfer, a volume distributed over n SPUs would need $z \times x \times (n-1)$ pieces of replicated data. But for chunks of over 16KB, the amount of replication needed is $z \times x \times (s-1)$, where s is the number of sub-chunks required.

Whether processing full chunks or sub-chunks, the transfer and processing proce-

dure is the same. See Algorithm 1 for an overview. Each chunk is used twice by the MT algorithm - once as the 8-cell front voxels and once as the 8-cell back voxels. By looping through the slices like this, only one chunk needs to be transferred during any iteration. This keeps data transfer to a minimum, and the buffering can still happen during processing due to the autonomous DMA controller. Processing any reasonably-sized chunk takes longer than transferring it, so usually no time is spent waiting for the buffering to complete.

Algorithm 1 Data transfer

```
1: Transfer chunks from slices 1 and 2
2: for i = 1 to numSPUs do
3:    if i ≤ (numSPUs − 2) then
4:       Start buffering chunk from slice i + 2
5:    end if
6:    Wait for chunks from slices i and i + 1 to finish buffering
7:    Process chunks (see Algorithm 2)
8: end for
```

4.4.5 Processing

For every voxel in a chunk, an 8-cell is created consisting of four voxels each from the two adjacent chunks - two voxels from each of two adjacent rows. Five tetrahedra are constructed from this 8-cell as described in Koide et al. [74]. Each tetrahedron is then processed by MT in order to produce zero, one or two triangles.

Algorithm 2 Processing

```
1: Given two chunks front and back
2: for all rows r in chunk front do
3:    for all voxels v in row r do
4:       Create 8-cell from voxels v and v + 1 from rows r and r + 1 in chunks front
         and back
5:       Decompose 8-cell into five tetrahedra
6:       Process tetrahedra
7:    end for
8: end for
```

Once the tetrahedra have been constructed, they are processed by a regular MT

algorithm such as the one proposed by Gueziec et al. [52].

Taking into account the specialised nature of the SPU hardware, certain optimisations can make a difference in execution speed. The lack of branch prediction means that non-predicted branches should be eliminated wherever possible. We build an interpolation table based on the 16 potential outcomes of tetrahedral evaluation, and perform vertex interpolation and triangle construction according to the results of a lookup in this table. This is similar to the methods used by Pascucci et al. [104] and Reck et al. [113], where isosurface extraction is performed on graphics hardware that has no branching capabilities.

Similarly, the SIMD capabilities of the SPU must be exploited in order to make full use of the capabilities of Cell. This class of acceleration has been the subject of previous research by Hansen and Hinker [54] as applied to Marching Cubes, and much of this work is still relevant to implementing Marching Tetrahedra on Cell.

4.5 Results

In this section we look at the results of testing the methods described in the previous sections. First we look at how isosurface simplification can significantly improve isoextraction performance, before analysing the speeds achieved with the above algorithm for isoextraction on Cell.

4.5.1 Isosurface Simplification

Tests of our isosurface simplification algorithm were done on a 2GHz Pentium 4 with 1GB of RAM and an ATI Radeon 9800 Pro graphics card. Four datasets were tested; three captured by confocal microscopy and one by medical imaging.

Four levels of detail were measured, with level 0 being the original dataset. Each subsequent level of detail represents the number of voxels removed, and consequently the size of the Gaussian filter kernel used to approximate the dataset. Generally, levels of detail beyond these reduce the dataset so much that the resulting volume bears little relation to the original.

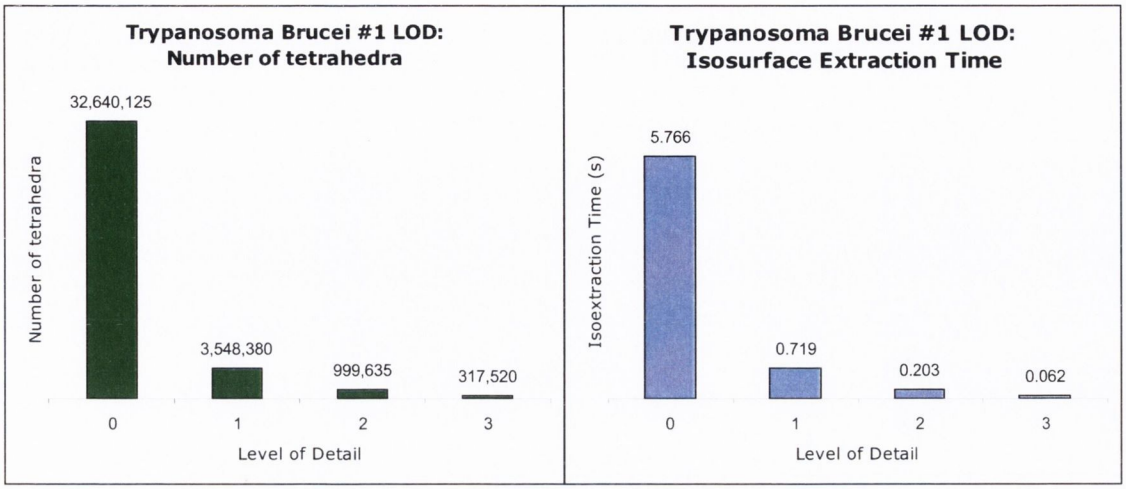
In each case we measure the number of tetrahedra in the resulting dataset, and the time it takes to extract an isosurface from this dataset. Also measured is the

amount of time it takes to perform the simplification - although for any level of detail, the simplification only needs to be performed once. Isosurfaces can then be extracted from the simplified dataset as many times as desired, at the listed speeds. See Figures 4.10 – 4.13 for specific details. For each dataset, two graphs are given; one detailing the number of tetrahedra at each level of detail, and one with the resulting time it takes to extract an isosurface from the reduced dataset. Below the graphs is a table stating the time it takes to reduce the original dataset for a given level of detail.

We can immediately see that even at the first level of detail beyond the original dataset, the number of tetrahedra is significantly reduced by a factor of between eight and ten. This has a direct effect on the time taken to extract an isosurface from the reduced dataset.

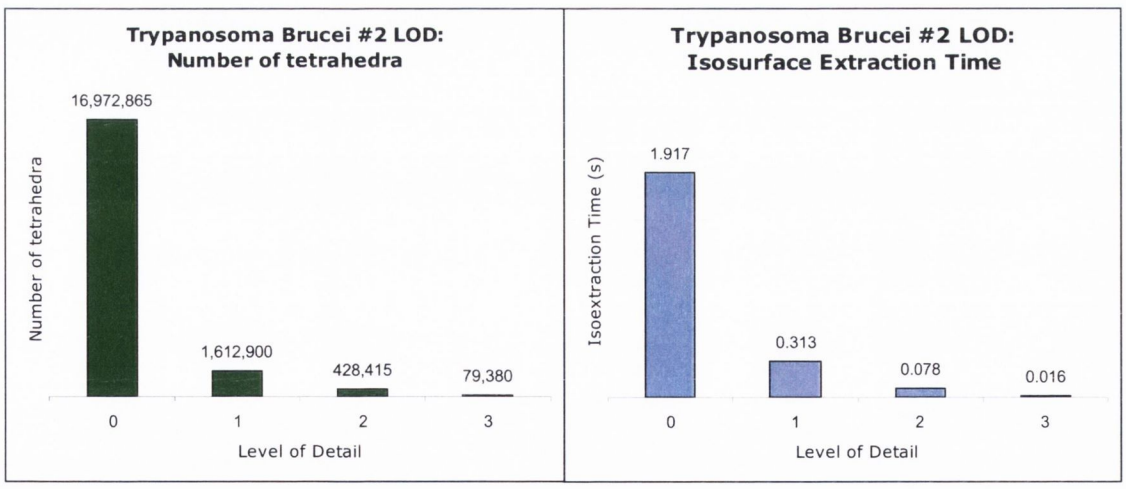
Generally, the datasets that have a high amount of empty space will perform the best in isosurface extraction, since many of the 8-cells can be ignored without needing to be decomposed into tetrahedra or tested for isosurface intersection. When simplification is performed on a dataset with a large amount of high-frequency noise, this noise will either be reduced or removed entirely by the averaging process, depending on the level of simplification. This in turn produces a dataset that will perform better in isosurface extraction that its original counterpart. A simplified dataset that had no noise in the first place will still perform better in isoextraction than the original, but the increase in performance will not be as dramatic and can only be attributed to the reduction in dataset size rather than a reduction of the ratio of 8-cells that need to be processed against those that do not.

Therefore, the datasets that achieve the highest decrease in isoextraction time after simplification are high-frequency, noisy datasets that can be reduced to lower-frequency, less noisy datasets. Conversely, the datasets that gain the least improvements in speed after simplification are those that are already well-formed and with low-frequency surfaces, as the ratio of populated 8-cells beforehand to empty 8-cells after simplification will be lower. In both cases, the first simplification level of detail will always achieve the highest increase in speed, as this is the step most likely to remove the small amounts of noise inherent to the imaging process. The increase in speed and reduction in number of tetrahedra from further simplification are dependent on the nature of the data.



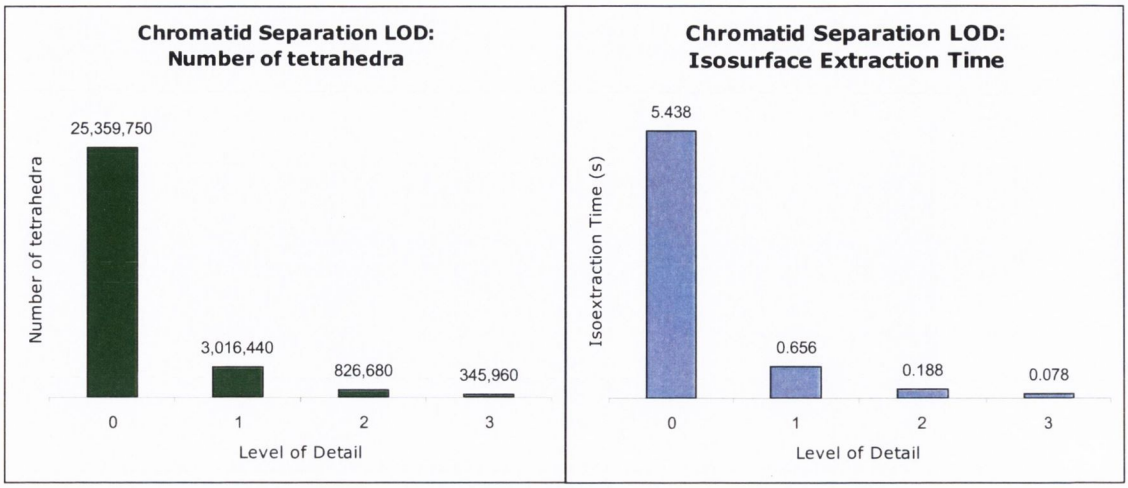
LOD	1	2	3
Simplification Time	1.031s	1.125s	1.078s

Figure 4.10: **Trypanosoma Brucei #1:** This is a $512 \times 512 \times 26$ dataset captured by a confocal fluorescence microscope. It is a scan of the *Trypanosoma Brucei*, a single-celled parasite that is responsible for the disease African Sleeping Sickness. See Figure 4.14 for screenshots.



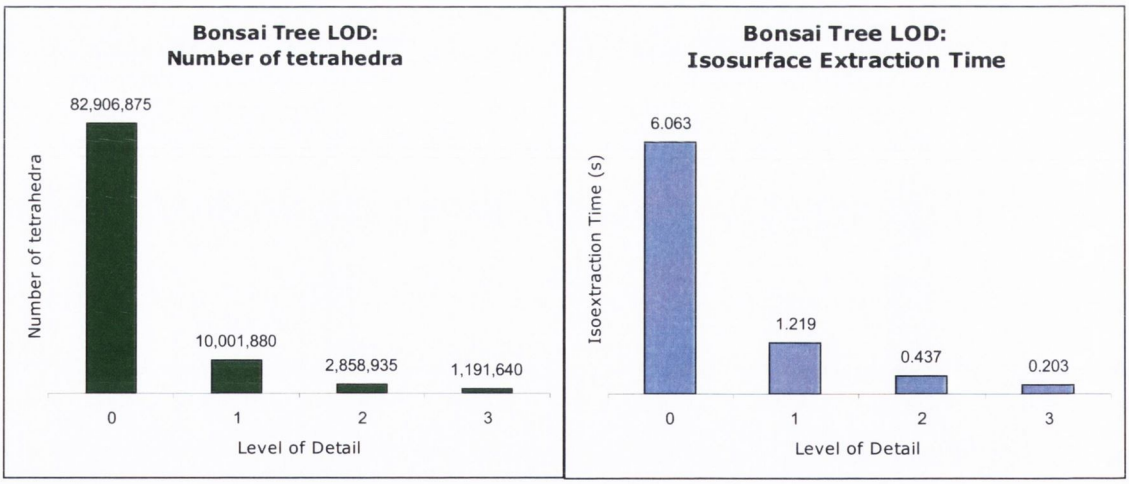
LOD	1	2	3
Simplification Time	0.5s	0.547s	0.437s

Figure 4.11: **Trypanosoma Brucei #2:** A scan of another Trypanosoma Brucei, this time $512 \times 512 \times 14$ in size. See Figure 4.15 for screenshots.



LOD	1	2	3
Simplification Time	0.641s	0.75s	0.828s

Figure 4.12: **Chromatid Separation:** A confocal fluorescence microscope scan of a chromosome separating into chromatids during cellular mitosis, $256 \times 256 \times 79$ in size. See Figure 4.16 for screenshots.



LOD	1	2	3
Simplification Time	1.812s	2.172s	2.453s

Figure 4.13: **Bonsai Tree:** A dataset of a bonsai tree, $256 \times 256 \times 256$ in size, captured by CT. See Figure 4.17 for screenshots.

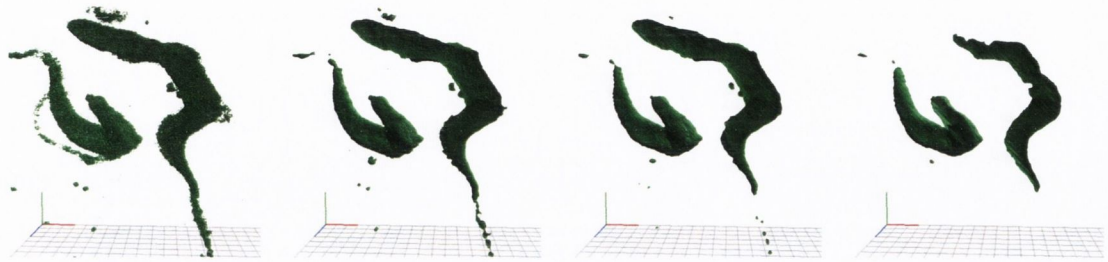


Figure 4.14: **Trypanosoma Brucei #1:** Simplification levels from $n = 0$ to $n = 3$

Figure 4.15: **Trypanosoma Brucei #2:** Simplification levels from $n = 0$ to $n = 3$

Figure 4.16: **Chromatid Separation:** Simplification levels from $n = 0$ to $n = 3$

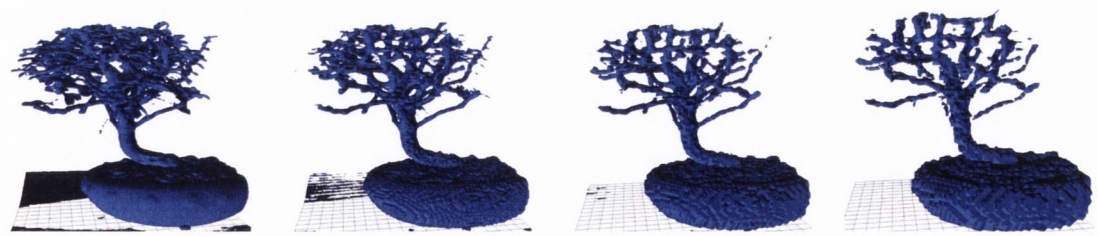


Figure 4.17: **Bonsai Tree:** Simplification levels from $n = 0$ to $n = 3$

4.5.2 Cell Isoextraction

While much of the development of our Cell isoextraction system was done using the Full System Simulator provided with the Cell SDK, we optimised and tested the performance of our technique on an IBM dual Cell "Blade" server located at the IBM T.J. Watson Research Center. This server consists of two 8-SPU Cell processors running at 2.1GHz, with 512MB of XDR DRAM. The dual Cell server contains two Cell processors connected to each other via the Broadband Engine Interface. When an application assigns work to an SPU, an idle SPU is chosen at random from the 16 available SPUs. We also tested an equivalent serial isosurface extraction algorithm for baseline comparison on a 2.0GHz Pentium 4 system with 1GB of RAM.

Performance tests were carried out on a variety of 8-bit datasets, ranging in size from 32^3 to 512^3. The amount of free memory on the test machine precluded volumes larger than these - however, we did provisionally test a 1024^3 volume by reusing the 512^3 dataset 8 times. This result does not take into account the different DMA sizes, cache utilisations and memory access patterns that would occur with a volume of that size. Nonetheless, it is included purely as a processing stress test.

The primary dataset used was a test "spherical shell" 8-bit volume (see Figure 4.18), constructed so that each scalar value is the Euclidian distance from that point to the center of the volume, modulated by 255 to fit inside a byte. This produces multiple shell isosurfaces for any specified isovalue. Also tested was the 256^3 Bonsai tree dataset[1] (see Figure 4.19), and a 512^3 "Head aneurysm" dataset[2].

We can immediately see in Figure 4.20 a demonstration of the improvement in performance achieved using Cell. Even with one SPU, the processing speed of 5.94Mtets/s is higher than the CPU speed of 5.17Mtets/s. This is more than likely due to a combination of the SPU's architecture being more suitable for purely compute-intensive operations, plus the fact that the SPU's clock speed is slightly higher than that of the CPU. At 8 SPUs - one full Cell processor - the processing speed of 47.4Mtets/s is roughly eight times that of the CPU. With both Cells operating at their full capacity for a total of 16 SPUs, the speed rises to 94.7Mtets/s - just under 16 times the CPU's speed.

[1] Courtesy of S. Roettger, VIS, University of Stuttgart

[2] Courtesy of Michael Meißner, Viatronix Inc., USA.

Figure 4.18: The 128^3 spherical shell volume. Each colour represents the isosurface extracted by a different SPU.

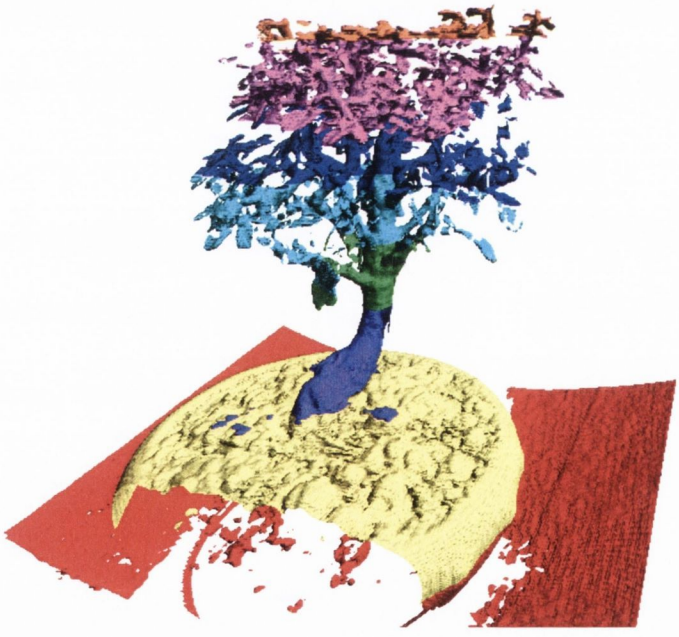


Figure 4.19: The 256^3 Bonsai tree volume. Again, the distribution of chunks is depicted by separate colours for each SPU.

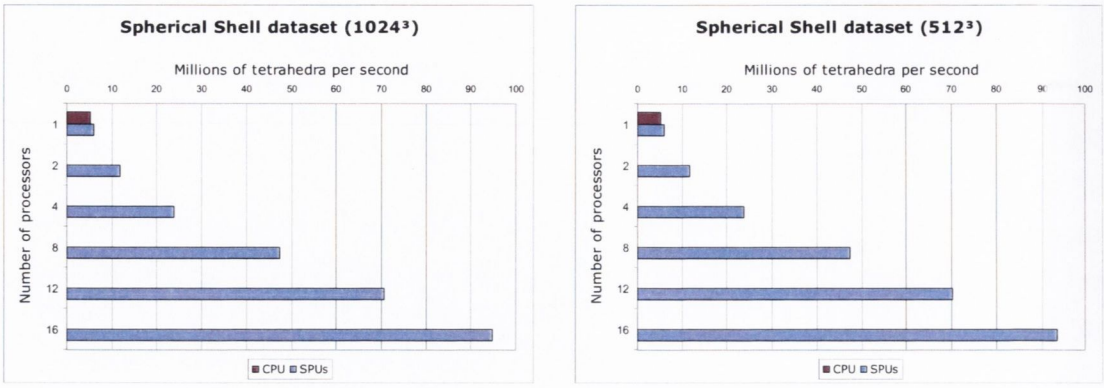


Figure 4.20: Test results for the 1024^3 and 512^3 spherical shell volumes

Similarly, Figure 4.21 demonstrates that the Cell continues to show marked improvement over the CPU for smaller volumes. However, looking carefully at the results of the 128^3 volume, we can see that this improvement starts to slow down as the number of SPUs involved grows. This is due to the size of DMA used to transfer data between the cores. At 16 SPUs, dividing each 128×128 slice equally gives 8 rows to one SPU and 9 each to the others. This corresponds to DMA sizes of 1024 bytes and 1152 bytes respectively. At this size, the efficiency of the DMA transfer starts to drop due to the cost of setting up and initiating the transfer.

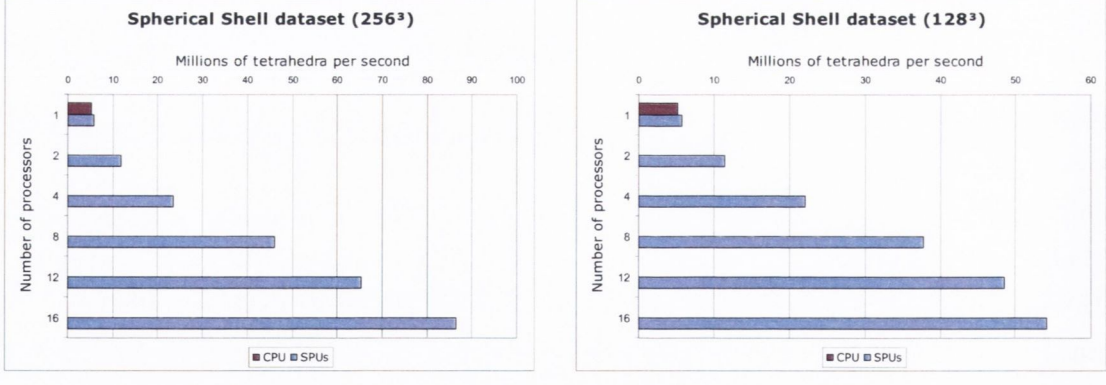


Figure 4.21: Test results for the 256^3 and 128^3 spherical shell volumes

As we reach the smaller volumes of 64^3 and 32^3 (see Figure 4.22), this trend starts to become more pronounced. As DMA sizes drop further, so too does the advantage of adding more processors, until we reach a turning point at 8 SPUs for

the 64^3 volume and 2 SPUs for the 32^3. Both of these points mark DMA sizes of approximately 512 bytes per transfer. Below that point, further distribution of work is actually detrimental to the overall processing speed of the system.

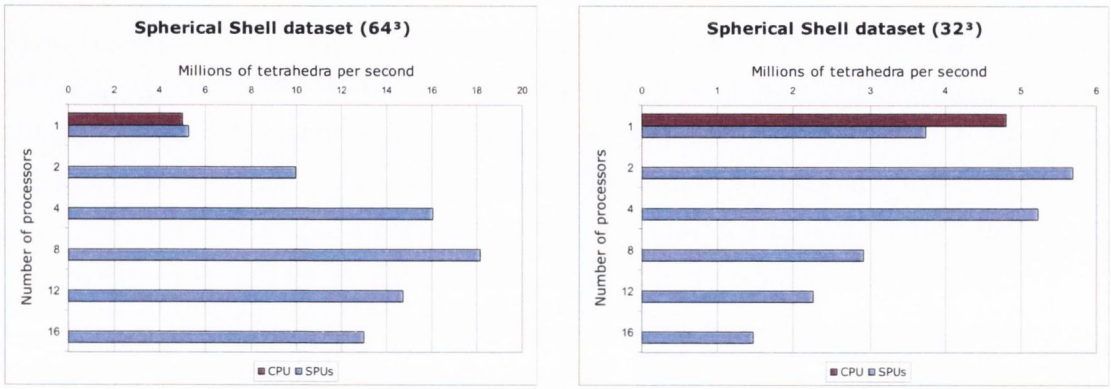


Figure 4.22: Test results for the 64^3 and 32^3 spherical shell volumes

As we turn to real datasets, the maximum processing speed possible increases to over 100 million tetrahedra per second for the Bonsai and Head Aneurysm datasets (see Figure 4.23). The reason for this is that these volumes are composed of more empty space than the spherical shell - if all 8 voxels of an 8-cell have an isovalue of 0, the whole cell can be skipped safely without being tested thoroughly for intersections, leading to an increase in tetrahedron throughput.

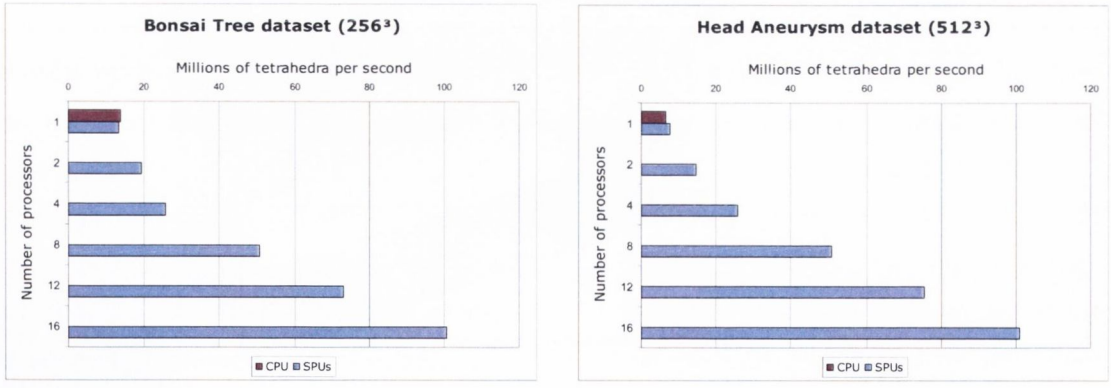


Figure 4.23: Test results for the Bonsai Tree and Head Aneurysm volumes

It should be noted that the quoted Cell clock speed of 2.1Ghz on our test system is well below the possible limits of the hardware, and is due to the system being an

internal IBM test platform. The Cell processors in use in the Playstation 3 console are clocked at a considerably higher 3.2GHz. Cell has the potential to reach 5GHz, albeit at a temperature that would exceed commercial safety limits. Reducing power consumption could lower this temperature and thus increase clock speeds - this has been identified by IBM as a future direction.

Furthermore, our results also compare extremely favorably to quoted GPU speeds of 9 million tetrahedra per second [113] via a vertex processor implementation. However, the comparison is only a superficial one, as current GPU implementations take advantage of spatial acceleration structures which our method currently does not. Indeed, Kipfer et al.'s processing speeds of 69.4m tetrahedra per second in the fragment processor [69] are dependent on the use of interval trees. GPU speeds also include rendering time, whereas our results only measure processing time; an entire processing and rendering pipeline would need to be implemented on both architectures for a fair comparison to be made. Additionally, GPU implementations are subject to the condition that the dataset fits entirely into relatively limited video memory.

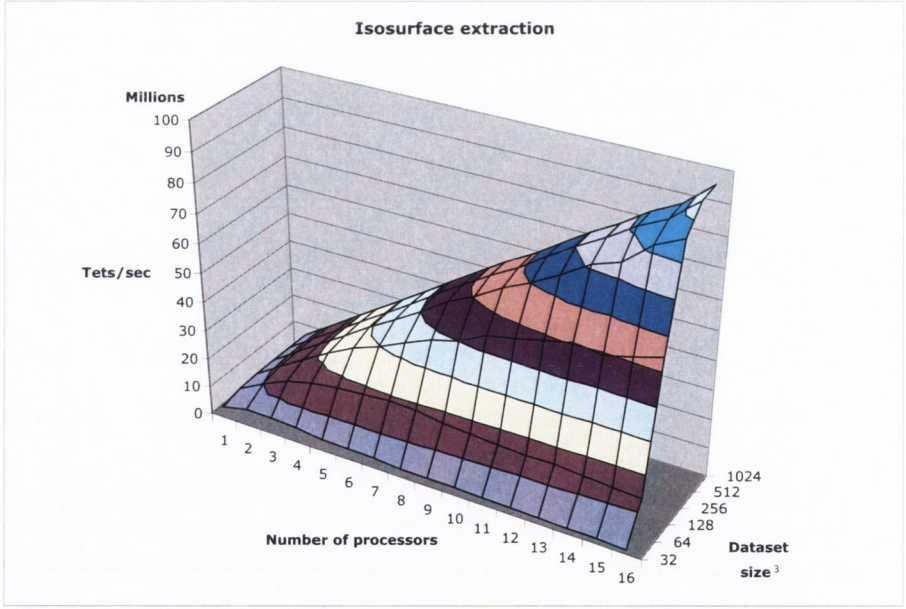


Figure 4.24: Comparison of isosurface extractions speeds: dataset size vs. number of processors vs. Mtets/sec.

Finally, in Figure 4.24 we can see a 3d plotted surface of the isosurface extraction speeds taken from the graphs in Figures 4.20 – 4.22. From this we can see a general trend that - as expected - the more processors involved in surface extraction, the faster the processing speed. The obvious exception from this the 32^3 dataset which suffers from excessive DMA transfer overhead as previously mentioned. Otherwise it becomes obvious from observing this graph that as dataset sizes increase and this overhead becomes insignificant, a more linear increase in processing speed can be expected.

4.6 Cluster Implementation

This section examines the advantages and disadvantages of applying the above algorithms to our custom Cluster architecture. In each case, the cluster offers the potential for greatly accelerating the algorithm compared to single-processor implementations.

4.6.1 Volume Visualisation

The cluster is a natural platform for performing hardware-accelerated volume visualisation. The Vizard II [88] has already demonstrated the applicability of FPGA hardware for performing ray casting and volume visualisation. By storing the original dataset in shared memory and assigning a section of the final rendered image to each node in much the same way as sort-first rendering, volume rendering speed can be increased by adding more nodes to the cluster. In this respect, extremely large datasets can be processed by distributing the storage across all memory banks available across the cluster. Altering the viewport will cause nodes to access different portions of this distributed volume according to the ray casting algorithm, but the SCI interconnect would minimize the penalties of extra latency introduced by accessing remote memory. The global address space implemented by the cluster would cause this switch from local memory to remote memory to be transparent to the application, simplifying implementation.

Additionally, if enough storage space was available to store the dataset multiple times, it would be possible to replicate portions of the dataset in multiple memory

banks. By distributing the dataset evenly across the cluster and localising the portions of the volume in memory local to the node processing that portion, access speeds would be increased compared to remote accesses to the same data. This would also increasing cache utilisation and reduce access times by eliminating stalls that might be introduced by multiple nodes accessing the same memory location in order to process the same portion of the volume. Compared to a similar Cell-based implementation of ray casting, having the entire global memory of the cluster available to each node's processor (as opposed to the limited local store of the SPUs) at high speeds with low latency provides a significant advantage. This is because the integration of samples over the ray from an arbitrary viewpoint requires accessing voxels in a non-linear way that is not amenable to the SPU's method of transferring chunks of data by DMA to the local store.

The existing GPU-based acceleration methods described earlier in this chapter can still be implemented on each node, performing texture-based direct volume rendering of the dataset. However this would preclude the use of each node's available FPGA, under-utilising the potential processing power available to the application. Instead, a hybrid approach can be employed by performing the bulk of the processing on the FPGA via ray-casting, and then using the GPUs for image post-processing and image-space lighting. Further investigation is needed to evaluate the potential of this approach.

4.6.2 Volume Simplification

Again, the locality of data necessary for the volume simplification algorithm described in Section 4.3 would translate very appropriately to a cluster implementation. The gaussian filter employed restricts data references to a small area inside the volume, allowing for a partitioning of the dataset into discrete sections with minimal overlap. Each node can then process and produce a simplified version of its partition, allowing efficient parallelisation and acceleration of the algorithm. This is comparable to the Cell implementation of isosurface extraction.

Each partition need not necessarily be programmatically partitioned and read into each node's local memory, but rather the partitioning can be done logically and the actual data can still reside distributed across the shared memory. Compared to

a similar implementation on Cell, the overhead of setting up DMAs and transferring all the volume data to the SPUs for a relatively small amount of processing is an inefficient use of the processing power, despite the suitability of the SPUs for performing the filtering on the data. This is not to say that Cell would not perform well, but using the distributed shared memory would avoid this situation altogether on the cluster.

4.6.3 Isosurface Extraction

However, performing volumetric simplification on the cluster may not be necessary at all if its application is solely for increasing surface extraction speeds, and not for Level of Detail or noise reduction purposes. Instead we can use the cluster to increase the speed of the isosurface extraction itself.

The algorithm described in Section 4.4 for performing isosurface extraction on Cell would transfer very well directly onto the cluster, with a few additional benefits. Like volume simplification, the logical partitioning of data for processing would be facilitated by distributed shared memory, avoiding memory copies while keeping access times low via the SCI fabric.

The most significant advantage of the cluster over a Cell implementation is the rendering capabilities of each node. Whereas the Cell needs to transfer the generated geometry back from the SPUs to main memory for rendering, a cluster node can feed it directly to its associated GPU for immediate rendering. The resultant image fragments could then be composited over the cluster in a sort-last manner into a final rendered isosurface.

Alternatively, the geometry could be inserted directly into a shared command buffer for even distribution of rendering work. Additionally, having a GPU on each node results in more raw rendering power than a single graphics card associated with a Cell processor. Multiple graphics cards could possibly be attached to a network of Cell processors, but at the expense of the general processing power needed for coordination, which detracts from the power available to the application itself. This contrasts to the cluster which is designed from the ground up for distributed rendering and has dedicated hardware for managing the extra complexity layer.

Chapter 5

Entertainment

For entertainment applications such as games, the user is often an active participant in a virtual world. In order to convey this participation, applications require not only a high and steady frame rate, but also a believable and consistent representation of the world in which their avatar exists. As processing power increases and simulation becomes more sophisticated, users expect these virtual worlds to resemble the real world more and more closely. Similarly, as development studios compete to produce superior visual effects and budgets for top-rated games escalate (the eagerly anticipated title *Spore* due in 2007 is estimated to have a budget of approximately $30 million [114]), they push the hardware to its limits to improve the realism and immersiveness of their games.

In this chapter we look at the use of commodity parallel hardware for accelerating the graphics algorithms used in entertainment applications - specifically, image-based crowd rendering. Section 5.1 gives a brief overview of the use of GPUs in games before concentrating on crowd rendering. Section 5.2 describes research on the development of a single-pass crowd rendering algorithm, optimised to take advantage of the programmable graphics hardware pipeline to render humans with fully customisable lighting and texture-based variation. Section 5.3 presents results based upon the use of this algorithm in a full crowd rendering system.

5.1 Parallel Hardware in Entertainment Applications

As chips get smaller and sequential architectures reach their limit in terms of silicon size and heat output, the real benefit of parallel architectures becomes apparent. Games are composed of many individual tasks, including but not limited to rendering. Artificial intelligence is required to give the virtual world some life, and to provide the player with lifelike allies and adversaries. A physics engine gives the world a sense of solidity and authenticity; pushing an object and seeing it fall in a realistic way makes the object seem that much more real to the player. As broadband internet access becomes widespread, more and more games are focusing on the potential of multiplayer interaction - this in turn requires sophisticated prediction code to compensate for the latency introduced by the network. All of these tasks can be run in parallel with minimal interaction. For example, collision detection for a tumbling wall of bricks can be computed at the same time as the path finding algorithm of a virtual agent.

It makes sense therefore that in order to maximise the amount of work capable of being done at any one time, all three of the major next generation games consoles contain multicore parallel processors. Sony's Playstation 3 contains the Cell processor, as described in Section 2.4. Microsoft's Xbox 360 contains the Xenon Processor produced by IBM, which in itself contains 3 dual-threaded PowerPC cores allowing for 6 simultaneous hardware threads to be run in parallel. Finally the Nintendo Wii's Broadway CPU is rumoured to contain either 1 or 2 PowerPC-based dual-core processors (again produced by IBM) - Nintendo has not released final specifications at the time of writing.

5.1.1 Graphics Hardware in Games

The most obvious and widespread application of parallel hardware to the entertainment industry is graphics hardware. Since the first 3D accelerator card came out, the games industry has always been a driving force behind the advancement of GPUs. Similarly, advances in graphics hardware have allowed many new techniques to be pioneered by games developers. In all of today's 3D games, a GPU of a certain

level is a prerequisite in order for the game to run.

Many of the advanced effects that give games their unique look and feel are the results of GPU-based algorithms. In recent years, one of the biggest improvements in visual quality and realism came from a technique called *normal mapping* [21, 20]. This is an effect which allows the interaction of light on an object's surface to be independent of its tessellation by encoding the surface's normal in a texture map. When evaluating the lighting equation at each pixel in the fragment processor, a shader looks up the normal vector for that point encoded in the normal map, and uses this instead of the actual interpolated surface normal. The effect is dramatic - models of drastically reduced polygon count can use high resolution textures and normal maps and still produce virtually the same image that a very highly tesselated model would.

Figure 5.1 shows an example from Epic Games' forthcoming *Unreal 3* engine. Artists first create an extremely detailed version of the model with 2 million polygons, as seen in Figure 5.1(a). This model is then reduced to a mesh of 5,287 polygons, which will be the version used in the game (see Figure 5.1(b)). A ray-casting engine is then used to cast rays out from the low resolution version to the high resolution one. The normal is determined at the intersection point, and stored in the normal map for that point on the low resolution model. This normal map is then used for all lighting calculations for that model in-game, resulting in a final rendering as seen in Figure 5.1(c). Upon closer inspection of the outline, the low resolution basis of the rendering is revealed. However, the effect is still extremely convincing, even more so when animated. The underlying techniques of normal mapping have spawned a whole class of texture-space effects, such as relief mapping [100] and parallax mapping [64].

5.1.2 Crowd Rendering

A major part of any virtual world is the people inhabiting it. Human and crowd rendering is both an important and difficult area. The importance of a crowd is most conspicuous by its absence, when a user would expect to see crowds of people such as in an urban environment. All of the recent titles from Rockstar's bestselling and notorious *Grand Theft Auto* series suffered from this problem (see Figure 5.2),

(a) High-resolution model (b) Low-resolution model

(c) Final in-game image

Figure 5.1: High-resolution, low-resolution and final rendered image of a model from the *Unreal 3* engine (©Epic Games)

where streets were modeled in great detail and filled with a variety of vehicles, but the number of human pedestrians visible at any time was limited to a handful. This gave the virtual city a somewhat deserted feel.

On the other hand, the difficulty of crowd rendering comes from the number of individual humans that make up a crowd. A large crowd requires that a large number of humans be rendered and animated, which can consume a lot of simulation and rendering time. The more varied the crowd is, the more polygons and textures must be stored in video memory. This can be a problem, as crowds are usually

Figure 5.2: Deserted streets in Grand Theft Auto: San Andreas (©Rockstar Games)

a background element in many entertainment applications and not the main focus of the scene. Therefore the proportion of the total polygon and texture memory budget allocated to the crowd is less.

5.1.3 Reducing Rendering Work

There are a number of methods for lowering the rendering cost associated with polygonal crowds. Level Of Detail (LOD) is a method of reducing the polygonal detail of a model according to certain criteria, usually distance from camera or importance in a scene [82]. O'Sullivan et al. [102] describe a framework for human and crowd LOD, also incorporating behavioral levels of detail. The main advantage of LOD is the reduction of the amount of polygonal data required to be drawn,

therefore reducing the total primitive count of the scene and lessening the burden on the GPU.

However, there is another hidden cost associated with rendering large amounts of objects that does not relate to the polygonal complexity of the object. Assuming an object resident in video memory (as opposed to immediate mode rendering where the geometry is specified every frame and transmitted across the graphics bus), an API function call needs to be submitted for every object drawn. If this draw call invalidates the current rendering state (for example by changing texture, changing a shader constant etc.), then the corresponding state changes need to be implemented in hardware before the rendering can proceed. This is particularly true for high level shading languages such as GLSL that have to map variable names to hardware registers. This blocks the pipeline and interrupts the organisation that the driver does in order to rearrange the primitives in optimal format for submission to the hardware. The extra API calls needed to change the state can easily make the application become CPU limited, resulting in performance degradation that scales with the number of objects being drawn. This can be offset by sorting the rendered objects according to their state, and rendering all objects of the same state together. However, the degree of success of this process depends on how many objects are to be rendered altogether, and how many share the same state.

To alleviate this effect, recent GPUs have support for 'mesh instancing' [34, 49]. This enables multiple objects with the same geometry to be drawn with a single API call, while still allowing properties such as the transformation matrix and material properties to be changed per-instance. This could be useful for rendering small crowds of identical humans. However, the GPU is still required to transform and rasterise every object in the scene, which can be extremely costly for many highly detailed objects. If the application is not CPU-limited by API calls, instancing will not bring any advantages.

5.1.4 A Further Level of Detail: Impostors

There is only so much simplification a model can undergo before it bears no resemblance to the model it represents. Therefore another level of detail is needed below that of the simplest geometric representation.

In 1995, Maciel et al. [84] first put forward the idea of using pre-generated images of geometric models as the lowest level of detail in an LOD system. These *planar impostors* provide a shortcut for rendering geometric models - instead of needing to render the entire model, it is rendered to a texture in an offline process and then used interactively for the rendering price of a single texture-mapped quad. Given the large increases in frame rates achieved by using impostors, a large amount of research has been done, leading to advances in the area of Image Based Rendering (IBR). An overview of these methods is given by Jeschke et al. [63].

Impostors have been used for many years in the computer games industry. They represent a cheap alternative for rendering complex geometric objects such as trees and plants. First-generation 3D games such as iD software's *Wolfenstein 3D* and *Doom*, developed for systems without the advantages of dedicated GPUs or even moderately fast CPUs, made extensive use of impostors for every game object except the level architecture. This led to the description of the technology as "two-and-a-half-D".

Impostors are a good solution for reducing the rendering costs associated with the large numbers of humans present in a crowd. Tecchia et al. [130] propose using one impostor per human, pre-generating multiple images for each viewpoint that the impostor will be viewed from. They choose 16*8 images - 16 viewpoints around the Y-axis for each of 8 elevations around the X-axis. These images are mirrored to produce a total of 32 viewpoints for each elevation, with a difference of 11.25 degrees between them (see Figure 5.3). Then during simulation a billboard facing the camera is rendered for every human in a crowd, textured with the viewpoint image appropriate for the angle at which that human is being viewed. A 'popping' artefact can sometimes be seen when changing from one viewpoint image to another, but at the distances that crowds are normally viewed at, this does not become a severe problem.

Rendering static images of humans from any angle is not in itself very useful. Especially in crowds, humans are constantly animated and moving. Tecchia et al. address this problem by creating 10 different animation frames and encoding these as impostors.

In contrast, Aubel et al. [8] propose an impostor-based technique for accelerating human rendering, but use dynamically generated impostors instead of a prepro-

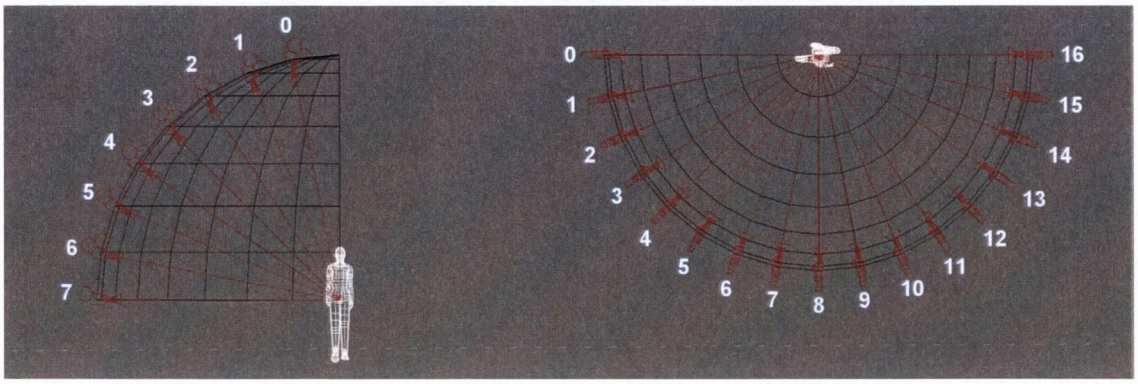


Figure 5.3: The 8 elevation and 17 rotation viewpoints captured to form an impostor.

cessing step. This has the advantage of consuming far less texture memory than pre-generated impostors, at the expense of the extra rendering work that needs to be done to update each impostor.

Any impostor technique breaks down when the impostor approaches the camera, and the two-dimensional basis of the technique becomes apparent in two ways; the popping artefacts become extremely noticeable due to the limited number of viewpoints, and the impostor images themselves become overly pixellated and the lack of fine detail becomes obvious. Both of these problems are a function of the number of impostor viewpoints and the resolution of the impostor images, but they will eventually affect any impostor-based solution. In our I3D paper [29], we overcome this by using a hybrid crowd representation of both geometric models and impostors called *geopostors*. Geometric representations are used for humans close to the camera, switching to impostors for those further away. To avoid popping, we switch imperceptibly between the two by using a 'pixel-to-texel' ratio. This ratio is used to determine when the size of an impostor image's rasterised texel is larger than a pixel. In this instance, aliasing will occur and so the geometric version needs to be used.

The main focus of this paper was the hybrid representation and switching between geometry and impostors. This was enabled by employing programmable graphics hardware to match the impostor's rendering to that of its geometric counterpart, as detailed in the contributions of this thesis in Section 5.2.

5.1.5 Hardware Implications of Impostor Usage

In terms of hardware usage, the increase in frame rates achieved by using impostors is due to the reduced load on both the vertex processor (having to transform just 4 vertices of a quad as opposed to every vertex in the geometric model) and the fragment processor only having to texture map a single quad. Additionally, being almost purely the domain of the fragment processor, impostors will continue to provide improvements in rendering speed as the number of pixel pipelines increase and more pixels can be processed in parallel.

As mentioned earlier in this section, it is best to minimise state changes during rendering a frame in order to take full advantage of the graphics hardware pipeline and avoid any stalls. For this reason, the number of texture changes should be kept to a minimum. Tecchia et al. take this into account by packing each 64×64 impostor viewpoint for a given animation frame into a single 1024×512 texture. We improve upon this by observing that for any given camera placement, the overall number of elevations used by all impostors in a scene is limited to only a few. We therefore sort the textures not by animation frame, but according to elevation. This keeps frame rates more constant and reduces the amount of *texture thrashing* that occurs when the total size of textures used in a scene exceeds the size of available video memory. When video memory is low, the driver swaps texture(s) out to system memory over the AGP bus, using a least recently used (LRU) algorithm. However, if all of the textures are needed in every frame, this swapping will happen continuously and the application will stutter as the textures are swapped in and out. Since all animation frames are invariably used over a maximum of ten frames, every animation texture will be needed by the application. By keeping only the subset of elevation textures that are used, the size of textures kept in video memory is much lower while the camera stays at a constant height.

However, there are some drawbacks to using impostors as substitutes for fully geometric models. The amount of texture memory required to represent a human from every required viewpoint is not inconsiderable and, depending on the application's video memory budget devoted to the crowd, this can be prohibitive. These effects can be somewhat ameliorated by texture compression, but still represent a major barrier for the widespread use of impostors. Similar to the widening gap

between CPU speed and memory latency, GPUs are going to continue increasing in performance faster than the memory that they access, which will lead to memory assets becoming more precious as requirements increase.

Another related bottleneck of impostor rendering is the use of hardware memory bandwidth. Being representations of convex and concave polygonal models, the majority of impostors contain large areas of transparency. In order to perform blending or alpha testing, accesses to the frame buffer must be made from the fragment processor, which can be costly in terms of both memory bandwidth and *fill rate* due to *overdraw*. Fill rate is the rate at which a GPU is capable of rasterising polygons into the frame buffer, usually measured in millions of pixels per second. Overdraw occurs when a polygon that has already been rendered to the frame buffer is drawn over by another polygon, hence the fill rate for the first polygon was wasted. Thus the more overdraw occurring, the less fill rate is available for the entire frame. As a result, the widespread use of impostors in a scene can lead to a memory bandwidth or fill rate bottleneck in the rendering system, thus limiting performance.

5.1.6 Introducing Variation

Even with individually animated impostors, a sea of unlit identical humans does not make a very convincing crowd (for an example see Figure 5.4). Variations in clothing, and dynamic lighting, are essential to make the 2D illusion of impostor crowd representations a convincing substitute for the corresponding 3D geometric models.

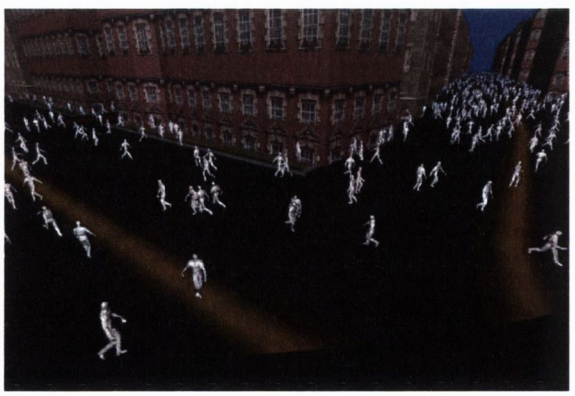


Figure 5.4: Unconvincing crowds in [130].

In an extension of their previous work, Tecchia et al. [131] introduce both colour variation and lighting to their crowd rendering system. They perform colour variation by splitting the impostor into regions encoded in the alpha channel. For example, one region might be the skin of the human, another region the trousers, etc. A multi-pass rendering algorithm then draws every impostor in multiple iterations. One pass is used for every region that is to be rendered in a different colour. At each pass, the alpha test function is set to discard every fragment that does not match the alpha value of the current region. The colour is then set, and the region is rendered with that colour. This method lets a single impostor be rendered many different times using differently coloured regions in each case, giving the impression of many different and varied humans instead of a single human model 'cloned' across the scene. This level of crowd variation is often overlooked in games. For example, Electronic Arts' next-generation console title *Fight Night Round 3* contains crowds that are obviously composed of the same models replicated many times (see Figure 5.5). A simple alteration of clothing colours would give the crowd a much more varied and interesting look.

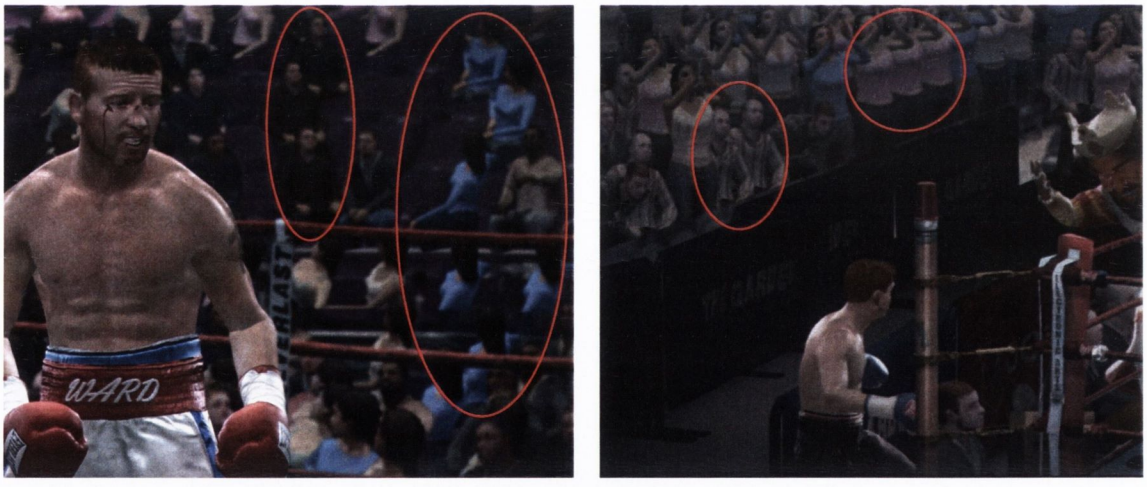


Figure 5.5: Scenes from *Fight Night Round 3* (©Electronic Arts). Note the lack of crowd variation in the indicated areas.

De Heras Ciechomski, Ulicny et al. [25, 137] take the alternative route of using various levels of detail to produce crowds composed of low-polygon models. To introduce variation, they reuse the same geometric model but apply different tex-

tures, colours and scaling factors in order to give the appearance of a diverse crowd of people. They apply this model to a scene involving a crowded amphitheatre in ancient Rome as well as a modern urban setting, rendering hundreds of animated and varied humans at interactive frame rates.

5.1.7 Dynamic Impostor Lighting

Even more important than introducing colour variation into a crowd scene is the lighting of the impostors, especially when using a hybrid impostor/geometry system. A complete lack of lighting would be immediately obvious and is not sufficient. Pre-baking lighting into the impostor image is a better solution, but fails under various circumstances; a human will be lit identically no matter whether it is daytime or nighttime, they will be lit from the same direction regardless of their orientation, and most importantly their lighting will not match the lighting of any geometric representations to which they should switch in a hybrid system. The polygonal versions of the humans will invariably be lit dynamically with whatever lighting equation is deemed suitable; for example Blinn-Phong for the regular OpenGL fixed function pipeline. Upon switching to an impostor representation, any changes in lighting will become painfully obvious and seen as a visible pop. This detracts from the visual quality of the overall scene. Therefore if the switch from impostor to geometry and vice versa is to be seamless, the lighting of the impostors must match precisely.

In [132], Tecchia et al. address the dynamic lighting issue by using a multi-pass algorithm. An additional texture is generated for every impostor image in an extra step of the pre-generation procedure. This image contains the normals of its respective impostor. The x,y,z of the normal at every pixel is *range compressed* and encoded into RGB channels. Presuming a well-formed normal in the range $[-1, 1]$, the mapping from normal to 8-bit RGB value is:

$$R \rightarrow ((x + 1) \times 0.5) \times 255$$
$$G \rightarrow ((y + 1) \times 0.5) \times 255$$
$$B \rightarrow ((z + 1) \times 0.5) \times 255$$

They use the OpenGL 1.3 DOT3_RGB_ARB extension to perform a dot product of the normal at every pixel and a per-impostor light vector, evaluating a lighting

equation approximately equal to that of the OpenGL fixed function pipeline. Each component of the equation is rendered in a separate pass, which is accumulated in the frame buffer and combined with the multi-pass colour variation method to give a coloured, dynamically lit final impostor. This is done with a total of 8 passes; 5 passes for lighting and an additional 3 to add colour variation.

An example normal map of a single frame of animation from every viewpoint can be seen in Figure 5.6. Here the individual impostor images have been tightly packed to fit into the smallest texture possible, reducing video memory consumption.

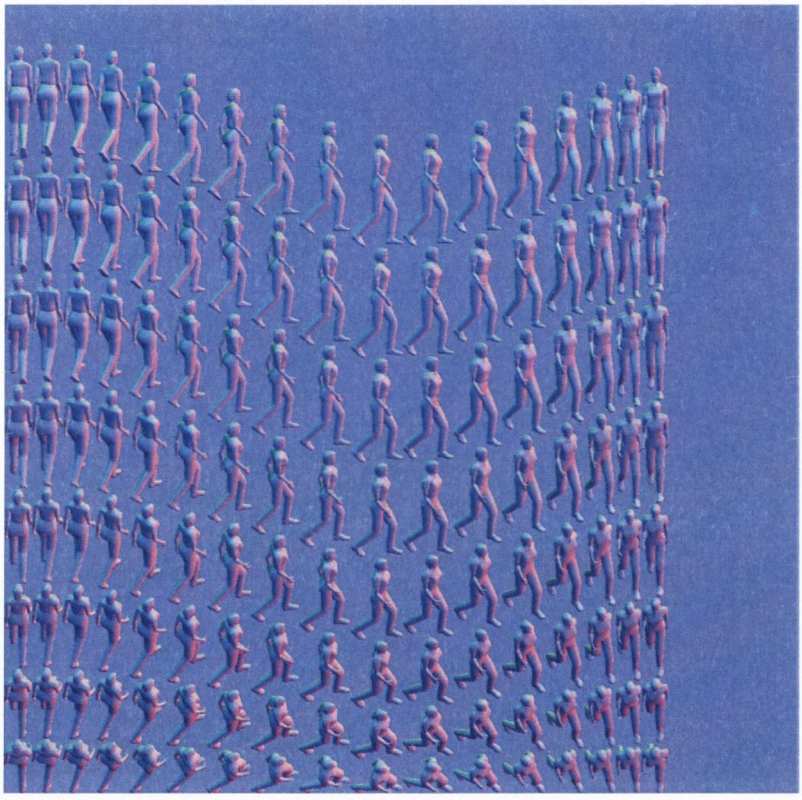


Figure 5.6: A 1024×1024 impostor normal map for a single frame of animation

5.2 Accelerating Crowd Rendering

When dealing with the large numbers of individuals that comprise a crowd of humans, any increase in the rendering performance of a single unit will lead to a larger

increase of performance for the crowd as a whole, allowing more detailed simulation or an increase in the number of impostors for the same computational cost. Therefore close attention must be paid to maximising use of the hardware architecture upon which the crowd is being rendered.

The methods outlined by Tecchia et al. as discussed in Sections 5.1.6 and 5.1.7 perform well on previous generation graphics cards, but suffer from a number of deficiencies when it comes to implementation on the latest parallel hardware. This section outlines these problems and details solutions that overcome each in order to improve the rendering acceleration of every impostor, and by extension the entire crowd.

5.2.1 Disadvantages of Multi-pass Algorithms

The main weakness of the algorithms used in [131] and [132] are their reliance on multi-pass techniques. Multi-pass techniques can be extremely beneficial for algorithms which cannot be composited into one pass, and require the accumulation of renderings - using the output of one pass as an input to the next pass. However, the primary disadvantages of multi-pass algorithms compared to single-pass are as follows:

More driver overhead: For every pass, the driver is going to spend time in the API calls necessary to set the state for the next pass. This state may be different from the state of the last pass - a different texture bound to the texture unit, different material properties, different lighting or different texture filtering. This means that the current state is invalidated and the pipeline must be flushed. The acceleration achieved by highly parallel GPUs is dependent on having many vertices and pixels being processed simultaneously, so flushing the pipeline is extremely undesirable. Having this sort of state thrashing happen for every pass of every impostor for every frame can result in severely degraded performance and a CPU bottleneck in the driver.

More vertex transformation: Every time an impostor is rendered for a pass in the multi-pass algorithm, the four vertices of that impostor must be transformed. This means that for an 8-pass algorithm, 32 vertices are being trans-

formed for every impostor. This represents a major increase in the amount of work that must be done by the vertex processor, possibly introducing another application bottleneck.

More fill rate: This is possibly the most damaging aspect of multi-pass rendering in the case of impostors. Impostors are already heavily reliant on the fill rate speed of the GPU because they are an entirely image-based method. All the blending and alpha testing required to render a non-opaque polygon are amplified by the multi-pass method of culling any fragment except those with a particular alpha value. As in the impact of extra vertex transformation, an 8-pass algorithm will cause eight times the amount of fragment processing to occur. In addition, redrawing the same quad many times results in much overdraw, further reducing the fill rate of the overall frame.

To overcome these problems, we propose a generalised single-pass solution implemented on programmable hardware that allows dynamic lighting and impostor colour variation to a greater extent than previously possible. This has also been published in ACM Transaction on Graphics, at I3D, and presented at SIGGRAPH 2005.

5.2.2 Dynamic Impostor Lighting

While the use of the OpenGL DOT3_RGB_ARB extension allows Tecchia et al. to compute a per-pixel dot product, the coefficients of the dot product they use are the normal vector (encoded in the normal map) and per-impostor light and half-angle vectors as per the Blinn-Phong lighting equation. The reason they use per-impostor instead of per-pixel vectors is to avoid the overwhelming computation of computing both vectors for every pixel, which could not be performed in hardware using the fixed function pipeline. While the result is correct for a light source set at an infinite distance (as each per-pixel vector for an infinite light source would be parallel anyway), the invariance of the light vectors over the impostor leads to an inaccuracy that increases as the light gets closer to the impostor. With a light source directly beside the impostor, a lack of accurate light vectors would lead to incorrect lighting that would differ significantly from the lighting of the impostor's

geometric counterpart.

An additional restriction of using the fixed function pipeline, even with the per-pixel capabilities of DOT3_RGB_ARB, is that the lighting equation used is restricted to purely Blinn-Phong. While this is a perfectly acceptable model for the diffuse lighting of human impostors, it precludes the possibility of using an alternative lighting equation such as anisotropic lighting (for a human wearing material such as velvet, for example). This is true for both the impostor and the geometric version, as the lighting equations must match in order to avoid artefacts when switching from one representation to another.

To avoid these problems, we implement both geometric and impostor lighting using programmable hardware. We use a simplified version of the OpenGL lighting equation which removes the specular, emission and shininess contributions in order to reduce the amount of computation needed. The final impostor shading equation is:

$$Pixel_{RGB} = (((\overrightarrow{L} \cdot \overrightarrow{N}) \times Region_{RGB}) + Ambient_{RGB}) \times Detail_{RGB} \tag{5.1}$$

where $\overrightarrow{L}$ is the light vector from that pixel to the light, interpolated over the impostor's quad by the rasteriser and normalised in the fragment processor. $\overrightarrow{N}$ is the decompressed normal for that pixel retrieved from the normal map. $Region_{RGB}$ is the colour for the region of that pixel given by the colour variation algorithm as described in the next section. $Ambient_{RGB}$ is a constant ambient colour. $Detail_{RGB}$ is a detail map that is used to add extra details such as face and clothing variations. It should be noted that while this lighting equation suits the rendering of plain-clothed humans, any other lighting equation could be used instead by simply changing the shader associated with both the impostor and geometric model. For example, an anisotropic metallic shader could be implemented for rendering both impostors and geometric versions of cars including specular highlights and even environment-mapped reflections, subject to performance constraints.

Using this single-pass impostor lighting method implemented with programmable hardware eliminates the excess overdraw introduced by an equivalent multi-pass algorithm and allows any lighting equation to be used for both the impostor and the geometric representation. It also eliminates the extra API calls necessary for

setting up impostor lighting with DOT3_RGB_ARB, reducing the potential for a CPU bottleneck in the driver. By exploiting the programmable pipeline, we allow for a more powerful and generalised impostor rendering algorithm at a reduced cost compared to the multi-pass fixed function pipeline implementation.

5.2.3 Impostor Variation

As dynamic lighting can be implemented in a single pass by employing programmable graphics hardware, a method to introduce colour variation for impostors in the same pass is also required or else the advantages of avoiding a multi-pass algorithm will be lost.

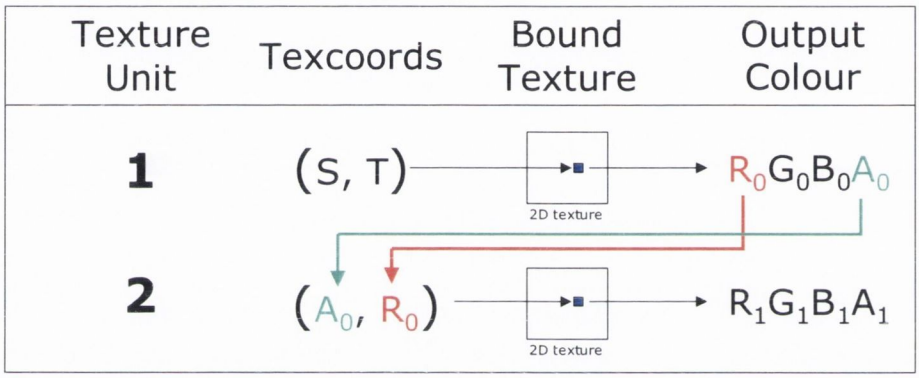


Figure 5.7: *Texture indirection*, using the red and alpha channels of a 2D texture lookup as the texture coordinates of another 2D lookup.

We achieve this by using a feature of programmable hardware called *texture indirection*. This is where any channel of the RGBA value resulting from one texture lookup can be used as the texture coordinates of a subsequent texture lookup (see Figure 5.7). We already have the different colouring regions encoded in the alpha channel of each impostor, so we can use this alpha value to perform another lookup into a special colour map. This colour map is simply a one-dimensional texture where every pixel corresponds to the colouring of a particular region in the impostor. After this second texture lookup, we have an unshaded impostor that has each region coloured according to the colour map supplied. This is then combined with the rest of the impostor shading equation (see Equation 5.1) to produce the final impostor image. An illustration of the entire impostor shading procedure can be seen in

Figure 5.8.

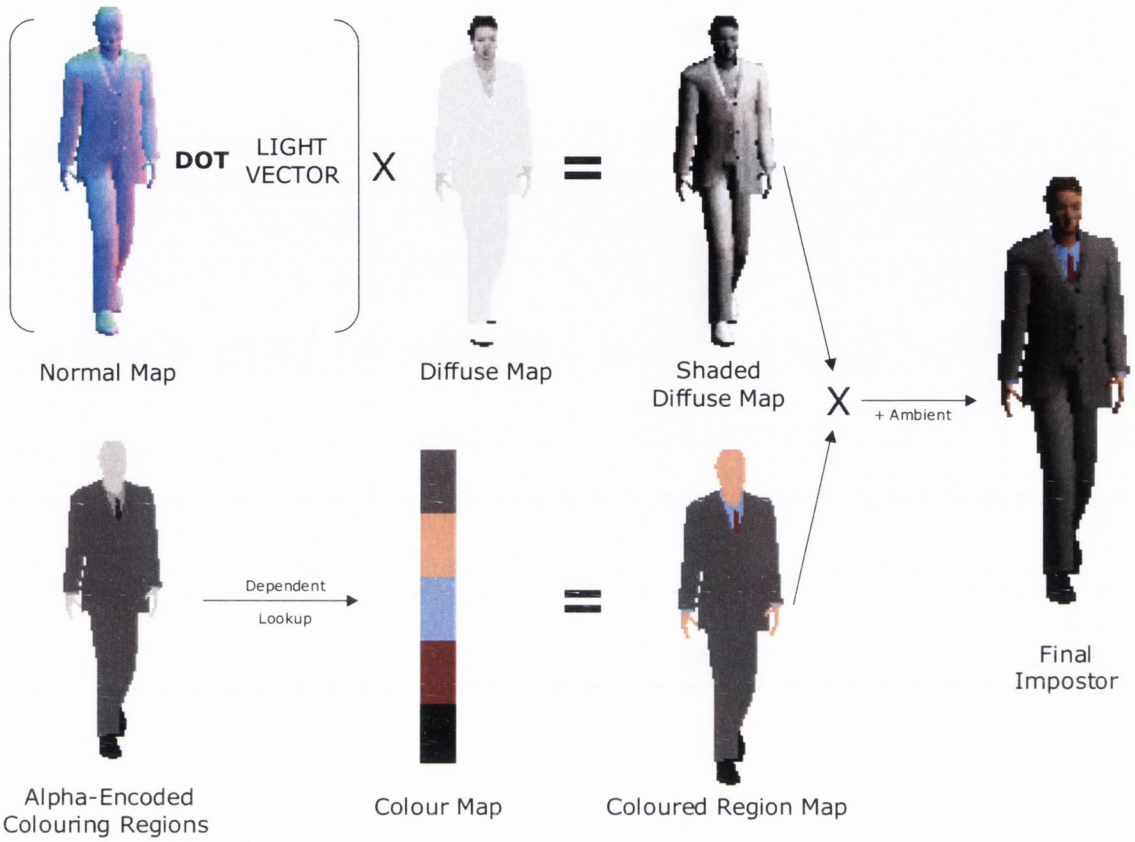


Figure 5.8: The single-pass impostor shading and colouring process

In a multi-pass algorithm, one pass must be made for every colour region in the impostor, bringing with it the additional CPU, vertex and fill rate overhead that each extra pass requires. By employing this texture indirection technique in programmable hardware, the number of colouring regions available to an impostor is instead only limited by the precision of the alpha channel. For a regular impostor with 8 bits per channel, this allows a maximum of 255 differently coloured regions. This is reduced to 16 regions if S3 texture compression [120] is employed to reduce the amount of texture memory used by an imposter, because under S3 compression the alpha channel is compressed to a 4-bit representation. It should also be noted that due to the occurrence of only a single instance of texture indirection in this algorithm, shading performance is independent of the number of regions being coloured; one big colour region will have the same performance as many small regions being coloured

differently.

For similar reasons as for single-pass dynamic lighting, using the fragment shader to control colouring with texture-encoded colour maps means that the API calls used in a multi-pass algorithm to set the diffuse colour at each pass are not necessary. Again, this reduces the number of API calls necessary for every single impostor, further reducing the possibility of a function-call bottleneck in the driver.

5.2.4 Authoring Outfits

When dealing with a large crowd of coloured impostors, the chosen colours used for outfits are very important. A randomly-chosen set of colours can produce garish outfits that are jarring to the eye and detract from the believability and realism of a scene. In addition, specifying which colours to use programmatically can be difficult and inefficient without adding the extra complexity of a scripting language interface to the impostor rendering algorithm.

By employing texture indirection and encoding impostor colours into textures, it is possible to shift the control of outfit authoring into an artist-controlled tool that constructs textures quickly and easily. This allows for the rapid generation and display of many different outfits that are appropriate to the impostor in question, with a minimum of programming overhead. An example of the outfit authoring tool designed for our impostors is shown in Figure 5.9. These colour maps can also be exported and applied to the geometric models, ensuring an exact match and thus minimal artefacts when switching between representations.

The storage space required to store each colour map is negligible, as only one pixel is required to represent each colour region. Therefore, over 1000 different outfits with 4 different regions each could be designed for an impostor and only require approximately 100k of memory. Furthermore, with careful assignment of colouring regions, the same colour map can be used for more than one type of impostor.

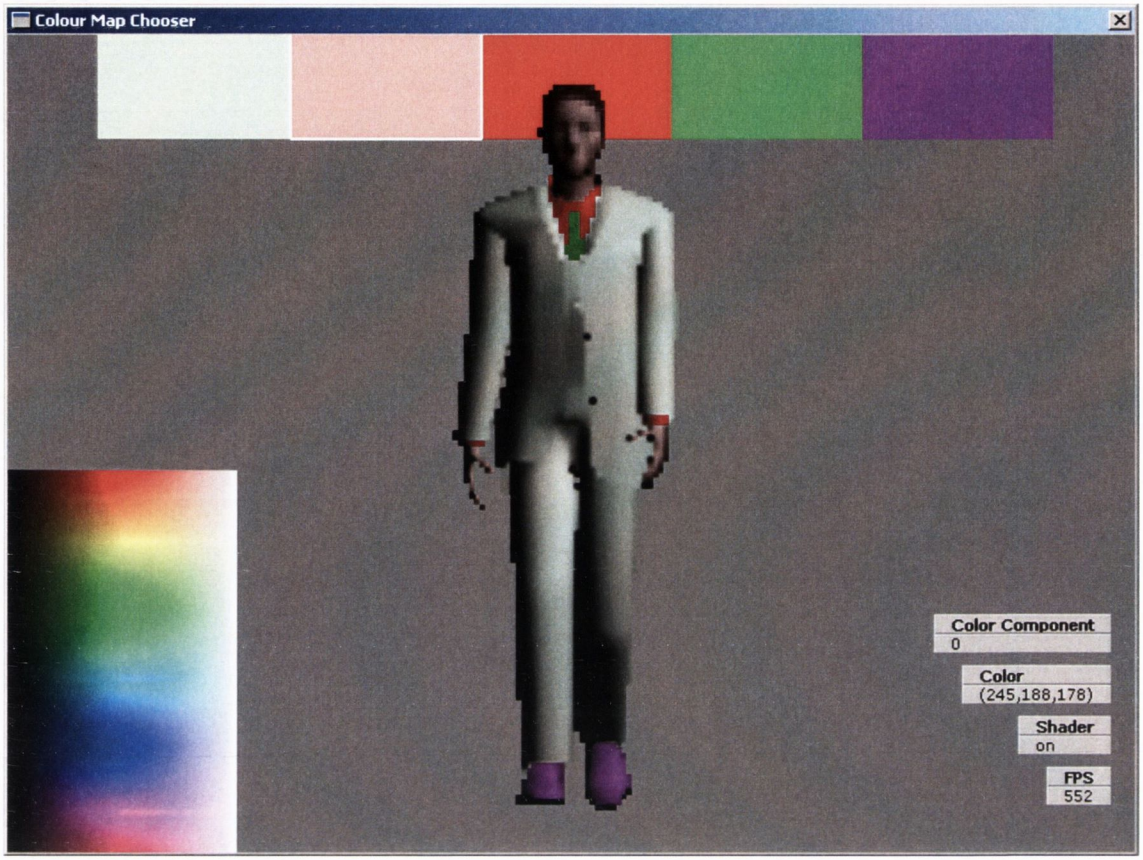


Figure 5.9: The outfit tool used for choosing impostor colour maps

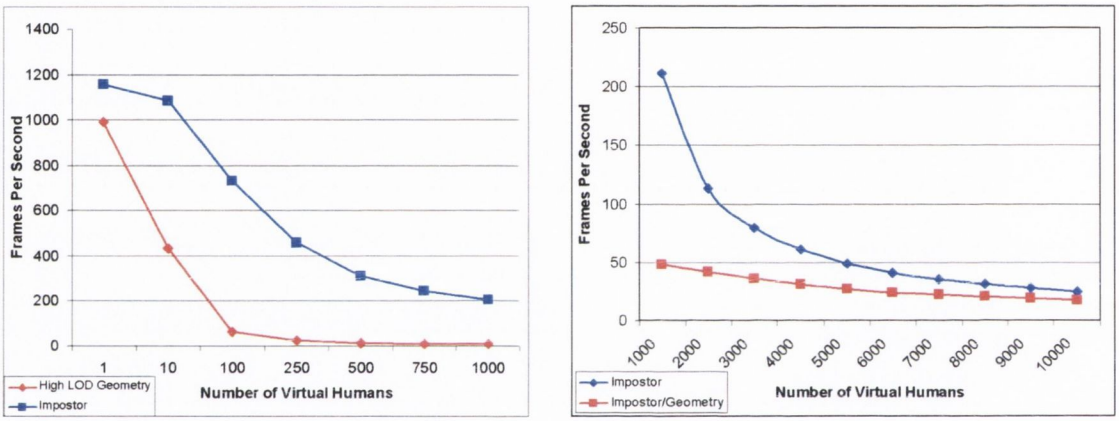
5.3 Results

We measured the performance of the impostors as part of the Geopostor crowd rendering system, as this was the system that they were developed for. All of our tests were performed using a Pentium 4 2Ghz processor, with 512Mb RAM and a GeForce 4 Ti4600 3D card with 128MB of video memory.

We ran tests investigating how the number of virtual humans and the representation used affected the frame rate. These tests used an impostor and a geometric representation (consisting of approximately 2200 triangles) for 1, 10, 100, 250, 500 and 1000 virtual humans as shown in Figure 5.10(a). It should be noted that for each test, all of the virtual humans were fully lit but never frustum or occlusion culled and were therefore always on-screen.

We also tested how using our two LOD representations affected the systems per-

formance in comparison to just using an impostor representation. These tests were carried out for 1,000 - 10,000 virtual humans at 1,000 human intervals. A maximum of 10,000 virtual humans was chosen as this was considered to be the maximum amount that would be needed on-screen for scenes such as an army of characters or a stadium of spectators. In these tests, the number of virtual humans using the geometric representation was set to 100 to keep their rendering cost constant thus allowing the performance impact of using the impostors to be measured. The graph in Figure 5.10(b) illustrates that in the impostor/geometry case, the impostor representation has a minimal impact on the rendering time as the number of virtual humans increases.



(a) (b)

Figure 5.10: (a) Impostor vs Geometry. (b) Impostor vs Impostor/Geometry

Our results so far have convinced us that human impostors are a excellent substitute for geometry, not only because of proven rendering efficiency gains, but also in terms of visual fidelity. At certain distances, it is virtually impossible to determine whether the high-resolution model or the impostor is being rendered. Furthermore, these results have been validated by perceptual experiments [87, 53] which also determined the thresholds at which switching between the impostor and geometric representations are indistinguishable. This would not have been possible without programmable graphics hardware and the methods described in this chapter for matching exactly both the lighting and the colour variation of the two representations.

5.4 Cluster Implementation

The implementation of a crowd rendering system is not as directly relevant to the cluster as the previous algorithms such as isosurface extraction. This is largely due to the fact that the problem here is a rendering one, and as such is quite different to the processing-bound applications in scientific visualisation.

While programmable graphics hardware allows us to render large crowds of convincing impostor-based human representations, the major limiting factors are in the graphics card itself; texture memory is limited and as such the amount of different impostor textures that can be stored is also limited. A cluster implementation of the crowd rendering system will also encounter this limitations, as it uses the same commodity graphics hardware to perform the rendering work; every card that needs to render an impostor will require the impostor's texture to be resident in video memory, as texturing directly from system memory has shown to cause texture thrashing and is detrimental to framerate.

However in the context of a crowd rendering system, the entire resources of the cluster can still be put to good use. Along with the rendering system, a sophisticated simulation engine is needed to drive a large-scale crowd. Artificial intelligence, path finding, obstacle avoidance, behaviour planning and many other factors need to be evaluated at every frame in order to produce a convincing, dynamic group of humans. With care, these processes can be partitioned among the cluster's nodes and parallelised on the FPGAs in order to simulate a very large crowd at interactive rates. The most obvious partitioning would be a world-space one, where humans in one sector of the virtual world all reside on one node of the cluster. This would allow humans close to each other to interact with each other without having to access remote memory locations. Having the low-latency SCI interconnect allows the nodes to communicate at high speeds and therefore allows the behavioural algorithms of more distant individuals in the crowd to still interact with each other, perhaps at a lower update rate than those in close proximity.

Taking further advantage of the cluster's architecture can also provide other methods of simulating the crowds. An asymmetric approach could be taken, where one node is responsible for generating the actual impostor textures in real-time by rendering to textures and providing these textures to the other nodes for use in the

main human rendering system. This would provide a more consistent rendering look, rather than mixing pre-generated images with live rendered geometry. Alternatively, the humans could be distributed between the nodes based on the impostor textures used. Each node would be entirely responsible for rendering certain impostors, and then the final frame would be composited as a post-processing step. This would alleviate the problem of every video card needing to contain a copy of every impostor texture, allowing for more variety in the scene and therefore more realism.

Implementing a crowd system on the Cell processor would certainly allow for parallelisation of the simulation among the many SPUs. However as rendering speed is a large part of the bottleneck in the crowd system, there is a limit to the usefulness of increasing the amount of processing without a corresponding increase in rendering. In the cluster's case, being able to scale the rendering power along with the processing power has many obvious benefits and allows for a larger and more detailed crowd.

Chapter 6

Conclusions and Future Work

This thesis has described the exploitation of commodity parallel hardware for graphics algorithms and architectures. In this final chapter we summarise our contributions and suggest potential future avenues of investigation based upon this work, as well as briefly discussing the future of commodity parallel hardware.

6.1 Summary of Contributions

New cluster software infrastructure: We have described a new tightly-coupled, scalable cluster framework composed of heterogeneous commodity components, combined with a minimal custom hardware element. This cluster takes advantage of the strengths of its constituent parallel parts, and is connected by a low-latency, high-bandwidth SCI interconnect which implements a single distributed shared memory space to reduce data replication. Additional processing power is provided by reconfigurable FPGAs. A software framework has been proposed to implement a distributed graphics driver based upon ATI driver specifications for the R200, giving the cluster the potential to significantly accelerate existing and future parallel rendering algorithms.

Volume visualisation: We have presented an overview of a volume visualisation system for generating publishable images of scientific data, particularly confocal fluorescence microscopy data. It uses programmable graphics hardware in the form of texture shaders and register combiners to achieve a performance

increase and improve image quality, as well as implementing an adaptation of the marching cubes algorithm in order to generate polygonal surfaces of equal isovalues.

Volume simplification: In an attempt to reduce the amount of work that needs to be done during isosurface extraction of a complete volume, we have presented a simple, quick and effective method of volume simplification that retains the coarse features of the volume. We have demonstrated that even one level of simplification can reduce the size of the volume considerably, by up to a factor of ten. This method has also been shown to improve the visual quality of the inherently noisy surfaces extracted from confocal microscopy datasets.

Isosurface extraction on Cell: A novel algorithm for isosurface extraction on the new Cell processor has been demonstrated. We have demonstrated peak processing speeds of over 100 million tetrahedra per second on a dual Cell server, or over 47 million tetrahedra per second on a single Cell. We have shown that these increases have come about through the combination of a streaming and parallelisation scheme that takes advantage of Cell's ability to effectively eliminate memory latency by hiding it behind processing time. We have also given a general overview of the Cell processor and described how to leverage its power in the adaptation of an existing algorithm. This same approach can be used for other compute- and bandwidth-bound algorithms suitable for parallelisation, and similar improvements in processing times can be expected. Moreover, the fact that these improvements can be achieved using a commodity processor means that it is a superior alternative for applications where more processing power is needed, but the cost of expensive dedicated hardware would be prohibitive.

Crowd rendering: The methods used by Tecchia et al. [132] have been built upon and improved to produce a lit and varied human impostor in a single rendering pass. We have implemented a method that is controllable by artist-generated textures and allows many more regions of variation at no extra cost. Additionally, the use of the programmable pipeline permits the use of arbitrary lighting equations for impostors, an important consideration for other classes

of material such as cloth or metal.

6.2 Future Work

6.2.1 Isosurface Extraction and Volumetric Simplification

The approach to isosurface extraction on Cell described in Section 4.4 concentrates on eliminating memory latency and efficiently transferring data to the SPUs for processing. It does not address the issues of further acceleration by using spatially hierarchical data structures such as octrees [143] or interval trees [19]. There is little doubt that since the current limit of our method is the processing time needed by the SPUs, reducing the amount of data to be processed would lead to further increases in speed.

Another issue to be investigated is that of mixing SPU application models in order to improve overall system performance. For example, some SPUs could be dedicated to isosurface extraction while the others perform mesh simplification on the polygons already produced. A broad range of load-balancing implications are introduced by the two-tier PPE/SPE split, and these need to be explored.

Given the streaming methods presented here, there is no limit on the size of the dataset to be processed. However, as with conventional processors, the size of system memory still limits the amount of data that can be held at any time. This puts a limit on the speed of very large volume isosurface extraction, so out-of-core execution methods such as those proposed by Chiang et al. [16] will also have to be investigated with respect to implementation on Cell.

The volumetric simplification presented in Section 4.3 could also be applied to isosurface extraction on Cell. Due to the buffering methods, transfer time is not an issue. Therefore, simplification could be done on the SPUs before performing extraction. However it remains to be seen whether this would be beneficial to execution time or not. It would also require more replication in the SPUs to properly perform a high level of the 3D filtering, as the overlap between chunks and slices would need to be higher.

There is much future work to be done in this area of isosurface extraction on Cell, and on Cell hardware in general. Because it is a relatively new architecture,

its usefulness for compute-, latency- and bandwidth-bound problems is just becoming apparent. A significant change in attitude towards system design is needed in order to use it to its full potential, but as the hardware becomes more widespread and support tools improve, so too will the programming paradigms specific to this platform.

6.2.2 Parallel Commodity Cluster

A working prototype of at least two boards will be needed before a lot of the implementation and testing can be done. When this is possible, there are many design choices that must be validated before committing fully to them.

The implications of accessing remote buffers must be evaluated and compared to the local buffers usually accessed by the normal graphics driver in AGP memory with respect to access times and bandwidth. If remote indirect and ring buffer fetches are feasible, it allows a much more flexible and distributed model to be used when designing the cluster-based driver.

Another area that needs careful consideration is the concurrency mediation methods that are used to arbitrate simultaneous accesses to a custom board's local ring buffer. Using the shared memory space for the transfer of command packets is a powerful method of communicating quickly and efficiently with the cluster's GPUs, so a lightweight and yet robust solution to this problem must be found.

The possibility of reducing data replication with shared memory must be weighed carefully against the speed of data access. Due to the NUMA architecture of the cluster, it would be faster to keep a copy of the data in the local memory of every node that requires it. However, this would be a less efficient use of available memory and should be avoided if possible. Due to the fact that data is only loaded into video memory once and then accessed from there by the GPU, it is likely that the cost of transferring the data over the interconnect would be acceptable in order to eliminate replication.

6.3 The Future of Parallel Hardware

Given the recent improvements in commodity parallel hardware such as FPGAs and the Cell processor, and the research being published here and elsewhere on how best to exploit them for computer graphics algorithms, it is becoming clear that the future of processors (and therefore computing in general) lies in parallel systems. Even processor manufacturers such as Intel and AMD are recognising this fact and planning accordingly. In April of 2005, Intel announced the Pentium Extreme Edition, its first dual-core product. It has since modified its overall strategy to specify multicore functionality as a central feature of its future CPU architectural designs, also producing supporting software to help developers take full advantage of multicore platforms. The new Core Microarchitecture that Intel is promoting as the basis of the next generation of Intel products is further evidence of Intel's commitment to parallel hardware as the way forward.

The driving force behind the enormous increase in GPU power over the last ten years (the original 3dFX Voodoo was only released in 1996) has undeniably been the computer games industry. Furthermore, the parallelism inherent in the modern GPU allows processing speeds that are far beyond the capabilities of a comparable sequential processor - albeit only for a specialised purpose, in this case 3D rendering. Once such a powerful processor became available at an affordable price, it was inevitable that it would be adapted for use in other areas such as scientific simulation. While the GPU can be adapted to perform these tasks very well through the methodologies of GPGPU, it is clear that having to 'shoehorn' such generalised computational tasks into a specific graphics rendering paradigm is far from ideal. Certain desirable general purpose operations are either restrictive or impossible because of fundamental differences in the underlying architecture - writing to an arbitrary location in memory being one example. These currently have to be overcome by roundabout methods such as multipass algorithms that would not be necessary on a general purpose processor.

At the same time, the speed afforded by the modern GPU is possible because it is geared toward specific operations. Additionally, the reason that the GPU is so attractive to the proponents of GPGPU is their low cost, which is driven down by their primary consumers, games players. A general stream processor could allow

these operations with a more general and useful instruction set, but the demand would be nowhere near that of GPUs, resulting in a much higher market price.

Nevertheless, now that the advantages of parallel processing have been made obvious by GPUs, the need to adapt algorithms to fit into the GPU programming model will be lessened as other parallel architectures arise. The increase in research into the application of processors such as FPGAs and Cell shows that much more can be offered besides raw parallel processing power, including much greater flexibility in implementation, larger memory, and far greater scalability. While its importance in general computation may be lessened by other architectures, it is still clear that the GPU will continue to be at the cutting edge of graphics rendering hardware for the foreseeable future.

Bibliography

[1] K. Alnaes, E. Kristiansen, D. Gustavson, and D. James. Scalable Coherent Interface. *Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering*, pages 446 – 453, May 1990.

[2] N. Atay, J. Lockwood, and B. Bayazit. A collision detection chip on reconfigurable hardware. *Proceedings of 2005 Pacific Conference on Computer Graphics and Applications (Pacific Graphics)*, October 2005.

[3] ATI Technologies Inc. Crossfire. http://www.ati.com/technology/crossfire.

[4] ATI Technologies Inc. Product comparison guide. http://apps.ati.com/ATIcompare.

[5] ATI Technologies Inc. RAGE 128 software development guide. Internal Technical Reference Manual, 1999.

[6] ATI Technologies Inc. R200 programming reference guide. Internal Technical Reference Manual, 2001.

[7] ATI Technologies Inc. R200 register reference. Internal Technical Reference Manual, 2001.

[8] A. Aubel, R. Boulic, and D. Thalmann. Lowering the cost of virtual human rendering with structured animated impostors. *In Proceedings of WSCG '99*, 1999.

[9] J. S. Beeckler and W. J. Gross. FPGA particle graphics hardware. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, pages 85 – 94, April 2005.

[10] J. F. Blinn. Models of light reflection for computer synthesized pictures. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 192–198, New York, NY, USA, 1977. ACM Press.

[11] T. Boubekeur and C. Schlick. Generic mesh refinement on GPU. *Proceedings of Graphics Hardware*, 2005.

[12] I. Buck, G. Humphreys, and P. Hanrahan. Tracking graphics state for networked rendering. In *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 87–95, New York, NY, USA, 2000. ACM Press.

[13] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *VVS '94: Proceedings of the 1994 symposium on Volume visualization*, pages 91–98, New York, NY, USA, 1994. ACM Press.

[14] B. P. Carneiro, C. Silva, and A. E. Kaufman. Tetra-cubes: An algorithm to generate 3D isosurfaces based upon tetrahedra. *Proceedings of SIBGRAPI '96*, pages 205–210, 1996.

[15] M. Charalambous, P. Trancoso, and A. Stamatakis. Initial experiences porting a bioinformatics application to a graphics processor. *Proceedings of the 10th Panhellenic Conference in Informatics*, 2005.

[16] Y.-J. Chiang, C. T. Silva, and W. J. Schroeder. Interactive out-of-core isosurface extraction. In D. Ebert, H. Hagen, and H. Rushmeier, editors, *IEEE Visualization '98*, pages 167–174, 1998.

[17] A. C. Chow, G. C. Fossum, and D. A. Brokenshire. A programming example: Large FFT on the cell broadband engine. *IBM White Paper*, 2005.

[18] H. Christopherson, W. Pickell, A. Koller, S. Kannan, and E. Johnson. Small adaptive flight control systems for UAVs using FPGA/DSP technology. *AIAA 3rd "Unmanned Unlimited" Technical Conference, Workshop and Exhibit*, 2004.

[19] P. Cignoni, P. Marino, C. Montani, E. Puppo, and R. Scopigno. Speeding up isosurface extraction using interval trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158–170, 1997.

[20] P. Cignoni, C. Montani, R. Scopigno, and C. Rocchini. A general method for preserving attribute values on simplified meshes. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 59–66, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[21] J. Cohen, M. Olano, and D. Manocha. Appearance-preserving simplification. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 115–122, New York, NY, USA, 1998. ACM Press.

[22] J. Danskin and P. Hanrahan. Fast algorithms for volume ray tracing. In *VVS '92: Proceedings of the 1992 workshop on Volume visualization*, pages 91–98, New York, NY, USA, 1992. ACM Press.

[23] DARPA. HPC challenge 2005 results. http://icl.cs.utk.edu/hpcc/hpcc_results.cgi, 2005.

[24] M. A. de Barros and M. Akil. Low level image processing operators on FPGA: Implementation examples and performance evaluation. *ICPR-D*, 94:262–267, 1994.

[25] P. de Heras Ciechomski, B. Ulicny, R. Cetre, and D. Thalmann. A case study of a virtual audience in a reconstruction of an ancient roman odeon in aphrodisias. *VAST '04: The 5th International Symposium on Virtual Reality, Archaeology and Cultural Heirtage*, pages 9–17, 2004.

[26] T. DeRose, M. Kass, and T. Truong. Subdivision surfaces in character animation. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 85–94, New York, NY, USA, 1998. ACM Press.

[27] C. Dick and Y. Krikorian. A system-level design approach for FPGA-based DSP implementations. *DSP World*, Spring 1999.

[28] K. Diefendorff, P. Dubey, R. Hochsprung, and H. Scale. Altivec extension to PowerPC accelerates media processing. *IEEE Micro*, 20(2):85–95, 2000.

[29] S. Dobbyn, J. Hamill, K. O'Conor, and C. O'Sullivan. Geopostors: a real-time geometry / impostor crowd rendering system. In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 95–102, New York, NY, USA, 2005. ACM Press.

[30] Dolphin Interconnect Solutions Inc. Dolphin SuperSockets. http://www.dolphinics.com/products/software/sci_sockets.html.

[31] Dolphin Interconnect Solutions Inc. SISCI developer's kit. http://www.dolphinics.com/products/software/sisci_devkit.html.

[32] Dolphin Interconnect Solutions Inc. D331 PCI-SCI adapter card. http://www.dolphinics.com/products/hardware/pci64.html, 2006.

[33] S. Dominé and J. Spitzer. Texture shaders. http://developer.nvidia.com/object/texture_shaders.html, 2001.

[34] B. Dudash. DX10, batching, and performance considerations. http://developer.nvidia.com/object/dx10-instancing-gdc-2006, 2006.

[35] H. Edelsbrunner. Dynamic data structure for orthogonal intersection queries. Technical Report F59, Inst. Informationsverarb. Tech. Univ. Graz, Graz, Austria, 1980.

[36] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the cell broadband engine architecture. *IBM Systems Journal*, 45(1), 2006.

[37] M. Eldridge, H. Igehy, and P. Hanrahan. Pomegranate: A fully scalable graphics architecture. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 443–454, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[38] T. Erjavec. The power in Xilinx: Power architecture technology makes flexible design easy. http://www-128.ibm.com/developerworks/power/library/panljun04-xilinx/index.html, 2004.

[39] J. Eyre. FPGA/DSP blend tackles telecom apps. *Electronic Engineering Times*, July 2002.

[40] R. Fernando and M. J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics.* Addison-Wesley Professional, 2003.

[41] J. Gao and H.-W. Shen. Parallel view-dependent isosurface extraction using multi-pass occlusion culling. In *PVG '01: Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, pages 67–74, Piscataway, NJ, USA, 2001. IEEE Press.

[42] C. Giertsen and J. Peterson. Parallel volume rendering on a network of workstations. *IEEE Computer Graphics and Applications*, 13(6):16 – 23, November 1993.

[43] F. Goetz, T. Junklewitza, and G. Domik. Real-time marching cubes on the vertex shader. *Eurographics 2005 short presentations*, pages 5–9, 2005.

[44] H. Gouraud. Computer display of curved surfaces. *IEEE Transactions on Computers*, 20(6):623–629, 1971.

[45] N. Govindaraju, S. Redon, M. C. Lin, and D. Manocha. CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. *Proceedings of Graphics Hardware*, 2003.

[46] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High performance graphics coprocessor sorting for large database management. *Proceedings of ACM SIGMOD Conference*, 2006.

[47] GPGPU. General-Purpose Computation Using Graphics Hardware. http://www.gpgpu.org.

[48] Graphic Remedy. gDEBugger. http://www.gremedy.com, 2006.

[49] S. Green. NVIDIA OpenGL update. http://developer.nvidia.com/object/opengl-nvidia-extensions-gdc-2006.html, 2006.

[50] S. A. Green and D. J. Paddon. Exploiting coherence for multiprocessor ray tracing. *IEEE Comput. Graph. Appl.*, 9(6):12–26, 1989.

[51] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[52] A. Guéziec and R. Hummel. Exploiting triangulated surface extraction using tetrahedral decomposition. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):328–342, 1995.

[53] J. Hamill, R. McDonnell, S. Dobbyn, and C. O'Sullivan. Perceptual evaluation of impostor representations for virtual humans and buildings. *Eurographics 2005 Proceedings, Computer Graphics Forum*, 24(3), 2005.

[54] C. D. Hansen and P. Hinker. Massively parallel isosurface extraction. In *VIS '92: Proceedings of the 3rd conference on Visualization '92*, pages 77–83, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[55] M. Harris. *Real-Time Cloud Simulation and Rendering.* PhD thesis, University of North Carolina, 2003.

[56] E. Hart. ARB fragment program: Fragment level programmability in OpenGL. http://www.ati.com/developer/techpapers_archive.html. Game Developers Conference 2003.

[57] Havok. http://www.havok.com.

[58] A. Herout and P. Zemcík. Hardware pipeline for rendering clouds of circular points. In *WSCG (Full Papers)*, pages 17–22, 2005.

[59] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. WireGL: A scalable graphics system for clusters. *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 129–140, 2001.

[60] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: A stream processing framework for interactive graphics on clusters. *SIGGRAPH 2002, Computer Graphics Proceedings*, pages 693 – 702, 2002.

[61] IBM. Cell Broadband Engine resource center. http://www-128.ibm.com/developerworks/power/cell, 2005.

[62] H. Isobe, T. Miyagoshi, K. Shibata, and T. Yokoyama. Filamentary structure on the sun from the magnetic rayleigh-taylor instability. *Nature*, 434(7032):478 – 481, 2005.

[63] S. Jeschke, M. Wimmer, and W. Purgathofer. Image-based representations for accelerated rendering of complex scenes. *In Eurographics 2005 STAR Reports*, pages 1–20, 2005.

[64] T. Kaneko, T. Takahei, M. Inami, N. Kawakami, Y. Yanagida, T. Maeda, and S. Tachi. Detailed shape representation with parallax mapping. *Proceedings of ICAT 2001, Tokyo, Japan*, 2001.

[65] F. Kelly and A. Kokaram. Fast image interpolation for motion estimation using graphics hardware. *IS&T/SPIE Electronic Imaging - Real-Time Imaging VIII*, January 2004.

[66] F. Kelly and A. Kokaram. Graphics hardware for gradient based motion estimation. *IS&T/SPIE Electronic Imaging - Embedded Processors for Multimedia and Communications*, January 2004.

[67] E. Keppel. Approximating complex surfaces by triangulation of contour lines. *IBM Journal of Research and Development*, 19(1):2–11, 1975.

[68] J. Kessenich, D. Baldwin, and R. Rost. The OpenGL shading language specification v1.10.59. http://www.opengl.org.

[69] P. Kipfer and R. Westermann. GPU construction and transparent rendering of iso-surfaces. In G. Greiner, J. Hornegger, H. Niemann, and M. Stamminger, editors, *Proceedings Vision, Modeling and Visualization 2005*, pages 241–248. IOS Press, infix, 2005.

[70] T. Klein, S. Stegmaier, and T. Ertl. Hardware-accelerated reconstruction of polygonal isosurface representations on unstructured grids. In *PG '04: Proceedings of the Computer Graphics and Applications, 12th Pacific Conference on (PG'04)*, pages 186–195, Washington, DC, USA, 2004. IEEE Computer Society.

[71] J. M. Kniss, K. Engel, M. Hadwiger, and C. Rezk-Salama. High-quality volume graphics on consumer PC hardware. *Siggraph Course notes*, 42, 2002.

[72] J. M. Kniss, G. Kindlmann, and C. Hansen. Interactive volume rendering using multi-dimensional transfer functions and direct manipulation widgets. *Proceedings of IEEE Visualization*, pages 255–262, 2001.

[73] G. Knittel and G. Zachmann. High-performance collision detection hardware. Technical Report CG-2003-3, University Bonn, Informatikk II, Bonn, Germany, August 2003.

[74] A. Koide, A. Doi, and K. Kajioka. Polyhedral approximation approach to molecular orbital graphics. *J. Mol. Graph.*, 4(3):149–155, 1986.

[75] V. Kokkevis, S. Osman, and E. Larsen. High-performance physics solver design for next generation consoles. Presentation at Game Developers Conference, 2006. http://www.research.scea.com/research/research.html.

[76] K. Kreeger and A. Kaufman. PAVLOV: a programmable architecture for volume processing. In *HWWS '98: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 77–ff., New York, NY, USA, 1998. ACM Press.

[77] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 451–458, New York, NY, USA, 1994. ACM Press.

[78] D. Laur and P. Hanrahan. Hierarchical splatting: a progressive refinement algorithm for volume rendering. In *SIGGRAPH '91: Proceedings of the 18th*

annual conference on Computer graphics and interactive techniques, pages 285–288, New York, NY, USA, 1991. ACM Press.

[79] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):2937, May 1988.

[80] M. Levoy. Efficient ray tracing of volume data. *ACM Trans. Graph.*, 9(3):245–261, 1990.

[81] W. E. Lorensen and H. E. Cline. Marching cubes: a high resolution 3D surface construction algorithm. *Computer Graphics (SIGGRAPH '87 Proceedings)*, 21(4):163–170, 1987.

[82] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of detail for 3D computer graphics*. Morgan Kaufmann, 2002.

[83] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications*, 14(4):59–68, 1994.

[84] P. W. C. Maciel and P. Shirley. Visual navigation of large environments using textured cluster. *SI3D '95: Proceedings of the 1995 Symposium on Interactive 3D Graphics*, pages 95–102, 1995.

[85] P. Mackerras. A fast parallel marching-cubes implementation on the Fujitsu AP1000. *Australian National University*, Tech Report(TR-CS-92-10), 1992.

[86] N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99 – 108, June 1995.

[87] R. McDonnell, S. Dobbyn, and C. O'Sullivan. LOD human representations: A comparative study. *Proceedings of the First International Workshop on Crowd Simulation (V-CROWDS '05)*, 2005.

[88] M. Meißner, U. Kanus, G. Wetekam, J. Hirche, A. Ehlert, W. Straßer, M. Doggett, P. Forthmann, and R. Proksa. VIZARD II: a reconfigurable interactive volume rendering system. In *HWWS '02: Proceedings of the ACM*

SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, pages 137–146, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[89] K. Menzel. Parallel rendering techniques for multiprocessor systems. In *Proceedings of the Spring School on Computer Graphics (SSCG '94)*, pages 91–103, Bratislava, Slovakia, 1994. Comenius University Press.

[90] J. V. Miller, D. E. Breen, W. E. Lorensen, R. M. O'Bara, and M. J. Wozny. Geometrically deformed models: a method for extracting closed geometric models form volume data. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 217–226, New York, NY, USA, 1991. ACM Press.

[91] B. Minor, G. Fossum, and V. To. Terrain rendering engine (TRE). *IBM White Paper*, 2005.

[92] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, pages 23 – 32, July 1994.

[93] G. E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, April 19, 1965.

[94] MPI Forum. the MPI-2 standard. http://www-unix.mcs.anl.gov/mpi, 2003.

[95] K. Mueller and R. Yagel. Fast perspective volume rendering with splatting by utilizing a ray-driven approach. In *VIS '96: Proceedings of the 7th conference on Visualization '96*, pages 65–ff., Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[96] C. Niederauer, M. Houston, M. Agrawala, and G. Humphreys. Non-invasive interactive visualization of dynamic architectural environments. In *SI3D '03: Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 55–58, New York, NY, USA, 2003. ACM Press.

[97] G. M. Nielson and B. Hamann. The asymptotic decider: resolving the ambiguity in marching cubes. In *VIS '91: Proceedings of the 2nd conference on*

Visualization '91, pages 83–91, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[98] NVIDIA Corporation. cg: C for graphics. http://developer.nvidia.com/Cg.

[99] NVIDIA Corporation. SLI. http://www.slizone.com.

[100] M. M. Oliveira. *Relief Texture Mapping.* PhD thesis, University of North Carolina, 2000.

[101] R. Osborne, H. Pfister, H. Lauer, T. Ohkami, N. McKenzie, S. Gibson, and W. Hiatt. EM-Cube: an architecture for low-cost real-time volume rendering. In *HWWS '97: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 131–138, New York, NY, USA, 1997. ACM Press.

[102] C. O'Sullivan, J. Cassell, H. Vilhjálmsson, J. Dingliana, S. Dobbyn, B. McNamee, C. Peters, and T. Giang. Levels of detail for crowds and groups. *Computer Graphics Forum*, 21(4), November 2002.

[103] C. Parkinson, M. Cooper, I. Hillier, and W. Hewitt. MAVIS: an interactive visualization tool for computational chemistry calculations in a distributed networked environment. *In proceedings of the Pacific Symposium on Biocomputing, Kapalua, Maui, Hawaii, U.S.A.*, January 1998.

[104] V. Pascucci. Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping. *Joint Eurographics - IEEE TVCG Symposium on Visualization (VisSym)*, pages 293–300, 2004.

[105] B. Paul. Introduction to the direct rendering infrastructure. *Tutorial presented at the LinuxWorld 2000 conference*, 2000.

[106] PCI Special Interest Group. PCI-Express specifications. http://www.pcisig.com/specifications/pciexpress.

[107] PCI Special Interest Group. PCI-X 2.0 specifications. http://www.pcisig.com/specifications/pcix_20.

[108] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The VolumePro real-time ray-casting system. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 251–260, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[109] H. Pfister and A. Kaufman. Cube-4 - a scalable architecture for real-time volume rendering. In *VVS '96: Proceedings of the 1996 symposium on Volume visualization*, pages 47–ff., Piscataway, NJ, USA, 1996. IEEE Press.

[110] T. J. Purcell. *Ray Tracing on a Stream Processor.* PhD thesis, Stanford University, March 2004.

[111] A. Raabe, B. Bartyzel, J. K. Anlauf, and G. Zachmann. Hardware accelerated collision detection - an architecture and simulation results. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 130–135, Washington, DC, USA, 2005. IEEE Computer Society.

[112] A. Raabe, S. Hochgürtel, G. Zachmann, and J. K. Anlauf. Space-efficient FPGA-accelerated collision detection for virtual prototyping. In *Design Automation and Test in Europe (DATE)*, pages 6–10, Munich, Germany, March 2006.

[113] F. Reck, C. Dachsbacher, R. Grosso, G. Greiner, and M. Stamminger. Real-time isosurface extraction with graphics hardware. *Computer Graphics Forum*, 22(3):595–603, 2004.

[114] Red Herring. Will Wright changes EA's game. Red Herring magazine, March 2006.

[115] C. Reynolds. Crowd simulation on PS3. Presentation at Game Developers Conference, 2006. http://www.research.scea.com/research/research.html.

[116] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume on standard PC graphics hardware using multi-textures and

multi-stage rasterization. In *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 109–118, New York, NY, USA, 2000. ACM Press.

[117] S. Roettger and T. Ertl. A two-step approach for interactive pre-integrated volume rendering of unstructured grids. *In Symposium on Volume Visualization and Graphics*, pages 23–28, 2002.

[118] M. Sakamoto, H. Nishiyama, H. Satoh, S. Shimizu, T. Sanuki, K. Kamijoh, A. Watanabe, and A. Asahara. An implementation of the feldkamp algorithm for medical imaging on cell. *IBM White Paper*, 2005.

[119] M. Segal and K. Akeley. The OpenGL graphics system: A specification (version 1.5). http://www.opengl.org.

[120] SGI OpenGL Extension Registry. GL_EXT_texture_compression _s3tc. http://oss.sgi.com/projects/ogl-sample/registry/EXT/texture_compression_s3tc.txt.

[121] SGI. Silicon Graphics Prism. http://www.sgi.com/products/visualization/prism, 2006.

[122] SGI. Silicon Graphics VizServer. http://www.sgi.com/products/software/vizserver, 2006.

[123] J. Spitzer. Texture compositing with register combiners. http://developer.nvidia.com/object/registercombiners.html, 2000.

[124] J. Stewart, E. P. Bennett, and L. McMillan. PixelView: a view-independent graphics rendering architecture. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 75–84, New York, NY, USA, 2004. ACM Press.

[125] G. Stoll, M. Eldridge, D. Patterson, A. Webb, S. Berman, R. Levy, C. Caywood, M. Taveira, S. Hunt, and P. Hanrahan. Lightning-2: A high-performance display subsystem for PC clusters. In E. Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 141–148. ACM Press / ACM SIGGRAPH, 2001.

[126] M. Strengert, M. Magallón, D. Weiskopf, S. Guthe, and T. Ertl. Hierarchical visualization and compression of large volume datasets using GPU clusters. *Proc. Eurographics Symposium on Parallel Graphics and Visualization (EGPGV04)*, pages 41–48, 2004.

[127] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system: evolution, experiences, and trends. *Parallel Comput.*, 20(4):531–545, 1994.

[128] J. S. Sven Woop and P. Slusallek. RPU: A programmable ray processing unit for realtime ray tracing. In *Proceedings of ACM SIGGRAPH 2005*, July 2005.

[129] E. Swankoski, R. Brooks, V. Narayanan, M. Kandemir, and M. Irwin. A parallel architecture for secure FPGA symmetric encryption. In *Proceedings of RAW 2004*, 2004.

[130] F. Tecchia and Y. Chrysanthou. Real-time rendering of densely populated urban environments. *Proceedings of the Eurographics Workshop on Rendering Techniques*, pages 83–88, 2000.

[131] F. Tecchia, C. Loscos, and Y. Chrysanthou. Image based crowd rendering. *IEEE Computer Graphics and Applications*, 22(2):36–43, 2002.

[132] F. Tecchia, C. Loscos, and Y. Chrysanthou. Visualizing crowds in real-time. *Computer Graphics Forum*, 21(4):753–765, 2002.

[133] D. Trebilco. GLIntercept. http://glintercept.nutty.org, 2006.

[134] J. Trylska, R. Konecny, F. Tama, C. L. B. 3rd, and J. A. McCammon. Ribosome motions modulate electrostatic properties. *Biopolymers*, 74(6):423 – 431, August 2004.

[135] H. K. Tuy and L. T. Tuy. Direct 2-D display of 3-D objects. *IEEE Computer Graphics and Applications*, 4(10):29–33, November 1984.

[136] T. Udeshi and C. D. Hansen. Parallel multipipe rendering for very large isosurface visualization. In E. Gröller, H. Löffelmann, and W. Ribarsky, editors, *Data Visualization '99*, pages 99–108. Springer-Verlag Wien, 1999.

[137] B. Ulicny, P. de Heras Ciechomski, and D. Thalmann. Crowdbrush: Interactive authoring of real-time crowd scenes. *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/EUROGRAPHICS Symposium on Computer Animation*, pages 243–252, 2004.

[138] A. van Gelder and J. Wilhelms. Topological considerations in isosurface generation. *ACM Trans. Graph.*, 13(4):337–375, 1994.

[139] S. Venkatasubramanian. The graphics card as a stream computer. *Proceedings of IGMOD-DIMACS Workshop on Management and Processing of Data Streams*, 2003.

[140] M. S. Warren, J. K. Salmon, D. J. Becker, M. P. Goda, T. Sterling, and G. S. Winckelmans. Pentium Pro inside: I. A treecode at 430 gigaflops on ASCI Red, II. price/performance of $50/Mflop on Loki and Hyglac. *Proceedings of IEEE Supercomputing '97*, 1997.

[141] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 169–177, New York, NY, USA, 1998. ACM Press.

[142] L. Westover. Footprint evaluation for volume rendering. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 367–376, New York, NY, USA, 1990. ACM Press.

[143] J. Wilhelms and A. V. Gelder. Octrees for faster isosurface generation. *ACM Trans. Graph.*, 11(3):201–227, 1992.

[144] J. Worringen and T. Bemmerl. MPICH for SCI-connected clusters. *In SCI Europe '99*, pages 3 – 11, September 1999.

[145] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, 1995.

[146] C. Wynn. OpenGL vertex programming on future-generation GPUs. http://developer.nvidia.com/object/opengl_vertexprogramming.html.

[147] X. Zhang, C. Bajaj, and W. Blanke. Scalable isosurface visualization of massive datasets on COTS clusters. In *PVG '01: Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, pages 51–58, Piscataway, NJ, USA, 2001. IEEE Press.

[148] I. Zoraja, H. Hellwagner, and V. Sunderam. SCIPVM: Parallel distributed computing on SCI workstation clusters. *Concurrency: Practice and Experience*, 11(3):121–138, 1999.