Design for Scalability in 3D Computer Graphics Architectures

Ph.D. thesis

by

Hans Holten-Lund, M.Sc.

Computer Science and Technology
Informatics and Mathematical Modelling
Technical University of Denmark
http://www.imm.dtu.dk/cst/

July, 2001

This thesis has been submitted in partial fulfillment of the conditions for acquiring the Ph.D. degree at the Technical University of Denmark. The Ph.D. study has been carried out at the Section for Computer Systems and Technology at the Department of Informatics and Mathematical Modelling, supervised by Associate Professors Steen Pedersen and Jan Madsen.

Copenhagen, July 2001

Hans Holten-Lund

Abstract

This thesis describes useful methods and techniques for designing scalable hybrid parallel rendering architectures for 3D computer graphics. Various techniques for utilizing parallelism in a pipelined system are analyzed. During the Ph.D. study a prototype 3D graphics architecture named Hybris has been developed. Hybris is a prototype rendering architecture which can be tailored to many specific 3D graphics applications and implemented in various ways. Parallel software implementations for both single and multi-processor Windows 2000 systems have been demonstrated. Working hardware/software codesign implementations of Hybris for standard-cell based ASIC (simulated) and FPGA technologies have been demonstrated, using manual co-synthesis for translation of a Virtual Prototyping architecture specification written in C into both optimized C source for software and into to a synthesizable VHDL specification for hardware implementation. A flexible VRML 97 3D scene graph engine with a Java interface and C++ interface has been implemented to allow flexible integration of the rendering technology into Java and C++ applications. A 3D medical visualization workstation prototype (3D-Med) is examined as a case study and an application of the Hybris graphics architecture.

Preface

I would like to thank all the people who helped me build the foundations for the Hybris graphics architecture during my Ph.D. studies. A special thanks goes to my advisors; Steen Pedersen for encouraging me during my Ph.D. studies by coming up with challenging tasks such as designing and implementing a 3D medical visualization workstation from scratch to see if my ideas on computer graphics were of any actual use, and Jan Madsen for insight into codesign system design methodologies which are very useful for designing combined hardware/software systems such as the graphics architecture presented in this thesis.

Thanks must also go to Niels Jørgen Christensen for inspiring my interest in parallel computer graphics architectures.

Additionally I would like to thank all the people from 3D-Lab in Copenhagen; Tron Darvann, Per Larsen, Sven Kreiborg and others, for discussions about medical visualization, as well as Niels Egund from Århus Kommunehospital for being willing to test our prototypes in practice.

And also thanks to Professor Arie Kaufman who made it possible for me to carry out six months of my studies at his Visualization Lab in the Computer Science department of the State University of New York at Stony-Brook (SUNY-SB), NY, USA.

Further thanks goes to all the master's students who made contributions to the Hybris graphics architecture as well as the 3D medical visualization workstation.

Some of these master's students include: Martin Lütken [130] , Thomas Gleerup [71] and Henrik Ahrendt Sørensen [207] who proved to be an invaluable help for building and defining the Hybris graphics architecture.

Finally, I would like to thank the other master's students credited for helping building and defining the 3D medical visualization workstation. They include: Mogens Hvidtfeldt [97], Søren A. Møller [151], Kurt Jensen [109], Jacob Winther Madsen [131], Lars Bo Mortensen [160], Torben Yde Pedersen [176], Kenneth Haldbæk Petersen [178], Jan Dueholm Rasmussen [189] and Kim Theilgaard [222].

Contents

Preface v

Contents vii

1 Introduction 1
1.1 Parallel rendering – the next step . . . 2
1.2 Contribution of this thesis . . . 5
1.3 Thesis chapter overview . . . 6

2 Parallel Rendering and Scalability 9
2.1 Scalable 3D graphics architectures . . . 9
2.1.1 General purpose parallel computing system architectures . 10
2.1.2 Scalability of current PC-based 3D graphics processors . . 12
2.1.3 Summary . . . 17
2.2 Parallel rendering concepts . . . 17
2.2.1 Coherence . . . 18
2.2.2 Parallelism in rendering . . . 19
2.3 Parallel rendering as a sorting problem . . . 22
2.3.1 Sort-first . . . 23
2.3.2 Sort-middle . . . 24
2.3.3 Sort-last . . . 26
2.3.4 Hybrid sorting . . . 29
2.4 Bucket sorting . . . 30
2.4.1 Bounding box bucket sorting overlap . . . 31
2.4.2 Exact bucket sorting overlap . . . 33
2.5 Chapter summary . . . 39

3 Designing a Scalable Graphics Architecture 41
3.1 Understanding the problem . . . 41
3.2 Development of the Hybris rendering architecture . . . 43

3.2.1 Clipping . . . 44
3.2.2 Fast tile boundary clipping . . . 46
3.2.3 Fast floating point to fixed-point conversion . . . 48
3.2.4 Back-face culling . . . 49
3.2.5 Hierarchical back-face culling . . . 51
3.2.6 Pixel addressing rounding rules . . . 53
3.2.7 Sub-pixel triangle culling . . . 54
3.2.8 Sub-pixel shading correction . . . 55
3.2.9 An alternative: Point rendering . . . 57
3.2.10 Half-plane edge functions . . . 59
3.2.11 Packet data format for a triangle node . . . 62
3.3 Object partitioning . . . 65
3.3.1 Triangle strips . . . 66
3.3.2 Indexed triangle meshes . . . 67
3.3.3 Triangle mesh partitioning with MeTiS . . . 68
3.4 Partitioned object-parallel renderer front-end . . . 71
3.5 Triangle setup and bucket sorting . . . 73
3.6 Tile-based image-parallel renderer back-end . . . 77
3.6.1 Pipeline stages . . . 77
3.6.2 Load balancing the back-end rasterization pipeline . . . 79
3.6.3 Parallel tile rendering . . . 80
3.6.4 Image composition of tiles . . . 80
3.6.5 Interleaved pixel parallel back-end rasterization pipeline . 81
3.6.6 Anti-aliasing for the tile renderer . . . 84
3.7 Texture mapping . . . 89
3.8 Chapter summary . . . 91

4 Codesign for Hardware and Software Implementations 93
4.1 Design methodology . . . 93
4.1.1 Codesign using C language for architectural design . . . 95
4.1.2 Using C language for hardware description . . . 97
4.2 Standard physical interfaces . . . 99
4.2.1 AGP – A fast interface for graphics . . . 99
4.2.2 PCI . . . 102
4.3 Implementing the Hybris graphics architecture . . . 103
4.3.1 Single CPU software implementation . . . 103
4.3.2 Multiple CPU parallel software implementation . . . 106
4.4 ASIC implementation . . . 108
4.5 FPGA implementation . . . 111
4.5.1 PCI bandwidth . . . 113

4.5.2 VGA video output 115
4.5.3 Physical properties 120
4.6 Performance figures for the implementations 125
4.7 Prospects for future HW/SW implementations 128
4.8 Chapter summary 130

5 Interfaces and Applications 131
5.1 Virtual Reality 131
5.2 3D application interfaces 133
5.2.1 Immediate-mode graphics interface 133
5.2.2 Retained-mode graphics interface 133
5.3 The Hybris VRML engine 135
5.4 Introduction to visualization 136
5.4.1 Direct volume rendering 136
5.4.2 Surface model extraction 139
5.5 The 3D-Med medical visualization workstation 139
5.6 Chapter summary 140

6 Conclusion 143
6.1 Summary 143
6.2 Future work 145

Bibliography 147

A Published papers 169

Chapter 1

Introduction

The inspiration for this thesis is a desire to make it possible to do something useful with interactive 3D computer graphics. In this case the driving application is 3D medical visualization. Three dimensional scanning generates huge amounts of data. Visualizing these datasets requires graphics systems capable of processing the data. Interactive visualization raises the performance requirements for these graphics systems even higher.

Historically computer graphics has always been a very computationally demanding task resulting in great limitations on the achievable realism. That has changed much lately with the arrival of fast graphics processors for the PC. These PC based graphics processors primarily rely on texture mapping to achieve aesthetically pleasing interactive 3D computer graphics. Texture mapping is the process of applying two-dimensional images to three dimensional geometric objects in order to achieve an illusion of high complexity, suitable for computer games. Driven by the steady increase in computational power, greater attention on the geometric detail of three dimensional objects becomes possible.

The goal of this dissertation is to examine how we may improve the performance of graphics processors to allow greater geometric detail without sacrificing interactivity. In particular the focus is on computer graphics rendering algorithms and techniques for providing scalability in computer graphics architectures. The need for scalability comes from the desire to design and build an interactive 3D graphics system with the ability to handle very large datasets. Such a large dataset, possibly containing millions of polygons, cannot easily be handled by normal graphics workstations and PCs. This thesis will focus on the techniques and algorithms required to work with large datasets. Small datasets such as those found in common computer games are trivially handled by current graphics processors. A large dataset may be reduced by decimation to build a smaller dataset approxi-

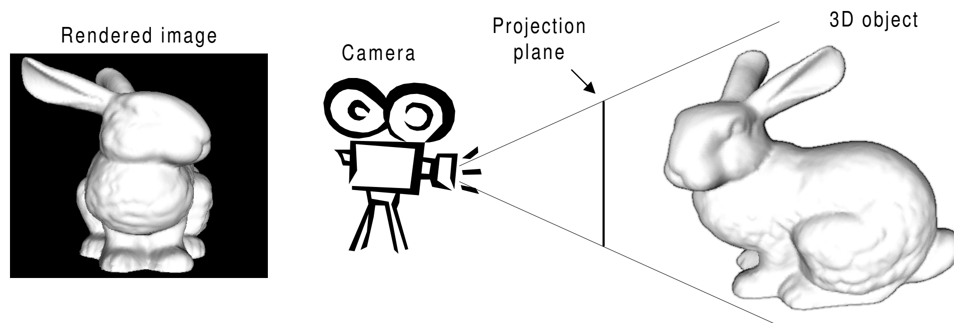

Figure 1.1: Example of perspective projection in 3D computer graphics. The Bunny 3D model on the right side is rendered to the computer screen plane. The 2D image to the left is the result of rendering the Bunny from the camera's point of view.

mating the original dataset, which would make it possible to view it on a standard graphics processor, but it would also result in a loss of detail. Scaling the performance of a computer graphics system to facilitate rendering of the large datasets without sacrificing detail is a better approach.

The process of generating an image in computer graphics is called *rendering* [63], figure 1.1 shows an example of how an object, the 3D model of The Bunny[1], is rendered on the screen using perspective projection, simulating how a real camera works. Rendering a dataset such as the Bunny is relatively straightforward using a sequential polygon renderer to render its 69,451 triangles. The triangles form a triangle mesh connecting 35,947 vertices. A simple sequential renderer will break the mesh into 69,451 individual triangles, resulting in a vertex count of 208,353 i.e. nearly 6 times as many as needed. This illustrates why a renderer must be carefully implemented in order to take advantage of special properties of the input data.

1.1 Parallel rendering – the next step

Parallel rendering is becoming very important in computer graphics. Historically parallel rendering has been used in massively parallel graphics supercomputers such as the Silicon Graphics RealityEngine [5] and InfiniteReality [158], as well as the UNC PixelFlow [155, 57], Pixel-Planes 5 [65] and many others. However these systems were prohibitively expensive especially when scaled to a high level of parallelism, with a typical system price tag of one million dollars.

[1] This is the Bunny model from the Stanford 3D scanning repository [208].

Computer animation for motion pictures is another area where parallel rendering is currently being used successfully. Since motion pictures do not require interactive rendering it becomes very simple to parallelize rendering as batch jobs. The computer animation studio simply employs a “rendering farm” of multiple workstations, possibly hundreds, each working on its own set of animation frames. This type of parallelism is often known as “embarrassingly parallel” because of its straightforward implementation. It is not suitable for interactive real-time rendering.

At the turn of the millennium we are witnessing a revolution in computational power in ordinary PCs, driven by the advances in semiconductor technology. This has up to now allowed commodity microprocessors such as the Pentium IV, Athlon, PowerPC and Alpha to scale to very high performance levels. The microprocessor computational power evolution was made possible by shrinking the dimensions of the transistor, which in turn makes it possible to fit many more transistors on a single chip *and* increase the clock frequency. Currently the minimum feature size of cutting-edge commercial CMOS semiconductor process technologies is 0.13 μ. Maximum clock frequency for a chip such as the Pentium IV CPU is 1.7 GHz, and the maximum number of transistors that can be fitted on a chip die as large as 400 mm^2 is more than 100 million transistors.

This progress in semiconductor technology development is not going to stop any time soon, as predicted by Gordon Moore’s historically proven law. Moore’s law predicts a two-fold increase in semiconductor transistor density every 18 months, i.e. exponential growth. Sources in the semiconductor industry claim that this development will continue for at least ten more years just by improving CMOS semiconductor technologies. Note that Moore’s law only claims technology improvements of the minimum transistor size in semiconductors, while improvements in computational power, complexity and speed are side-effects of putting improved technology to good use.

Ironically it is becoming increasingly more difficult to put all this on-chip real estate made available by technological advances to good use. Current microprocessors are using the added silicon area to embed larger cache memories, longer pipelines, multiple execution units and wider SIMD (Single Instruction Multiple Data) vector processing datapaths. A detailed description of these concepts in the context of the microarchitecture of the AMD K6-2 3D-Now! microprocessor is available in [206]. Current 3D graphics accelerators are also taking advantage of increased silicon area by integrating more parts and improved features of the rendering pipeline into a single chip. These parts were previously executed by microprocessors and/or several other ASICs (Application Specific Integrated Circuit). Some recent graphics accelerators [188, 187] are even integrating on-chip memories in order to overcome memory bandwidth problems.

Let us examine the hypothetical situation when an entire graphics rendering pipeline has been implemented in hardware on one chip, and the chip still has many nanoacres[2] of unused area left. Note that because of the numerous pins[3] required to interface with high performance memories and buses, a graphics ASIC design is often *pad-limited* so the die size cannot simply be reduced, leaving us with unused silicon area. Adding extra functionality to a chip to utilize the increasing chip area has limitations, though. Let us assume in this hypothetical case that better pipelining or "feature creep scalability" will not improve the speed of the design further. In order to scale to higher levels of performance (in computer graphics infinite rendering performance is preferred), parallelism is required. This can be achieved with a scalable graphics architecture, which will allow processing units of the graphics accelerator to be replicated across the entire chip. This thesis will show that replicating a single type of processing unit across the chip surface is not ideal, several different processor types are needed as well as an efficient communication network between them, forming a heterogeneous structure of processors, memories and communication. This is because a parallel renderer needs to redistribute temporary rendering data at least once at some point in the rendering pipeline in order to be scalable. Automatic load-balancing is also needed in a parallel renderer in order to keep all processors busy. While the parallel redistribution network provides a facility for internal load balancing, the input to the parallel renderer should also be partitioned so it can be distributed over the renderers to equalize the rendering workload.

Further, some level of programmability of the functional units is desirable as it will provide a tradeoff between having hardwired datapaths or general purpose microprocessors at each node. Several emerging new microarchitectures such as Intel's network processors and the Imagine [172] stream processor give an idea of what is possible when integrating multiple programmable parallel processing units on one chip. This is also evidenced by recent trends in the graphics accelerator market, where the new Nvidia GeForce 3 graphics processor provides programmable functional units for per-vertex and per-pixel processing, allowing the application writer to customize those parts of the pipeline by using simple stream processing scripts, rather than a hardwired pipeline. A promising emerging alternative method for achieving programmability is to use field programmable gate arrays (FPGAs).

To summarize, under the somewhat unlikely assumption that we cannot think of any extra functional features to add to the renderer *and* that the number of pipeline stages cannot be increased further, then one practical way to utilize any remaining chip area is to *parallelize*. This can be done by building a pipeline

[2] 1 nanoacre $\approx 4.0469\ mm^2$

[3] The Neon [140] chip has 824 pins (609 signal pins + power supply pins).

of parallel processing farms [62], each corresponding to a stage in the graphics pipeline. By using efficient communication for data redistribution between the worker processors in each stage, good scalability can be achieved. This is why parallel processing is likely to see a revival soon, but this time for embedded systems such as 3D graphics processors. By implementing such embedded parallel systems on a single chip, many of the communication problems limiting the scalability of parallel real-time graphics can be reduced.

The author's past experience with Transputer networks for rendering (ray-tracing) [87] by using networks containing up to 40 transputers were severely limited by the slow communication channels and all-to-one communication required to assemble the final image, making real-time rendering impossible although the slower ray-tracing algorithms did scale in performance from the parallelism. Today parallel real-time rendering looks far more promising as better communication paths are available.

1.2 Contribution of this thesis

This thesis is an attempt to describe and analyze useful methods and techniques for designing and implementing scalable parallel rendering architectures for 3D computer graphics. During the Ph.D. study a prototype 3D graphics architecture called Hybris has been developed. Hybris is also a prototype rendering system implementation testbed for experimental hardware/software codesign of graphics architectures, which can be tailored to many specific 3D graphics applications. Several variants of Hybris have been implemented during the Ph.D. study for both software and hardware. In software, both scanline and tile-based versions were implemented. Two different parallel software renderers were implemented, one based on functional parallelism and another based on data parallelism. Parallel software implementations for both single- and multi-processor Windows 2000 systems have been demonstrated. Efficient methods for partitioning and sorting were implemented to allow parallelism and efficient cache usage.

Working hardware/software codesign implementations of Hybris for standard cell design based ASIC (simulated) [71] and FPGA technologies [207] have been demonstrated. The hardware part of the design was carried out using manual translation of the software C source code to a synthesizable VHDL specification. The FPGA implementation was the first physically working hardware implementation of the Hybris renderer back-end. Currently the FPGA implementation shows great potential for future development, for example many of the parallel back-end architectural concepts demonstrated on the multiprocessor PC may be implemented. Additionally a fully functional VGA video output interface was implemented in

the FPGA, eliminating the need to transfer the final rendered frames back to the host PC.

For interfacing the graphics rendering architecture to applications an interface with a high abstraction level is required. Since common interfaces such as OpenGL enforce strict ordering of input data, they make data partitioning optimizations difficult, in effect relying on the calling application to do all front-end optimizations. The chosen solution is to rely on the ISO standard VRML 97 virtual reality modeling language [28]. VRML uses a scene graph programming model which does not assume anything about how an object is actually rendered. This abstraction allows object level optimizations to be made “behind the scenes”.

In Hybris a flexible VRML 97 scene graph engine with an EAI [135] Java interface and a custom object-oriented C++ interface has been implemented to allow flexible integration of the Hybris rendering technology into Java and C++ applications. The VRML abstraction allows Hybris to perform object level preprocessing and data partitioning optimizations. While lower level interfaces are also present internally in the Hybris architecture for the front-end and back-end graphics pipelines, these interfaces are not intended to be exposed to applications.

Finally the 3D-Med medical visualization workstation is an example of a complete application which uses the Hybris 3D computer graphics architecture for visualization of e.g. bone structures from CT-scans.

In summary figure 1.2 gives an overview of the various aspects of the work done in the Ph.D. study, however only some parts of the work will be covered in this thesis. The description of the design of the Hybris hybrid parallel graphics architecture is the main topic of this thesis. Hybrid parallel rendering offers some nice advantages; Good load balancing, low memory bandwidth requirements and good scalability.

1.3 Thesis chapter overview

This chapter has presented the motivation to design a scalable graphics architecture as well as an introduction to some of the concepts. The rest of this thesis is structured as follows:

Chapter 2 focuses on scalability in general with special focus on parallel rendering. State of the art in current commercial rendering architectures is covered. An introduction to recurring concepts in parallel rendering is given with an analysis of some of the available options.

Chapter 3 gives an in-depth view of the design concepts of the Hybris graphics architecture at an abstraction level slightly above the possible implementations. The potentially available parallelism of the architecture is described independent

of an actual implementation.

Chapter 4 goes forth by analyzing the possible implementation options for the architecture presented in chapter 3 by using codesign to map the designed architecture onto software and hardware components. We look at different base implementation platforms, such as single or dual CPU PC's for software implementations. ASIC and FPGA hardware accelerator implementations are also covered.

Chapter 5 discusses how to interface with the application that uses the graphics architecture. An implementation of VRML 97 is used as a high-level interface to the architecture. As an example of an application which uses the architecture, the 3D-Med medical visualization workstation is presented.

Finally, Chapter 6 summarizes the work with conclusions and suggestions for future work.

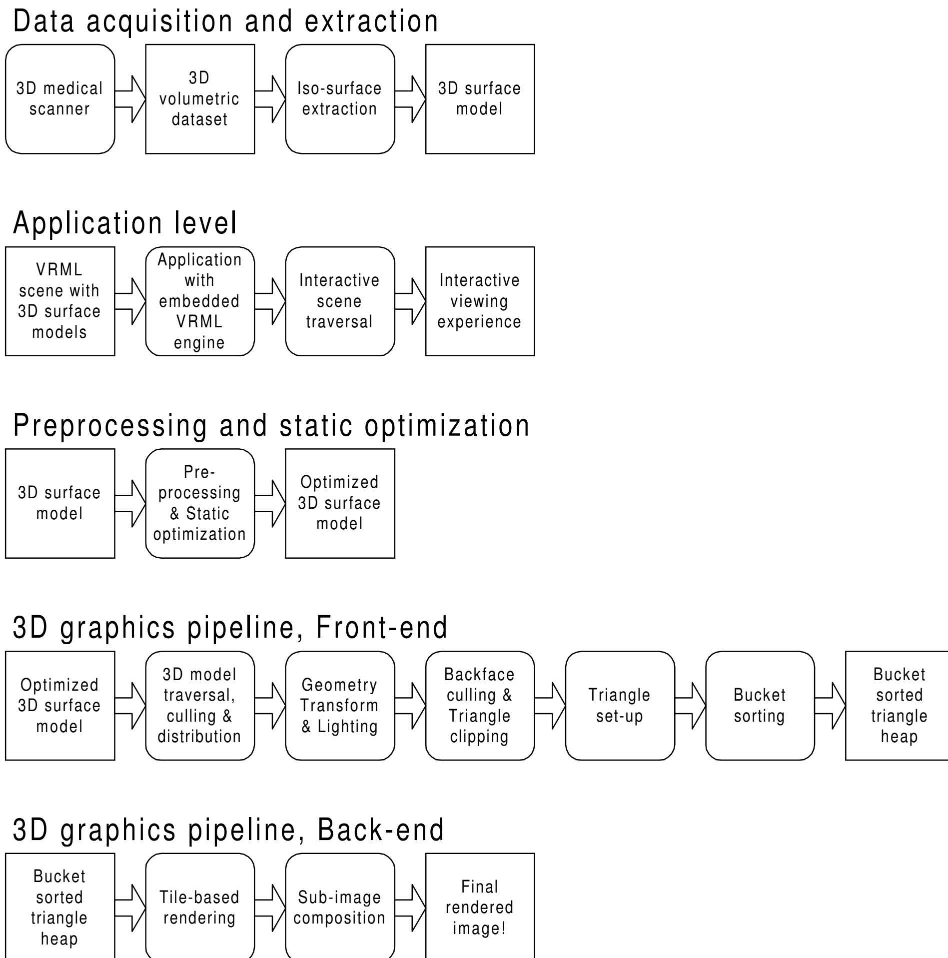

Figure 1.2: Overview of the Hybris 3D computer graphics architecture, including extensions for interactive virtual reality based on VRML as well as extensions to support the 3D-Med medical 3D visualization workstation.

Chapter 2

Parallel Rendering and Scalability

This chapter focuses on parallel rendering in general but with special focus on scalability. State of the art in current commercial rendering architectures is discussed, including recent PC based architectures. An introduction to recurring concepts in parallel rendering is given with an analysis of some of the available options. Architectural concepts and design methods for scalability of parallel rendering architectures on the system-architecture level will also be discussed.

2.1 Scalable 3D graphics architectures

In computer graphics there is a need for a scalable solution where the rendering performance increases as more parallel hardware processing units are added. Several scalable 3D graphics architectures have been published, e.g. the PixelFlow [155] and Pomegranate [51]. With the advent of highly integrated ASIC technologies and fast interconnects, scalable architectures are gaining interest.

Before parallelism can be applied to rendering in 3D computer graphics some methods for achieving scalability is needed. Definition of scalability: The ability of a system to take advantage of additional processing units. This usually implies that the system can take advantage of parallel execution. Two types of scalability can be considered; hardware scalability and software scalability. Hardware (i.e. performance) scalability is the ability to gain higher levels of performance on a fixed size problem. Software (i.e. data) scalability is the ability to encompass larger size problems, such as added model complexity or framebuffer resolution.

In popular terms software scalability means that a system can take advantage of faster/better/more complex hardware to improve the overall quality of the work

performed. This definition is often used with recent PC graphics hardware (e.g. Nvidia), where the intent is to make proper use of faster graphics hardware with existing software, i.e. the software should detect that powerful hardware is used and attempt to utilize it by increasing the quality of the task. In a computer graphics application such as a game this can mean using more detailed geometry in the 3D models, larger textures, additional texture layers, better illumination models, etc. This definition of scalability refers to the scalability of the application with respect to dynamically adapting itself to different generations of hardware.

As a combined definition, scalability refers to the ability of an algorithm to be able to efficiently utilize multiple parallel processors.

A simple form of scalability is when a software renderer is running on a multiprocessor computer, for example a PC configured in an SMP (Symmetric Multi Processing) configuration with two CPUs using shared main memory. The PC must be running an operating system such as Linux or Windows NT/2000 which supports multiple processors, processes and threads. In an SMP system the main memory is shared between all processors and a process can have two concurrently running threads with a speedup scaling factor of around 2 (the actual scaling factor depends on the main memory access patterns and synchronization of the threads). However hardware performance scalability is limited by the bandwidth of the shared main memory, because adding more processors does not increase the main memory bandwidth. This is why highly scalable multiprocessors use a distributed memory architecture where the processor/memory nodes are interconnected using a switched high-speed network. Unfortunately such an architecture may limit the communication bandwidth between the nodes, if the network is slower than the memory. For this reason scalable architectures often try to *partition* the workload to minimize the communication between processor nodes, in order to allow high scalability performance gains with a distributed processing architecture.

From a hardware point of view, scalability is the ability to speed up a system by adding more hardware components to the system.

2.1.1 General purpose parallel computing system architectures

This section presents a quick overview of some popular general purpose parallel computing system architectures.

SMP – Threads

Microsoft Windows NT and Windows 2000 supports parallel execution of multiple threads on a multiprocessor host using the Win32 thread API. For modern UNIX hosts Posix `pthreads` provides a similar thread API. Threads are useful

in shared memory architectures and provides a simple programming model where all data is shared in main memory. SMP (Symmetric Multi Processing) systems are characterized by having a single main memory subsystem to which all processing nodes are directly connected. This allows simple and fast communication between threads, but the processors cannot run at full speed when they must time-share the access to main memory. SMP using two CPUs is available in many PCs and workstations. Larger configurations are found in some server systems.

Distributed multi processing

In distributed multi processing architectures, each processor has its own local memory, which is not directly accessible by other processors. The processors are linked via a communication network, and all data must be explicitly distributed among the processors. This type of architecture allows each processor to run at full speed once it has all the data it needs. Unfortunately finding a good data distribution with minimal communication needs can be difficult, depending on the problem. Today distributed multi processing systems are often implemented by connecting a huge number of standard PCs in a high-speed switched network.

MPI – Message Passing Interface

MPI (Message Passing Interface) is an international standard API for communication between processing nodes in a distributed multi processing environment. MPI works by allowing processes to communicate by message passing, and allows process synchronization using barriers. MPI helps by hiding low-level synchronization and communication from the programmer, and should help make the distributed processing architectures more accessible and easier to use.

NUMA

NUMA (Non Uniform Memory Architecture) is a compromise between shared memory and distributed memory systems, which is used in many modern parallel supercomputers. NUMA allows the system to be partitioned into groups of SMP systems connected in a high speed network. Often special hardware support for thread programming is implemented in order for the application to assume a shared memory thread programming model.

Note that this hybrid general purpose architecture shares many similarities with the hybrid parallel graphics architectures which will be discussed later. Another interesting note is that SMP systems based on CPUs with large internal caches may be considered to be a NUMA system, as the caches will work much like the local memories in a distributed multi processing environment.

2.1.2 Scalability of current PC-based 3D graphics processors

Most manufacturers of low-cost high-performance PC-based 3D graphics processors have been reluctant to discuss the microarchitectures of their implementations. Only superficial details such as the amount of graphics memory, clock frequency, peak fill-rate, peak triangle-rate, 3D graphics feature set and vague marketing lingo about the actual implementations are available. The only exception to this is a description of Digital's Neon single-chip graphics accelerator [140, 141], which was published after their design project was cancelled. The Neon was actually made for Digital Alpha workstations using a 64-bit PCI bus [175] rather than the AGP interface [104], but otherwise it had features similar to many PC 3D graphics cards. Neon relied heavily on the high performance of Alpha workstations to create a well balanced system. Despite all this the Neon is not directly scalable without a redesign of the chip, but it features a novel memory subsystem using an 8 way memory controller to utilize 8 separate 32 bit SDRAM[1] memory controllers to gain high memory bandwidth and allow multiple simultaneous memory operations. Other publications related to the spin-off from the Neon design project include [139, 142, 144].

Scalability is not common in PC graphics accelerators because of very tight cost limitations favoring single chip implementations, still some attempts have been made to implement scalability. This is mainly done to provide the option for a faster graphics system to those willing to pay, or to squeeze more performance from a dated technology. In the following we take a look at some representative PC graphics architectures and their scalability options.

3dfx – SLI

Some scalable designs use multiple graphics cards by interleaving the framebuffer on a scanline basis. The interleaved graphics method renders even scanlines on one card and odd on the other. All scene geometry is broadcast to both graphics cards. A well known example of this configuration is the Voodoo 2 SLI (Scan Line Interleaved) configuration of two 3dfx Voodoo 2 3D graphics PCI cards. In the SLI configuration the two PCI boards are connected via a ribbon cable to act as one board. The ribbon cable is used to send rendered pixels from the slave board to the master board, which assembles the odd and even scanlines to create the video image. The SLI configuration improves performance by doubling the pixel drawing speed, as two independent memory buses are used to double the pixel bandwidth. Yet, since the geometry is sent to both processors, the geometry processing speed is not improved. This makes the SLI approach somewhat inefficient.

[1] SDRAM: Synchronous Dynamic Random Access Memory [147].

Since the Voodoo 2 boards, 3dfx finally released the VSA-100 (Voodoo Scalable Architecture) graphics chip in 2000. The VSA-100 is essentially a single chip implementation of the Voodoo 3 (which was a single chip version of the Voodoo 2 + a 2D graphics core) combined with the SLI capabilities of the Voodoo 2 chipset. This allows VSA-100 to employ board level scanline interleaving using up to 32 VSA chips. Each chip needs it own local 32 MB framebuffer memory. The Voodoo 4 board uses one VSA-100 chip, while the Voodoo 5 5500 uses 2, and the Voodoo 5 6000 uses 4 VSA chips. The VSA-100 based graphics boards basically distribute the workload like a Voodoo 2 SLI system by broadcasting data to all processors, duplicating the geometry setup calculations on all chips. However the real purpose for the parallelism is fast supersampling anti-aliasing which requires four VSA-100 chips to work. A high-performance configuration called the AAlchemy, produced by Quantum3D, uses up to 32 VSA-100 chips in parallel to render fast antialiased 3D graphics. Of the Voodoo 5 boards, only the 5500 version with 2 processors made it to the market. The 4 processor 6000 version needed for full quality antialiasing required an external power supply and was never released on the PC market. Unfortunately 3dfx was liquidated and acquired by Nvidia in early 2001, so no further development of these products may be seen.

ATI – MAXX

Another example of scalability is the ATI Rage Fury MAXX card which uses two Rage 128 Pro chips in an AFR (Alternate Frame Rendering) configuration. With AFR, one chip renders even frames while the other chip renders odd frames. Each chip processes triangle setup for its own frame without waiting for the other chip, making AFR more efficient than 3dfx's SLI technique. The AFR method is also nicely load balanced since the frame-to-frame coherence is usually quite good in interactive 3D systems. However because each chip needs data for two different frames, the software driver needs to completely store the data needed for at least one frame while the other frame is being rendered. This introduces pipelining latency in the system. Another drawback in the hardware design is that the graphics board requires two independent framebuffers, one for each graphics chip, doubling the memory usage from 32 Mb to 64 Mb. Additionally the bandwidth over the AGP bus is critical since the design effectively makes both graphics chips available on the AGP bus, both needing different data simultaneously. ATI's newest graphics accelerator, the Radeon [159], is not available in an AFR configuration, presumably because of driver problems, as the MAXX configuration with two devices on the AGP interface does not work properly with the Windows 2000 operating system. The Radeon implements other nice features such as a geometry processor for transformation, lighting and clipping, as well as a hierarchical z-buffer [75] to

improve visibility culling.

According to user testing[2] the latency introduced by the AFR technique is not significant enough to influence the interactive gameplay of the computer game Quake 3 Arena. This is an important observation relevant for any system which relies on increased latency to improve performance (Such as the Hybris architecture presented later in this thesis).

PGC

Metabyte/Wicked 3D's PGC (Parallel Graphics Configuration) technique uses two graphics boards in parallel to work on different sections of a frame. One board renders the top half of the frame, while the other board renders the bottom half of the frame. The PGC system includes an external hardware component that collects the analog video outputs from both graphics boards (the upper and lower regions) and integrates them into a single image video signal for the monitor. PGC allows two slightly modified standard graphics boards to be used in parallel by this technique. The analog video merging technique may introduce image tearing because of difficulties with video timing and DAC calibration. A digital video merger would not suffer from these problems. Since PGC statically divides the image in two halves, poor load balancing may occur e.g. if the rendered scene is more detailed in the lower than the upper half (flight simulators have this behaviour).

3Dlabs – Parascale

The 3Dlabs Wildcat II 5110 dual pipeline graphics accelerator, which was introduced early 2001, is an example of an AGP Pro based graphics accelerator. AGP Pro is simply an AGP interface for workstation PCs which allows large cards with a power consumption up to 110W, see section 4.2.

Wildcat II is an implementation of the 3Dlabs Parascale [1] scalable graphics architecture, which allows a graphics system to use up to four Geometry Accelerator ASICs and up to four Rasterization Engine ASICs, scaling the performance up to four times. The dual pipeline Wildcat II should supposedly reach a performance of 12 Mtriangles/sec, while a quad pipeline implementation should reach 20 Mtriangles/sec, according to marketing pamphlets on `http://www.3dlabs.com`.

Parascale is similar to the 3Dlabs Jetstream architecture, of which some information was given in the keynote presentation at the 1999 workshop on graphics hardware [225]. The Jetstream architecture is based on continued development of the GLINT Delta and Gamma [224] front-end geometry processors. The Jetstream

[2] Review at `http://www.tomshardware.com`

architecture works by dividing the scene into interleaved strips of scanlines, allowing better texture map cache coherency compared to scanline interleaving. The architecture utilizes a rendering ASIC and a geometry processor ASIC. The geometry processor ASIC is a specialized active AGP to AGP bridge placed between the host's AGP port and the rendering ASIC. Any transmitted vertex data is processed by the geometry processor. Using two output AGP ports, the chip is able to divide the vertex data stream in two streams, one to be processed locally and sent to a rendering ASIC connected to port one, and one to be passed on to the next geometry ASIC connected to AGP port two. This way the architecture is scalable until the bottleneck becomes the AGP input bandwidth for the first ASIC in the chain.

PowerVR – Naomi II

PowerVR [188] is an innovative scalable tile-based rendering architecture for low-cost PCs, TV game consoles and high-performance arcade game systems. It is manufactured by STMicroelectronics and designed by Imagination Technology.

The PowerVR architecture is used in a scaled configuration for the recently announced Sega Naomi II arcade game system, where two tile rendering ASICs are used along with one geometry co-processor ASIC which handles floating point calculations for transformation, lighting and clipping, offloading the system's CPU. The configuration is able to render 10 Mtriangles/sec sustained throughput in real game applications.

A low cost single chip configuration of the PowerVR architecture is used in the Sega Dreamcast TV game console, where low cost is the main limiting design factor.

For PCs PowerVR has previously been implemented in several less successful designs, but lately (March 2001) the PowerVR Kyro II graphics accelerator chip was announced, showing new high performance levels for a tile based renderer. Benchmarks[3] show that in certain real-world circumstances the Kyro II is able to outperform even the Nvidia GeForce 2 Ultra. This is remarkable, as the Kyro II is clocked at 175 MHz, uses 128 bits wide 175 MHz SDR SDRAM memory (1.4 Gbytes/s peak bandwidth) and relies on the host PC to perform geometry calculations for transformation, lighting and clipping. In comparison the GeForce 2 Ultra includes a hardwired geometry pipeline, is clocked at 250 MHz and uses 128 bits wide 230 MHz DDR SDRAM memory (Double Data Rate, 7.4 Gbytes/s peak bandwidth).

[3]Benchmarks at `http://www.anandtech.com`

GigaPixel

The key feature of GigaPixel's Giga3D architecture [187] is that it implements tile-based rendering, much like the PowerVR architecture. The key benefit of this type of rendering is that tiles of reasonable size can be completely rendered using on-chip memory, without having to access external SDRAM using read-modify-write cycles. Using the tiling architecture it is possible to perform efficient visibility-culling, which removes graphics primitives and pixels which do not contribute to the final image. Finally the tiling architecture allows very efficient implementation of anti-aliasing using jittered supersampling to produce a high image quality. Since Giga3D is able to render using small on-chip memories, it achieves an image quality equivalent to a classical architecture using three to ten times lower external memory bandwidth.

The GigaPixel Giga3D architecture never resulted in any actual products other than the prototype GP-1 which was successfully demonstrated at Comdex 99. In late 2000 GigaPixel was acquired by 3dfx, supposedly to merge Giga3D IP into upcoming Voodoo graphics accelerators. However in early 2001 3dfx succumbed to financial difficulties and was sold to Nvidia. Thus Nvidia now owns the IP of two of its former competitors, 3dfx and GigaPixel.

Nvidia

While Nvidia currently produces the most complex PC graphics accelerators (in terms of special effects features implemented), they do not produce any directly scalable graphics architectures. The latest GeForce 3 graphics accelerator from Nvidia was introduced in March 2001. The GeForce 3 is implemented on a single 0.18 μ chip using 57 million transistors. The GeForce series of graphics accelerators implement a hardwired geometry processor for transformation and lighting calculations. The newest GeForce 3 extends this geometry processor with a simple programming interface to allowing customized vertex stream processing for alternative lighting models and transformations. Nvidia seems to employ internal fine-grained pixel-level parallelism possibly with CPU-like caches, and relies on very fast memory technologies to solve the memory bottleneck problems. The GeForce 3 requires 64 MB memory organized in 4 banks of 32 bit wide DDR SDRAM clocked at 230 MHz (effectively 460 MHz) to get a peak memory bandwidth of 7.4 GB/s. The memory organization with 4 banks is very similar to the Neon's [140] crossbar memory controller. Since Nvidia now owns both 3dfx and GigaPixel technology, they may want to explore other design options.

2.1.3 Summary

While current PC based 3D graphics architectures are using scalable concepts to some degree, supercomputer 3D graphics systems have used other more elaborate and expensive scalable architectures, e.g. the PixelFlow [57] and the SGI Infinite-Reality [158]. Note that current ASIC technology is able to integrate much of these past systems onto a single chip, e.g. the PixelFlow is currently being implemented in the FUZION chip [137], and the datapaths of the InfiniteReality has inspired the Nvidia GeForce series of graphics processors.

While most PC based 3D graphics accelerators are using a poorly scalable computation model very similar to the PC itself (hardwired CPU with external memory) this is beginning to change. Some nice examples are the PowerVR and Giga3D which use an on-chip pixel buffer to render one tile of pixels at a time. Other examples such as the 3Dlabs Jetstream architecture is an attempt to fit an architecture similar to the SGI InfiniteReality on a single board. Finally Nvidia is using a crossbar memory architecture for the GeForce 3 to squeeze more performance from the hardwired CPU with external memory approach. The GeForce 3 even features some limited programmability in form of downloadable vertex-shader and pixel-shader [138] programs, reducing the difference between CPU's and graphics processors.

2.2 Parallel rendering concepts

Rendering a complex scene is so computationally intensive that it requires billions of floating-point, fixed-point and integer operations for each frame. Real-time interactive rendering places even higher demands on the rendering system as a minimum framerate has to be maintained. Note that real-time rendering usually is a soft-real-time problem where the response time may be stretched slightly without major problems. Most real-time implementations do not require a hard-real-time guaranteed response time to function (although that would be nice). In order to maintain the processing power needed for interactive rendering of complex scenes, we can rely on Moore's law and wait until microprocessors and graphics accelerators become fast enough to solve the problem. Alternatively, if the needed processing power is wanted now, the only option left is parallel processing, in this case *parallel rendering*. Some concepts which are important for parallel rendering will be discussed in the following sections.

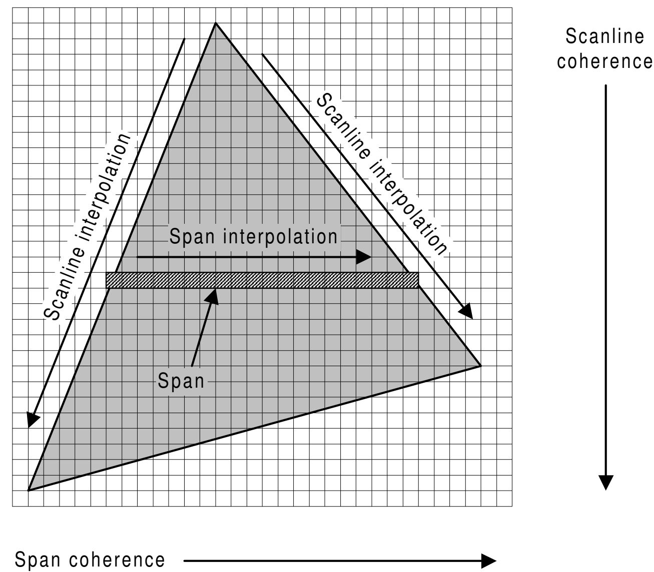

Figure 2.1: Spatial coherence in a screen space region (32x32 tile).

2.2.1 Coherence

The term *coherence* is used in computer graphics to describe that features nearby in space or time have similar properties [37, 215]. Coherence is useful for reducing the computational requirements of rendering, by allowing incremental processing of data by a sequential algorithm. Because of this it is important for a parallel rendering algorithm to *preserve* coherence, or it will suffer from computational overhead. Several types of coherence can be identified in rendering:

Spatial coherence refers to the property that pixels tend to have values similar to their neighbors both in horizontal and vertical directions, stepping from one scanline to the next (scanline coherence), and between pixels within a span (span coherence). Figure 2.1 illustrates these types of spatial coherence. A sequential rendering algorithm can use these kinds of coherence to reduce computation costs while interpolating parameters between triangle vertices during scan conversion. A popular incremental linear interpolation algorithm is forward differencing or DDA (Digital Differential Analyzer) [58]. In a parallel renderer which partitions the screen into regions, coherence may be forfeited at region boundaries. Because of this, triangles which overlap several regions may cause a computational overhead in a parallel renderer.

Temporal coherence is based on the observation that consecutive frames in an animation or interactive application tend to be similar. This may be useful for

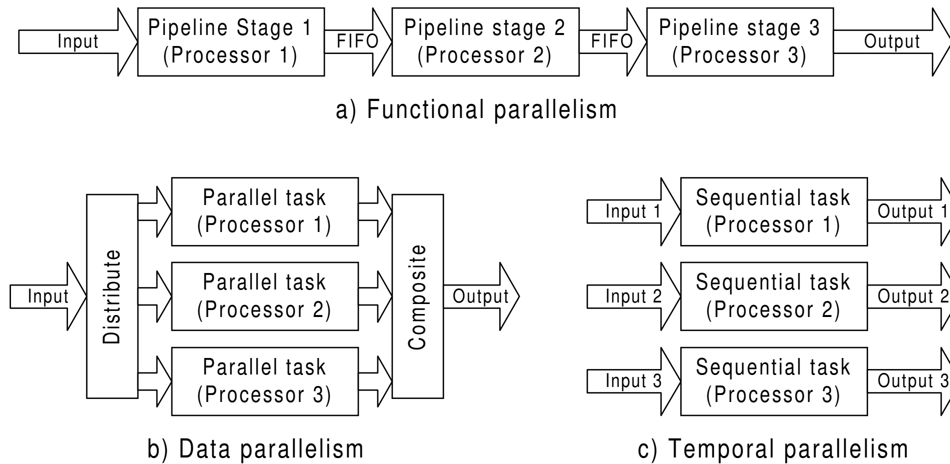

Figure 2.2: A process parallelized over three processors by using three different types of parallelism. a) Functional parallelism. b) Data parallelism. c) Temporal parallelism.

predicting workloads in a parallel renderer in order to improve load balancing. A good example of temporal coherence is MPEG video compression which relies on incremental frame to frame encoding and motion compensation to achieve its high compression ratios.

Data coherence is a more abstract term, but can for example be described as the tendency for multiple triangles or other data to contribute to nearby pixel regions. Data coherence is improved by locally caching data and *reusing* the cached data. It is related to both spatial and temporal coherence as multiple triangles contributing to the same screen region can be grouped together to improve communication efficiency and usage of cached pixels. These properties are important for the efficiency of a parallel renderer.

Statistical studies on workload characteristics examining different forms of coherence in various rendering tasks were published in [150, 32].

2.2.2 Parallelism in rendering

Many different types of parallelism can be exploited in rendering. These are functional parallelism, data parallelism and temporal parallelism. Figure 2.2 presents an overview. These basic types of parallelism can be used alone or combined into hybrid systems to exploit multiple forms of parallelism at once.

Figure 2.3: Standard rendering pipeline.

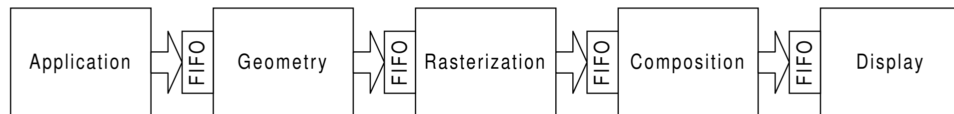

Figure 2.4: A pipelined parallel renderer using FIFOs to queue data between stages.

Functional parallelism – Pipelining

In computer graphics 3D surface rendering can easily be expressed as a pipeline, where triangles are fed into the pipeline and data is serially passed from one processing unit to the next in the data path, and pixels are produced at the end. The standard rendering pipeline (figure 2.3) is an obvious candidate for functional parallelism as each individual stage may be mapped to individual processors. Several early commercial hardware renderers [5, 45] were based on functional parallelism by physically arranging programmable floating-point microprocessors in a pipeline, and mapping different stages of the rendering pipeline to different microprocessors (or "Geometry Engines").

Functional parallelism has some major drawbacks, though. The overall speed of the pipeline is limited by its slowest stage, and it is susceptible to pipeline stalls. Most pipelines use FIFO[4] queues to balance the pipeline loads by queuing data between pipeline stages (figure 2.4), allowing upstream stages to continue working while a downstream stage is busy. A pipeline stall occurs when a pipeline stage is using more time to complete its task than the others, for example when a rasterizer is busy filling a huge triangle. Small pipeline stalls can be avoided by using FIFO queues to balance the load, provided that the processed data stream provides the pipeline with an even workload distribution averaged over time.

The level of parallelism achieved by functional parallelism is proportional to the number of pipeline stages. Functional parallelism does not scale well, since the pipeline has to be redesigned for a different number of pipeline stages each time the system is scaled. To achieve higher levels of performance, an alternate strategy is required.

[4] First-In-First-Out

Data parallelism

While it may be simple to perform rendering using a single data stream through multiple specialized pipelined processors, it may be preferable to split the load into several data streams. This allows us to process multiple data items concurrently by replicating a number of identical processing units. Data parallelism is necessary to build scalable renderers because large numbers of processors can be utilized, making massively parallel systems possible. It is also possible to build different versions of a data parallel system, scaling the performance levels to match the required tasks simply by varying the number of processing elements.

Data parallelism can be implemented in rendering in many different ways. Two basic classes of data parallelism in rendering may be conceived, *object parallelism* and *image parallelism*.

Object-parallel rendering refers to an architecture which splits the rendering workload so each processor works independently on individual geometric objects in a scene.

Image-parallel rendering refers to partitioning the processing of the pixels for the final image. Each processor is responsible for its own set of pixels, and works independently of the other processors.

Object parallelism and image parallelism can be combined to perform object parallel computations at the *front-end* of the rendering pipeline, and image parallelism can be exploited at the *back-end* of the rendering pipeline. Load balancing between the front-end and back-end must be handled. The workload must also be balanced between the individual workers in each stage. Communication patterns between front-end and back-end are crucial for the scalability of such a system, which will be discussed later. Functional and data parallelism can also be combined to gain additional speed, e.g. by building a pipeline of processor farms. Theory behind pipelined processor farms (PPF) is covered in the recently published book [62]. Figure 2.5 shows how a pipeline of parallel processor farms can be used to parallelize the 3D graphics pipeline. Note that the data communication between the pipeline stages is *not* simple. Data redistribution or sorting is required between some of the pipeline stages to implement a parallel renderer.

Temporal parallelism

Temporal parallelism works by rendering several different frames of an animation concurrently. Batch renderers, such as those used for rendering 3D animated special effects for Hollywood movies, typically use temporal parallelism to distribute the workload over a “rendering farm” of workstations, each rendering their own set

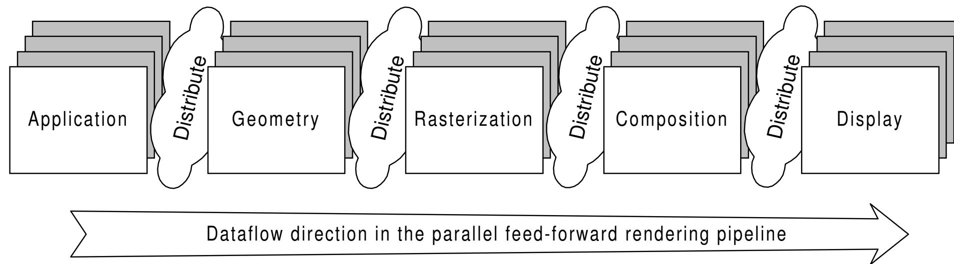

Figure 2.5: Parallel feed-forward rendering pipeline.

of animation frames. This is the only type of parallel rendering that is considered to be embarrassingly parallel, as each worker process executes the entire rendering pipeline implemented as a sequential 3D renderer.

When used for interactive real-time rendering, temporal parallelism can exploit the latency in the rendering pipeline to overlap rendering of two consecutive frames by using two separate graphics pipelines. This technique requires that the frame rate is high enough to hide the effect of latency. Note that in this case the achieved parallelism can also be thought of as high level pipelining.

2.3 Parallel rendering as a sorting problem

Molnar [154] classified parallel rendering architectures by treating the task as a sorting problem, with three possible classifications: *Sort-first*, *Sort-middle* and *Sort-last*. These three classes indicate possible locations in the graphics pipeline where work redistribution must be done in order to implement a scalable parallel renderer. Sort-first redistributes work before screen-space transformation, sort-middle redistributes work after transformation and sort-last redistributes partial results after rendering to build a complete image. Figure 2.6 illustrates these three possible locations where redistribution may take place in a parallel rendering architecture.

Note that it is also possible to perform more than one redistribution in a parallel renderer, a possibility which is gaining popularity in current graphics architecture research such as [51, 195, 167] as well as this thesis. Multiple redistribution hybrid parallel architectures are more feasible to build today as the added communication overhead is more manageable with the advent of high-speed communication links and crossbar switches, as well as the recent possibility to integrate significant amounts of on-chip memories in modern ASICs as well as larger cache sizes in CPUs. Going a step further, all this may be integrated on one graphics-system-on-a-chip (SoC).

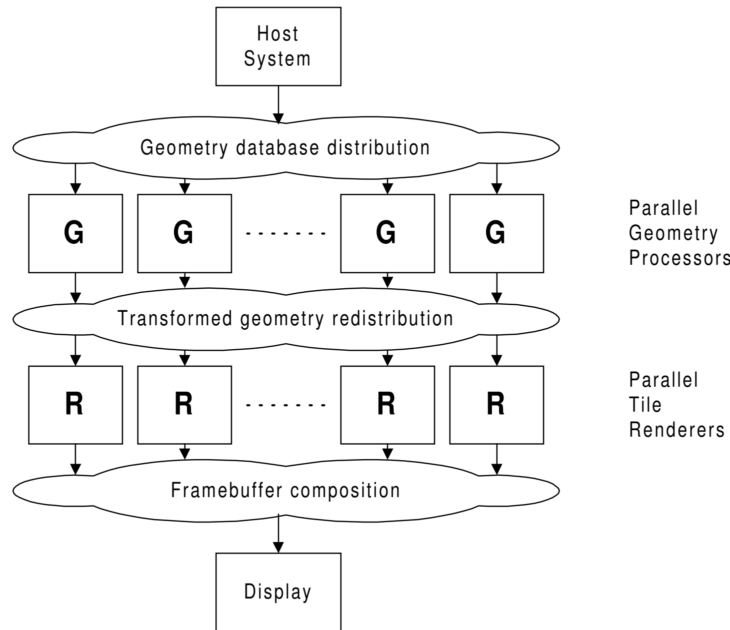

Figure 2.6: Generic parallelism with redistribution.

2.3.1 Sort-first

A sort-first parallel rendering algorithm redistributes the raw triangles before they are transformed. Figure 2.7 shows a conceptual example of a sort-first parallel renderer which subdivides the framebuffer into four equal sized regions. In order to redistribute primitives *before* transformation, the sort-first renderer must first determine which three dimensional sub-spaces will correspond to the screen regions after transformation and screen-space projection. A popular method for implementing sort-first is hierarchical pre-transformation of triangles to determine which region it falls into. This is a non-trivial problem, but once the redistribution has been completed the remaining parallel rendering task is trivially implemented. Sort-first is quite suitable for parallelizing commodity PC graphics accelerators e.g. in a large display using multiple PC's [196]. A weakness is poor load balancing as triangles may concentrate into a few of the renderers leaving others idle. This problem can be overcome by using smaller regions for better load balancing of the renderers, but it comes at an increased sorting cost and also reduces the available coherence.

If we abandon the idea of completely solving the parallel processing redistribution problem for sort-first, several other sorting options for sort-first are feasible, e.g. we might sort the primitives according to size, spatial or temporal proxim-

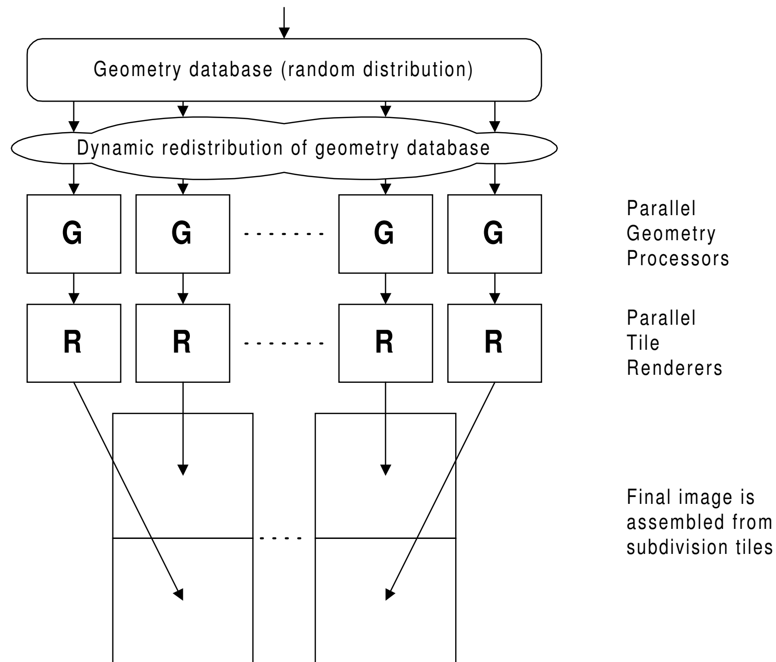

Figure 2.7: Sort-first parallelism using tiled image subdivision.

ity, surface properties, etc. Hybris uses sort-first to redistribute triangles according to object-space spatial groupings which may not necessarily coinciding with any screen-space subdivision. While this does not solve the sorting problem completely, it can be applied as a preprocessing step to help other types of sorting. Together this can form a *hybrid* sorting architecture. An example of a hybrid parallel graphics architecture primarily based on sort-first and sort-last is given in [195].

Further research on various screen-space subdivision schemes suitable for sort-first parallel rendering can be found in [161, 162]. Other research focuses on the use of sort-first to enable parallel rendering using standard PC 3D graphics hardware [27, 94, 95]. As multiple PCs are used in this case, the possibility of a parallel graphics interface is examined in [99, 101].

2.3.2 Sort-middle

The sort-middle parallel rendering algorithm redistributes the triangles after they have been transformed, but before they are rasterized. Figure 2.8 shows a conceptual overview of a sort-middle parallel renderer using a tiled subdivision of the framebuffer into four equal sized regions or tiles. Tiling in sort-middle architectures can use either a few large or many small tiles, depending on architectural

choices. The sort-middle redistribution over multiple tiles is often referred to as *bucket sorting, binning, blocking* or *chunking*, which is necessary to determine which tile(s) a triangle covers. Many sort-middle architectures (e.g. SGI Infinite-Reality) ignore bucket sorting and simply broadcasts all triangles to all tiles, resulting in poor scalability as all processors must process all triangles.

When implementing bucket-sorting there is a choice between physically having a processor per region or using virtual processors. The *processor per region* version of sort-middle provides a simple redistribution mechanism implementable using a broadcast or crossbar system to distribute triangles to the tile processors, requiring no buffering other than perhaps some FIFOs. The *virtual processor* version of sort-middle distributes the workload across more tiles than there are tile processors. This approach requires extensive buffering of the entire scene in a bucket sorted triangle heap, but allows an implementation to use a few fast tile renderers, each rendering into a *virtual local framebuffer*. Load balancing between the tile renderer processors works best with the last method, as the workload can be dynamically balanced. In comparison the processor per region sort-middle architectures depend on an even distribution of triangles to achieve good load balancing, although this can be improved by interleaving the framebuffer but for the cost of less efficient bucket sorting (i.e. broadcast).

While no sorting is required in the geometry processors, some data distribution is required. Round robin distribution of individual triangles over the geometry processors is used in [38] with good results. The sort-middle data redistribution itself is fairly straightforward, as triangles have been transformed to screen space coordinates. Since sort-middle can be implemented in many different ways, some simple forms of sort-middle have become quite popular in commercial graphics architectures. Interleaved framebuffer architectures, e.g. scan-line interleaving, are based on sort-middle where the "sorting" has been simplified to a broadcast to all worker processors. While this architecture is easy to implement with a very simple broadcast stage, poor load-balancing and poor utilization of worker processors may occur if the rendering process is not fill-limited. An example of a sort-middle architecture using broadcast to interleaved renderers is the Silicon Graphics GTX 3D graphics subsystem [63] shown in figure 2.9. This system is generally considered to be a sort-middle parallel architecture although it is not fully parallel, as a single processor pipeline handles the data redistribution. Some methods for scaling the pipelined front-end is by using SIMD-like data parallelism [81] or even better by using MIMD data parallelism [158]. Similar interleaved architectures are found in [4, 5, 45]. In general the interleaved architectures perform best for scenes with large triangles as a very small triangle will only be processed by a few of the renderers while the others are idle because of the broadcast mechanism.

Non-interleaved tiled sort-middle architectures are found in [65] as well as

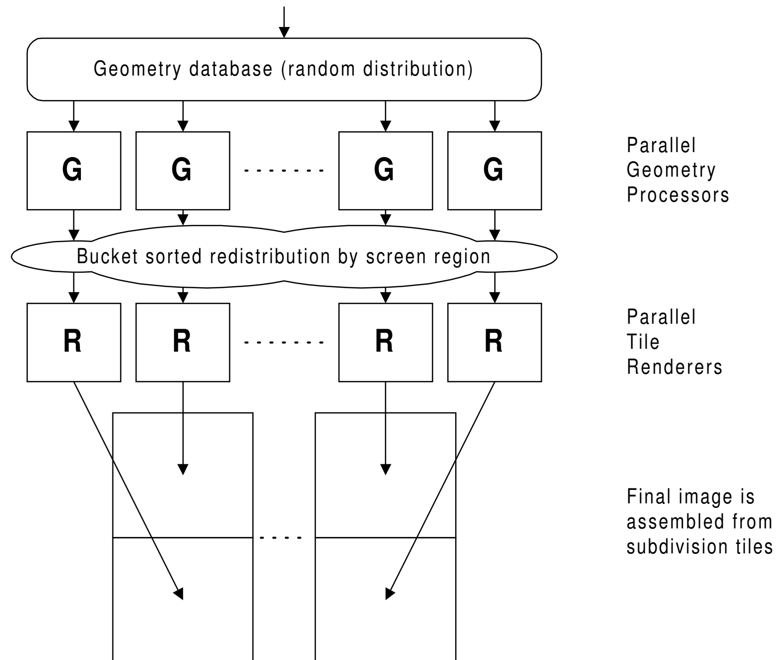

Figure 2.8: Sort-middle parallelism using tiled image subdivision.

[234] and [38]. An interesting sort-middle architecture is the SAGE processor-per-pixel scanline renderer [68]. A two-step redistribution scheme is used for sort-middle in [52, 53, 54] to improve scalability. Recent tile-based graphics architectures suitable for sort-middle parallelism include the PowerVR [188] architecture in its arcade game parallel configuration with one geometry processor chip and two tile processor chips. The 3Dlabs Wildcat II [1] is another an example of a sort-middle architecture.

2.3.3 Sort-last

In sort-last, redistribution of pixels (not triangles) is done after rasterization. Figure 2.10 gives a conceptual overview of a sort-last parallel renderer using image composition of four full-frame image layer tiles. The advantage of sort-last is that scalability is quite simple since each triangle is only processed by one front-end and one back-end processor.

Image composition requires a high-speed image composition network which composites each final pixel from pixel values from all the rendering processors. To do this correctly the composition processor needs pixel values such as colour, transparency *and* depth. Some methods for image composition of images with

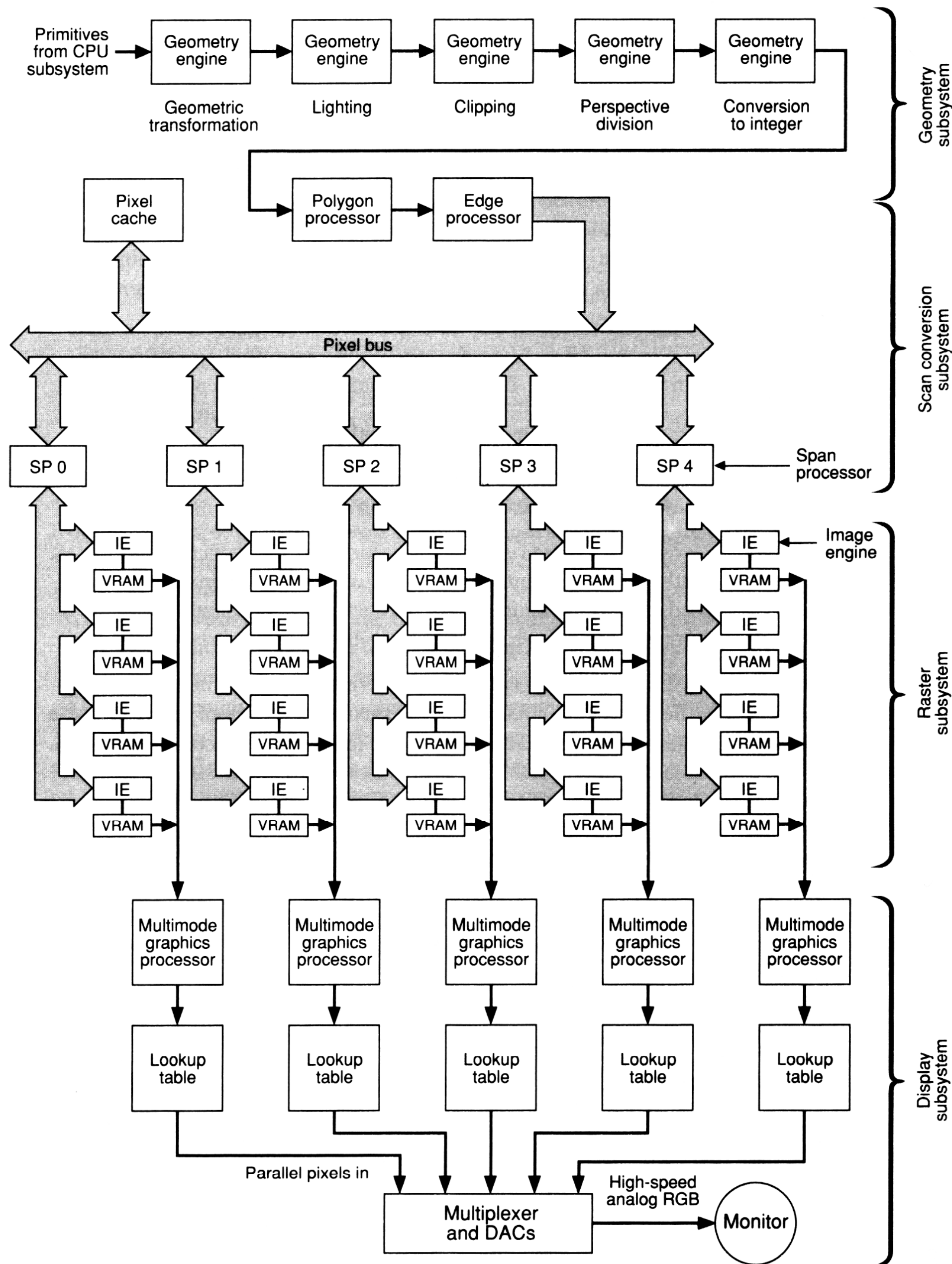

Figure 2.9: The Silicon Graphics GTX 3D graphics subsystem, which uses a 5x4 interleaved array of 20 image engines in the back-end, each with their own framebuffer. The front-end is implemented using a pipeline of identical programmable processors executing different stages of the geometry pipeline. (From [63]).

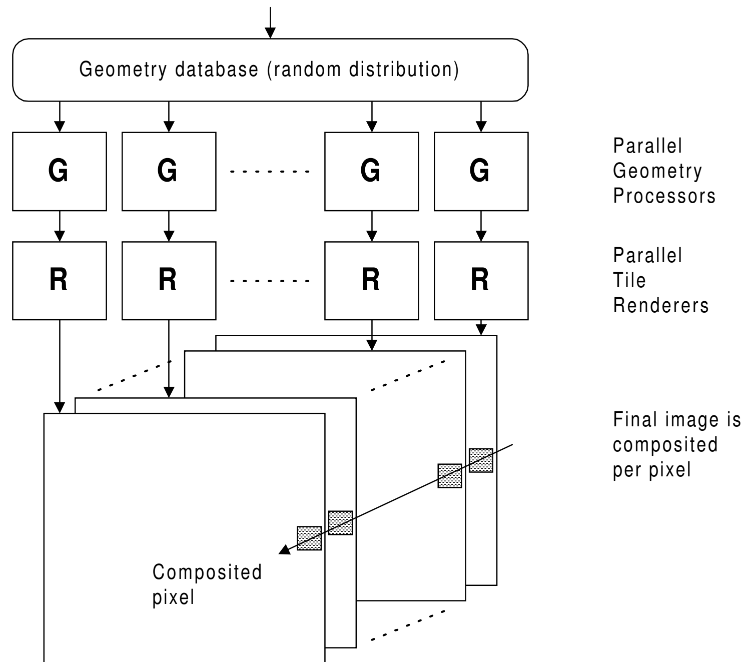

Figure 2.10: Sort-last parallelism using image composition.

transparency using the `over` operator are discussed in [49, 20, 17]. A major difficulty with image composition arises when multiple transparent triangles are layered at the same pixel, since depth ordering may be lost in the renderers. A simple way to handle this is to ignore transparency and use pixel depth to determine the pixel value. Correct handling of transparency in sort-last requires *fragment sorting* which involves sorting pixel contribution fragments from each triangle touching that pixel. Load balancing in sort-last architectures is quite vulnerable to large triangles which may place much more work on one renderer, unless large triangles are subdivided.

A good example of a sort-last architecture is the PixelFlow [155, 156, 57] architecture which uses a high-speed image composition network to combine full screen images from multiple renderers into a complete image by using pixel depth comparisons. PixelFlow achieves high performance and scalability but does not handle transparency correctly. Since PixelFlow is based on the Pixel-Planes 5 [65] processor-per-pixel architecture, it is not vulnerable to load imbalance caused by large triangles.

The Truga001 [102] architecture is a newer example of a sort-last architecture which uses a binary tree of pixel composition circuits (Truga002) to combine the

results from several renderers into one image. Other interesting sort-last architectures include the processor-per-primitive system found in [44], as well as the fragment sorting sort-last based commercial workstations E&S Freedom 3000 [55] and Kubota Denali [121]. The 3DRAM memory chip described in [46] which is used in some Sun workstations can also be considered as sort-last.

2.3.4 Hybrid sorting

While Molnar [154] only considered parallel graphics architectures belonging to *one* of the sorting classes; sort-first, sort-middle or sort-last, other possibilities do exist. The three main sorting classifications may be combined into hybrid sorting. The Hybris hybrid parallel graphics architecture is an example of an architecture combining the different sorting methods. As mentioned in [69] hybrid rendering is likely to be what is needed to raise the performance bar for rendering architectures:

> It is very likely that the rasterization systems of the future will not be based on any one single technique, but will use a creative combination of the best features of known techniques to obtain even higher levels of performance.

A recent example of such a "sort everywhere" architecture is the Pomegranate [51] architecture which was presented at Siggraph 2000. Pomegranate uses a fast redistribution network based on 4x4 crossbar switches and 10 Gbytes/s point-to-point communication channels between all stages of the parallel rendering pipeline, to build a hybrid sort-middle / fragment sorting sort-last architecture.

The VC-1 [167] is an example of a hybrid sort-middle, sort-last architecture using multiple virtual local framebuffer renderers (small tiles) which combines their results using image composition of tiles before writing them to a global framebuffer. This is essentially a tile-based renderer version of PixelFlow's image composition network.

Compared to the sorting-classes defined earlier, a hybrid sorting parallel graphics architecture does *not* necessarily need to do a complete sort at each redistribution step, as long as the combined redistribution gives the correct result. In addition, by redistributing data at multiple points, the data redistribution workload is distributed over several sorting steps. The hybrid sort-first / sort-last architecture in [195] is a good example of this, as it allows the renderers to re-balance their workload to keep all processors working.

As we have seen, good load balancing in parallel graphics architectures depends on the partitioning of the tasks. To achieve good load balancing in a tile partitioned renderer using a combination of the sort-first, sort-middle or sort-last

architectures, a high tile granularity level is preferred as well as a tiled virtual processor architecture.

Finally, the technology available today for manufacturing ASICs supports integration of small fast blocks of on-chip memory, ideal for implementing a parallel-renderer-on-a-chip where each rendering processor in a pipeline of processor farms include local memory for computations. The Hybris graphics architecture relies on such localized buffering to scale to high performance levels. This development of ASIC technology and graphics architectures was predicted in the paper [69] with this quote:

> Improving the speed of rasterization techniques surveyed here requires integrating on one chip logic and small fast memories, a capability not well supported by today's ASIC design and fabrication techniques. Nevertheless, we should expect to see more designs using "pixel caches" or virtual buffers for the same reason caches are used on all high performance computers: small memories have fast response.

In the next section we will take a closer look at what it takes to perform bucket sorting, which is needed for a virtual local buffer tile based sort-middle architecture such as the Hybris graphics architecture.

2.4 Bucket sorting

The requirements for a sorting algorithm can be relaxed slightly if we allow an approximate sort by discretizing the sort keys. This approach allows sorting of a dataset with n elements in linear time into a set of buckets. Within each bucket the elements are not sorted. This process is known as *bucket sorting*, *binning*, *chunking* or *hashing* [3, 41]. A hashing function is used to determine which bucket or set of buckets in the hash table are relevant for the given element.

In computer graphics rendering algorithms, bucket sorting is a method for optimizing the rendering system to overcome bandwidth limitations. Bucket sorting is used by several known rendering systems, such as the Pixar RenderMan and Reyes [226, 35] software rendering architectures and hardware architectures such as PowerVR [188], GigaPixel [187], Pixel-Planes 5 & PixelFlow [65, 57], as well as Apple [116, 115, 235], Talisman [223] and VC-1 [167].

Bucket sorted rendering, which is required for tile-based rendering, works by first sorting the scene into equal sized square screen-space regions (tiles) and then rendering each tile independently. A motivating factor for using tile-based rendering is that image and depth buffers can be stored on-chip using dense SRAM or embedded DRAM. Tiling also gives us the possibility to render multiple tiles

in parallel. Traditional renderers require random access to the entire framebuffer, which needs to be implemented using a large and fast memory subsystem. In contrast a tile-based renderer only needs random memory access within each tile. A small tile size is preferred because smaller and faster memories can be used. In addition a small tile size reduces the cost of a parallel tile-based renderer, where multiple renderers access their own tile-buffer. Load balancing in a parallel renderer is also improved with smaller tile size. Also, a small tile size allows more information to be stored per pixel by using wider memories. Finally a triangle is ideally only processed by the tile-renderers assigned to the tiles it overlaps. This overlap is also one of the drawbacks of the tile-based renderer, as we need to sort the triangles into buckets for all the tiles they cover. Also, the set-up cost for each triangle is magnified by the overlap factor. Because of these factors, we need to understand how to find the optimal tile-size.

A simple and fast bucket sorting algorithm is *bounding box* bucket sorting. The bounding box for each triangle is simple to calculate from the minimum and maximum values of the x and y coordinates for the three vertices of a triangle. Using the minimum and maximum values we can find the tiles overlapped by the triangle's bounding box. The triangle is then placed into each bucket it overlaps. The bounding box algorithm is cheap, easy and robust. Unfortunately it suffers from overlap overhead when a triangle is not completely inside a single tile and covers multiple tiles. Additionally the algorithm may place a triangle into buckets that are not actually overlapped by the triangle. In [36, 30] the architectural implications of bounding box bucket rendering is analyzed.

Alternatively *exact* bucket sorting may be applied. Exact bucket sorting is more complex as it requires that each triangle is scan converted into tiles, treating tiles as large pixels. Exact bucket sorting requires triangle set-up in the sorting stage, but may reduce the number of buckets that a triangle is placed into. Large triangles overlapping many tiles will be placed into fewer buckets, but small triangles may not benefit.

2.4.1 Bounding box bucket sorting overlap

The Molnar/Eyles equation (2.1) defined in [156] expresses the *overlap factor* O which denotes the average number of screen regions that each triangle overlaps. The screen is divided into equal sized regions of width W and height H, and the average bounding box size of the triangles has width w and height h.

$$O = \left(\frac{W+w}{W}\right)\left(\frac{H+h}{H}\right) \tag{2.1}$$

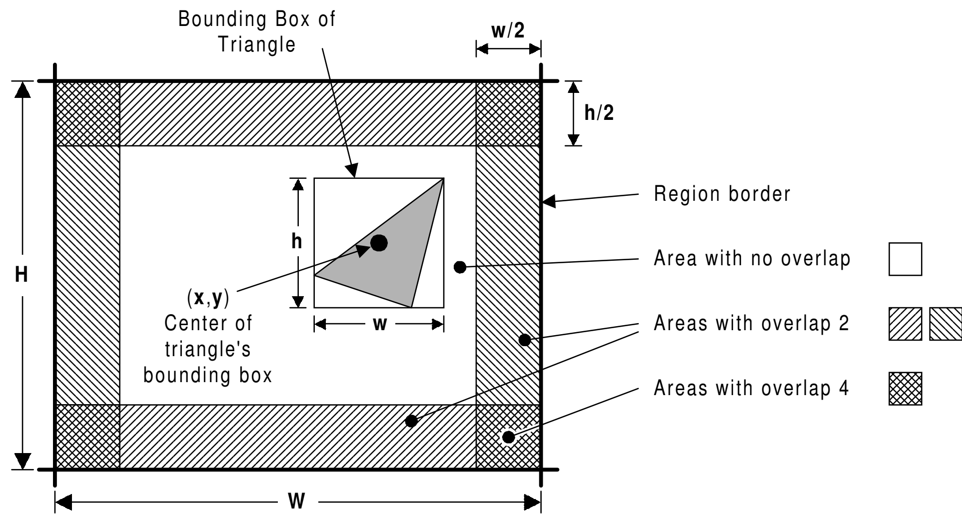

Figure 2.11: Geometric construction for bounding box overlap. The regions have width W and height H. The triangle's bounding box has the width w and height h. If the center of the bounding box (x,y) is located in the internal area there is no overlap. If it is located in the edge areas the overlap is two, and four in the corner areas. Equation (2.1) was derived from this construction.

Derivations and experimental proof of this equation are found in [156, 154]. Figure 2.11 illustrates the geometric relationship of the parameters in the equation.

Equation (2.1) allows both triangle bounding boxes and screen regions to have arbitrary sizes and aspect ratios, even when a triangle is larger than the region, as mentioned in [154]. Now we will take a closer look at the screen region aspect ratio. Assuming that the average triangle bounding box is square with constant side length s, that the screen region area A_r is constant, and with a variable screen region aspect ratio $R = W/H$, equation (2.2) can be derived from (2.1), by substituting $W = R\sqrt{A_r}$ and $H = R\sqrt{A_r}$ as shown in [54].

$$O = \left(\frac{R\sqrt{A_r}+s}{R\sqrt{A_r}}\right)\left(\frac{R^{-1}\sqrt{A_r}+s}{R^{-1}\sqrt{A_r}}\right) \tag{2.2}$$

The minimum overlap factor under these conditions is found at the local minimum of O, which is when the aspect ratio is $R = 1$. Therefore we can assume that square regions should be preferred to minimize overlap. Empirical testing in [233, 234] indicate that square regions provide the highest performance. While square regions make intuitive sense, a possible explanation is that square regions minimize the perimeter length while maximizing spatial coherence in both horizontal (span) and

vertical (scanline) directions, see figure 2.1.

Equation (2.3) is a variation of (2.1) assuming square screen regions, where S is the side length of a tile. The equation is derived in [36]. This is the overlap factor of a tile-based renderer using square tiles and bucket sorting of triangles based on their bounding box.

$$O = 1 + \frac{w}{S} + \frac{h}{S} + \frac{wh}{S^2} \tag{2.3}$$

Experimental research in [36] based on equation (2.3) suggest that in practice the overlap factor is independent of the aspect ratio of the triangle bounding box. The aspect ratio is calculated as $r = w/h$ or $r = h/w$. This suggests that we can simplify the equation by assuming $s = w = h = \sqrt{A}$ where s is the side length and A is the area of the triangle bounding box. Further, the ratio of bounding box area A per triangle area a can be defined as $\rho = A/a$ leading to the following equation (2.4).

$$O = \left(\frac{S+s}{S}\right)^2 = \left(\frac{S+\sqrt{\rho a}}{S}\right)^2 = 1 + \frac{2\sqrt{\rho a}}{S} + \frac{\rho a}{S^2} \tag{2.4}$$

According to [30] the area ratio ρ can be approximated to $\rho = 3$ when triangles are used. Note that these equations assume that each triangle is placed into each bucket its bounding box covers. Figure 2.12 shows an example of how triangles are placed in buckets and figure 2.13 shows the structure of the hash table with buckets for each tile.

Incidentally a parallel tile-based renderer is an excellent example of a sort-middle architecture. Bucket sorting is required even when not parallelized, which adds some overhead.

2.4.2 Exact bucket sorting overlap

When small triangles such as T_2 or T_3 in figure 2.12 stay within a tile or straddle the boundary between two tiles, then exact bucket sorting and bounding box bucket sorting will have the same overlap factor. But as soon as a triangle bounding box straddles both horizontal *and* vertical tile boundaries, the differences appear. T_1 in figure 2.12 overlaps 3 tiles when exactly bucket sorted and 4 tiles when sorted by the bounding box method. Figure 2.13 shows how the hash table of the buckets is represented by the algorithm. As the triangle size increases relative to the tile size, this effect is increased. Figure 2.14 shows what happens when a long and narrow triangle spans across the entire screen. The bounding box method would place it in all buckets, but exact bucket sorting only places it in the buckets actually affected by for the shaded tiles. The overhead of placing a triangle into buckets it does not actually cover causes each tile renderer affected to waste time that could be used

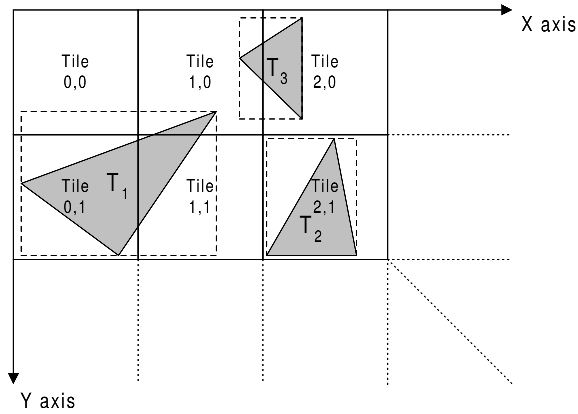

Figure 2.12: Examples of overlap in a tile-based renderer. Triangle T_2 is completely inside one tile. T_3 overlaps two tiles. T_1 overlaps 4 tiles if bounding box bucket sorting is used, but will only overlap 3 tiles if exact bucket sorting is used.

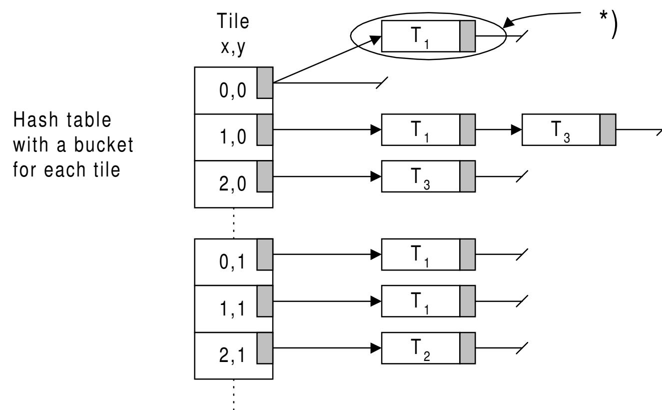

Figure 2.13: Hash table for the bucket sorted triangles in figure 2.12. Each bucket maintains a list of triangles overlapping its tile. *) This triangle entry is created when using bounding box bucket sorting, but not if exact bucket sorting is used.

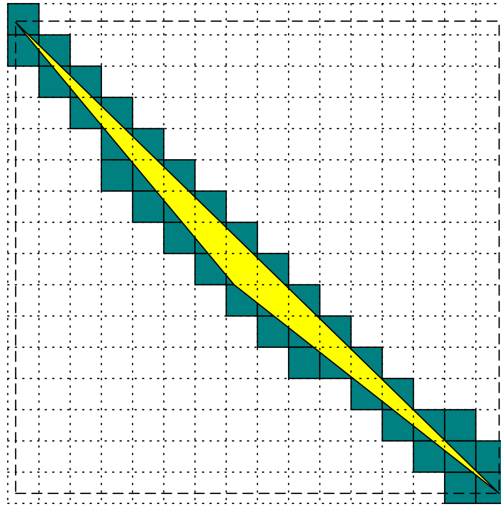

Figure 2.14: Bounding box bucket sorting will place this long and narrow triangle in *all* buckets, resulting in an overlap of $O = 256$. Exact bucket sorting will only place the triangle in the buckets for the shaded tiles, in this case resulting in a lower overlap $O = 40$.

rendering other triangles. The communication network for distributing triangles into buckets is also burdened by this overhead.

A method for exact bucket sorting is discussed in [169]. It relies on testing triangle edges against region edges using the following algorithm which determines that a triangle intersects a region if:

1. a triangle vertex is inside the region, or

2. a region corner is inside the triangle, or

3. a region edge intersects a triangle edge.

The algorithm requires these tests to be made for every region overlapped by the triangle's bounding box.

Alternatively each tile can be treated as a large pixel. By scan converting the triangle into these large pixels, the exact coverage can be determined. Unfortunately this duplicates the work done by the tile renderers to some degree, and requires triangle set-up in the sorting stage. Performing triangle set-up before bucket sorting has some side-benefits, though. The *Hybris* renderer performs triangle set-up in the bucket sorting stage as an optimization to allow reuse of rasterization parameters across all tiles covered by the triangle. Knowing the triangle set-up parameters,

exact bucket sorting can be performed using the same rasterization algorithm and rasterization rounding rules that is used in the tile renderers. Further, these rasterization rounding rules can be used to cull triangles prior to bucket sorting. While exact bucket sorting adds a scan conversion overhead it may improve the overlap factor, in particular for large triangles such as the one in figure 2.14.

Returning to equation 2.1 which in [156] was derived from this integral:

$$O = \int_0^H \int_0^W p(x,y)r(x,y)dxdy \tag{2.5}$$

where $p(x,y)$ is the probability for the triangle bounding box center to be placed at (x,y) and $r(x,y)$ is the number of regions affected by this placement. Assuming even distribution over square tiles we get:

$$O = \int_0^S \int_0^S \frac{1}{S^2} r(x,y)dxdy \tag{2.6}$$

Revisiting the geometric construction in figure 2.11 and assuming a square triangle bounding box and square tiles, then, for triangles with a bounding box size smaller than the tile size, the following can be stated: If the triangle bounding box is completely within a tile or overlaps two tiles either horizontally or vertically, exact and bounding box bucket sorting has the same overlap. Only in the case when the triangle bounding box overlaps four tiles, the triangle itself overlaps either three or four tiles. Based on the triangle bounding box area to triangle area ratio ρ we can derive an expression for the overlap factor of exact bucket sorting for small triangles. In the corner four tile overlap case, the estimated overlap with exact bucket sorting becomes

$$O_{corner} = 3 + \frac{1}{\rho} \tag{2.7}$$

allowing the following derivation, where $A_{corners}$, A_{edges} and A_{center} are the areas of the corner, edge and center regions in figure 2.11, s is the side length of the square triangle bounding box and S is the side length of the square tile.

$$\begin{aligned}
A_{corners} &= s^2 \\
A_{edges} &= (sS + sS - 2s^2) \\
A_{center} &= (S-s)(S-s) \\
O &= \frac{O_{corner}A_{corners} + 2A_{edges} + 1A_{center}}{S^2} \\
O &= 1 + \frac{2s}{S} + \frac{(O_{corner}-3)s^2}{S^2}
\end{aligned}$$

by substituting O_{corner} from (2.7) we get

$$O = 1 + \frac{2s}{S} + \frac{s^2}{\rho S^2} \tag{2.8}$$

This derivation results in the following nice expression (2.10) for the estimated overlap factor of exact bucket sorting when the triangle area a is known, by substituting $s = \sqrt{\rho a}$ in (2.8)

$$O = 1 + \frac{2\sqrt{\rho a}}{S} + \frac{a}{S^2} \tag{2.9}$$

substituting again to get rid of ρ gives

$$O = 1 + \frac{2s}{S} + \frac{a}{S^2} \tag{2.10}$$

where the area ratio ρ can be approximated to $\rho = 3$ when triangles are used, as previously stated in the discussion for (2.4). Note that $\rho = 3$ is an average estimate, a rare case such as the long and narrow triangle in figure 2.14 has a much higher bounding box area to triangle area ratio. Subtracting equation (2.9) from (2.4) yields the difference

$$\Delta O = O_{BBox} - O_{Exact} = \frac{a(\rho - 1)}{S^2} \tag{2.11}$$

which is low for relatively small triangles.

Figure 2.15 lists some estimated overlap factors calculated for bounding box and exact bucket sorting, where it is clear that the overlap factor is high for large triangles and that exact bucket sorting significantly reduces the overlap in this case. However for small triangles the difference between exact and bounding box sorting is negligible, as shown in equation 2.11.

The tile size for the renderer should be chosen to be small enough to allow good load balancing and efficient caching while at the same time be large enough to allow good coherence within the tile as well as a low bucket sorting overlap factor. A practical tile size satisfying these criteria, as well as being small enough to be practically implemented in caches and on-chip RAM, seems to be 32x32 pixels. Note that 2^nx2^n tile sizes are preferred in order to match hardware resources. Figure 2.16 plots a comparison between exact and bounding box overlap for 32x32 tiles. This suggests that an adaptive bucket sorting algorithm which dynamically chooses between using bounding box bucket sorting for small triangles and exact bucket sorting for large triangles can be useful.

	Triangle area:					
	1	4	16	64	256	1024
Tile side length:	Bounding box overlap:					
8	1.480	2.054	3.482	7.464	19.928	62.856
16	1.228	1.480	2.054	3.482	7.464	19.928
32	1.111	1.228	1.480	2.054	3.482	7.464
64	1.055	1.111	1.228	1.480	2.054	3.482
128	1.027	1.055	1.111	1.228	1.480	2.054
256	1.014	1.027	1.055	1.111	1.228	1.480

	Triangle area:					
	1	4	16	64	256	1024
Tile side length:	Exact overlap:					
8	1.449	1.929	2.982	5.464	11.928	30.856
16	1.220	1.449	1.929	2.982	5.464	11.928
32	1.109	1.220	1.449	1.929	2.982	5.464
64	1.054	1.109	1.220	1.449	1.929	2.982
128	1.027	1.054	1.109	1.220	1.449	1.929
256	1.014	1.027	1.054	1.109	1.220	1.449

Figure 2.15: Example bounding box and exact overlap for some interesting tile and triangle sizes.

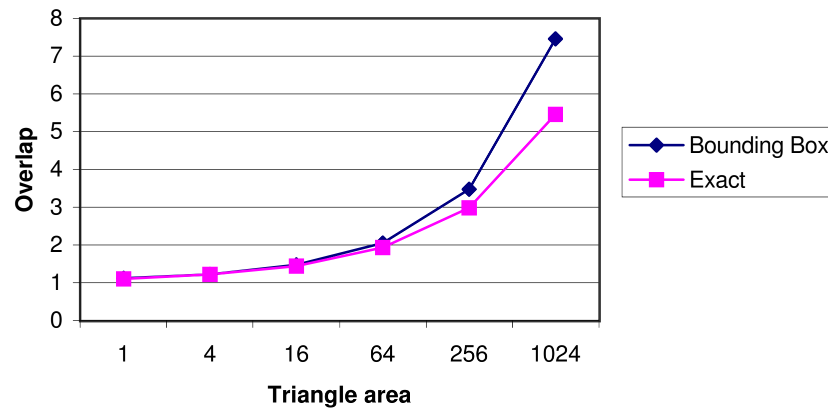

Figure 2.16: Overlap for 32x32 pixel tile.

2.5 Chapter summary

In this chapter we have discussed parallel rendering in general with a special focus on scalability. State of the art in current scalable commercial rendering architectures has been covered. An introduction to general concepts in parallel rendering has been given with an overview of some of the available options for implementing a scalable graphics architecture. This overview seems to point towards a primarily sort-middle architecture based on image-parallel subdivision of the screen into many small square tiles mapped to virtual local framebuffers. For each tile bucket sorting and buffering of work is used to load balance the jobs across virtual processors, each optimized for rendering one small square tile. In addition a partial sort-first architecture using object-parallel subdivision of the 3D model input data looks promising. The input data is split into many small sub-objects to distribute work over several geometry processors while maintaining data coherence. Finally sort-last is used to assemble the final image from tiles. Image composition of overlapping tiles might be useful in order to allow the architecture to scale even further, if correct handling of transparency is not an issue.

Chapter 3

Designing a Scalable Graphics Architecture

This chapter presents an analysis of how the *Hybris* graphics system architecture is designed and implemented at a high abstraction level slightly above the possible implementations. The intention is to specify a portable and scalable architecture which may be implemented for many different software or hardware based computer technologies. Hybris is designed to reduce the computational load at many levels and to be scalable. The graphics architecture was originally named HPGA which is short for **H**ybrid **P**arallel **G**raphics **A**rchitecture, and later renamed to Hybris (Danish for Hubris). This graphics architecture is a hybrid because it applies a combination of several types of parallelism in order to scale. Chapter 18 of [63] defines a hybrid-parallel graphics system as one which uses a combination of object-order and image-order rasterization techniques.

3.1 Understanding the problem

In order to define and implement a scalable graphics architecture we need to understand the operation of all parts of the graphics pipeline. This understanding will help the design for a scalable graphics architecture.

Traditionally the *graphics pipeline* is a serial processing pipeline for processing one graphics primitive at a time. While a straightforward implementation of this model lends itself towards easy implementation in both software and hardware, it is not necessarily the most optimal.

Designing a *scalable* architecture for graphics forces us to take a new look at the graphics pipeline. Distributed processing is needed for good scalability, leading to an architecture composed of multiple localized data processing units. The pro-

cessing units are not identical, several different unit types are needed to form the graphics pipeline. From the overview of scalability in graphics architectures given in the earlier chapter, giving an idea of the generic parallel graphics architecture, we must find a practical implementation.

For developing the *Hybris* graphics architecture, the architecture has been evolved mainly in a software environment, reflecting useful software implementation methods. The architecture is targeted towards an implementation with efficient utilization of CPU and system resources such as instruction scheduling, caches and memory bandwidth. Additionally the 3D graphics rendering algorithms used in Hybris are optimized towards achieving these goals. Using a general purpose computer for development has allowed us to apply an abstraction of the design process above a straightforward implementation of the archetypical graphics pipeline. The main difference between the software implementation and the standard graphics pipeline is how *memory* is used.

The usage of memory in the graphics architecture allows buffering of temporary data and variables. While buffering enables re-use of earlier calculated data values, an equally important aspect is that the memory can be partitioned to match the data coherence present in computer graphics. A useful way to improve memory usage in any system is to apply loop fusion and strip mining techniques [117, 143, 228], which are examined in relation to Hybris in the paper [88], see also the next chapter.

To enable a scalable architecture, a workload distribution scheme must be applied as well as a practical way to collect and combine partial results into the final result, in this case one rendered frame of interactive real-time animation.

The software version of the Hybris rendering architecture was originally developed by breaking the rendering pipeline apart into several independent functions or loops, each reading its input data from memory and writing output data back to memory. This isolation of components made it possible to test each component separately and test various implementations of each component. However, when the components are configured into a graphics pipeline this approach is not necessarily optimal when compared to the direct pipeline approach where no temporary data exist in memory like that.

The advantage of exposing temporary data in memory is greater freedom to experiment with various memory access schemes for data reuse and data access coherence as well as enabling a means for data redistribution for use in parallel implementations. Data blocking or chunking schemes has proved to be a very efficient method for optimizing data access performance in computer systems equipped with caches. Previously, data blocking schemes have been widely employed for supercomputer applications e.g. in implementations of the linpack and scalapack mathematical subroutine libraries. Today these techniques are not restricted to su-

percomputers, but can be employed by modern personal computers as well as new ASIC technologies with enough space for on-chip memories.

The Hybris graphics architecture is an attempt at applying data blocking techniques to computer graphics. This was done by experimenting with various code transformations by manually applying techniques such as loop fusion and strip mining to achieve good cache and memory utilization.

3.2 Development of the Hybris rendering architecture

As indicated earlier it is desirable to localize the data processing in the rendering architecture. An early attempt at this using an approach based on a scanline z-buffer algorithm [84, 89, 130] worked nicely in a single processor environment, but because of data dependencies between scanlines it caused many problems with the design and implementation of an efficient scalable graphics architecture. Some similar scanline based architectures are discussed in [116, 31, 68].

The basic rendering technique used in Hybris is a shaded triangle renderer using a combination of the Gouraud and Phong smooth shading techniques originally proposed in [73, 182]. This thesis will not discuss all topics relevant for implementation of a 3D graphics system, as some of the basic topics as well as a description of the scanline renderer can be found in [84, 89, 130] as well as textbooks on computer graphics such as [63, 230, 229, 152]. In the following the focus is on the properties of the tile-based rendering pipeline in Hybris, with emphasis on topics relevant to parallelism, scalability and efficient implementations.

The tile-based version of the Hybris renderer was developed after it was realized that it is not always a good idea to maintain an active triangle list. An active triangle list allows incremental rendering one scanline at a time without duplicating triangle nodes for each scanline. While an active triangle list is good in the sense that it minimizes data-redundancy, it has to be updated when crossing over a scanline boundary. Bidirectional dataflow is needed between the the active triangle list and the scanline renderer. To reduce the impact of this problem a multi-scanline shaped on-chip buffer was investigated instead of the single scanline on-chip buffer that was used by the scanline renderer. A multi-scanline on-chip buffer would allow the active triangle list to be updated less often, but it would also require much more chip-area for the on-chip buffer to make this worthwhile. To reduce the on-chip memory area and retain the advantages of multi-scanline rendering a tile-shaped buffer seems ideal. However the active triangle list needed for incremental bucket sorting would have to be updated when crossing tile boundaries both in horizontal and vertical directions. Additionally triangles would have to be clipped on the vertical boundaries between tiles, which at first seems to be a major problem.

Tiled rendering using complete bucket sorting seems to be wasteful in the sense that it requires copying of each triangle to *all* the buckets it covers. This is in contrast to incremental bucket sorting which places a triangle only in the first bucket it covers, using an auxiliary active triangle list to keep track of triangles while rendering. The advantage of complete bucket sorting is that it removes the need to maintain an active triangle list. This means that the tile renderer can now simply read the triangles to be rendered in a tile sequentially from the bucket in the triangle heap. More importantly, it never needs to write back to the triangle heap nor keep track of active triangles. In a hardware implementation this simplifies the interface to the bucket sorted triangle heap. From a memory point of view it also removes the need to change between reading and writing to the triangle heap. Unfortunately, the tile renderer now has to clip every triangle to the tile boundary e.g. by using direct evaluation, while the scanline renderer could continue directly to the next scanline by incremental evaluation.

The techniques used in the Hybris tile renderer can be compared with other approaches. Kelley et al. [116] describe a scanline rendering hardware architecture, which uses direct evaluation interpolation to avoid incremental write-back to the active triangle list. Each node in their active triangle list specify only start and end interpolation values, so each scanline renderer must use direct evaluation to interpolate spans and pixel values, using a computationally expensive division for every scanline a triangle covers. A side benefit of this is reduced memory requirements in their active triangle list because forward differencing slopes are not stored. In comparison the Hybris tile renderer relies on a combination of forward differencing and direct evaluation to gain benefits of both approaches.

In the following sections we consider various rendering concepts relevant to the tile renderer. In order to improve the speed of the basic graphics pipeline, several optimization techniques may be applied to reduce computational load. Many of these techniques are not required for scalability, but are implemented in the architecture because of the achievable speedup. Other optimization techniques are made possible by the way data is partitioned in order to allow scaling.

3.2.1 Clipping

To ensure that triangle coordinates sent to the tile renderer are in the correct numerical range, geometric clipping must be used. A very popular polygon clipping algorithm is the Sutherland-Hodgman reentrant polygon clipping algorithm [214], which in our case can be used to clip triangles against the six planes of the view-volume corresponding to the screen, see figure 3.2. The algorithm breaks the clipping problem down to clipping the triangles against one clipping region edge at a time. Figure 3.1 illustrates how the algorithm clips a polygon against a rectangular

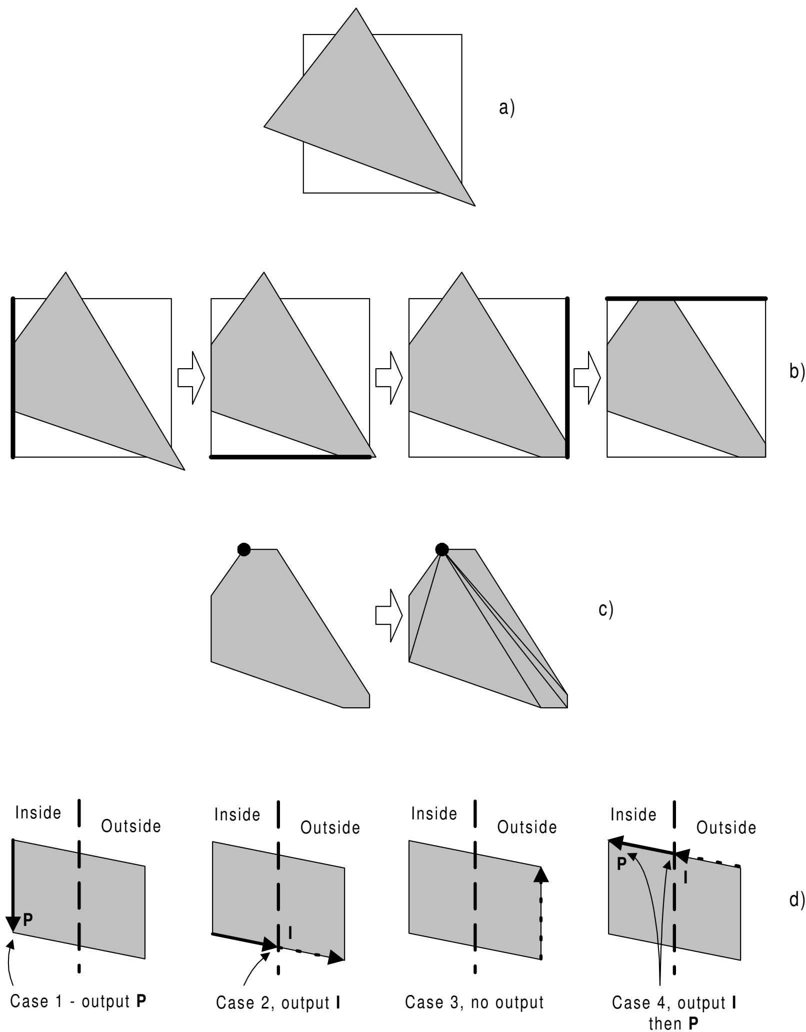

Figure 3.1: Sutherland-Hodgman reentrant polygon clipping [214]. a) The polygon and a rectangular clipping region. b) The polygon is clipped in turn against each region edge. c) The resulting polygon (7 edges in this case) must be triangulated in order to be processed by a triangle renderer. d) For each polygon edge, these four cases are considered by the Sutherland-Hodgman algorithm for each region edge.

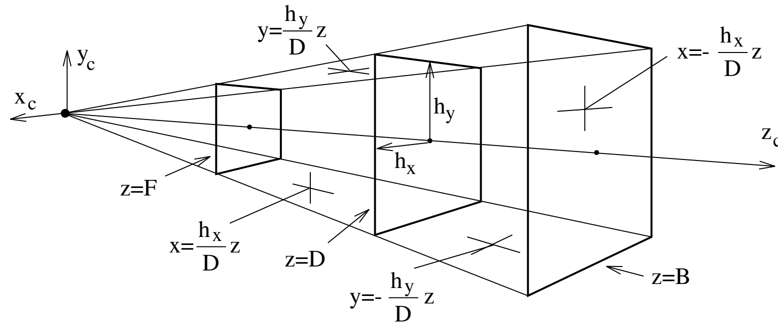

Figure 3.2: View-volume, which defines the 3D region visible on the screen. It is defined by the perspective projection parameters h_x, h_y, D, F, B, forming a truncated pyramid.

clipping region. Geometric triangle clipping is done before triangle setup, and is part of the object-based front-end rendering pipeline.

While the Sutherland-Hodgman clipping algorithm works nicely for clipping against the screen boundary, it would cause a huge overhead to clip triangles to every tile. The next section discusses a solution to this problem.

3.2.2 Fast tile boundary clipping

The tile renderer must clip the triangles in each bucket against their tile boundary. We can rely on the fact that all triangles have been clipped to the global framebuffer. Since we now know the bounded numeric range of the x and y coordinates, a technique known as *guard-band clipping* can be applied. Guard-band clipping is facilitated by an extension of the interpolation calculations needed to render a triangle. This allows a simple form of clipping known as scissoring. Instead of relying on completely clipped coordinates to facilitate simple DDA interpolation, coordinates are now allowed to extend outside the current tile. Figure 3.3 shows the different cases which may appear in guard-band clipping.

In practice guard-band clipping is implemented by introducing special case checking before the forward differencing DDA interpolation loops to adjust the interpolation starting points. This requires extra multiplications and conditional testing, but since the extra calculations can be overlapped with the following forward differencing interpolation by pipelining, we essentially get "free" clipping of triangles inside the outside guard-band boundary.

Earlier, the guard-band clipping technique has been described in [5] as scissoring, where it is used to optimize full-frame viewport clipping in the SGI Real-

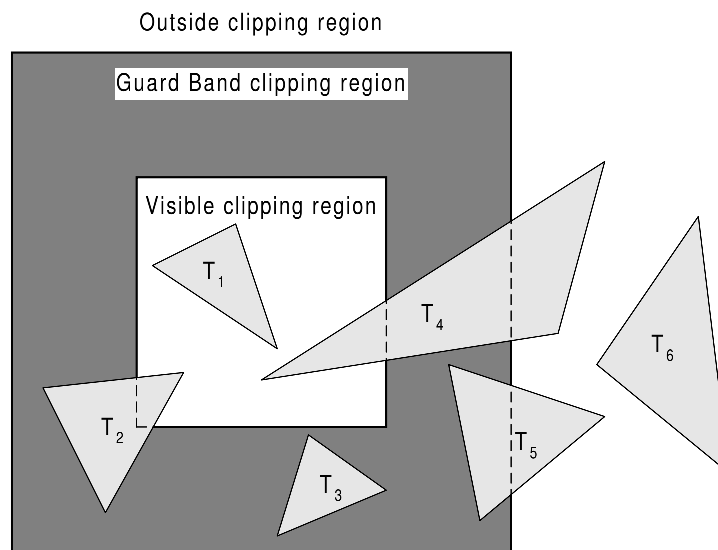

Figure 3.3: Guard-band clipping. While only the visible region is rendered, the numerical range of coordinates inside the guard-band region must be representable in the rasterizer. In this example, T_1 is completely inside the visible region and is not clipped. T_2 straddles the boundary between guard-band and visible regions, and can be clipped (scissored) during rasterization. T_4 straddles all region boundaries and must be clipped at least against the outside boundary. T_3, T_5 and T_6 are completely outside the visible region and can be safely culled. Alternatively (but less efficient) T_3 could also safely be sent to the rasterizer, while T_5 must be clipped against the outside guard-band boundary first and T_6 must be culled.

ityEngine, using a combination of extending the numerical range of pixel coordinates and scissoring to adjust triangles to the viewport. By allowing triangles to extend well beyond the viewport boundary, trivial culling of triangles outside the viewport can be combined with clipping against the outside boundary. Remaining triangles inside the outside guard-band boundary may then be trivially accepted.

While the Hybris tile renderer automatically uses scissoring for clipping against each tile boundary it can also take advantage of guard-band clipping against the entire visible viewport. For performance reasons negative valued screen coordinates are not allowed in the tile renderer back-end, but by extending the numerical range to allow a guard-band of “ghost” tiles around the viewport, full viewport guard-band clipping can be implemented by adding an offset to the screen coordinates (both triangle and tile position). This can be applied to the FPGA implementation [207] with no changes to the hardware, since it only renders the active

tiles.

Note that guard-band clipping cannot handle all possible triangle locations, so in the cases where triangles extend beyond the outside guard-band boundary regular geometric clipping must be used, as described in the previous section.

3.2.3 Fast floating point to fixed-point conversion

ANSI C specifies different rounding rules for floating point to integer conversion than all other regular floating point operations. Floating point to integer conversions *truncates* the value while regular floating point operations *rounds* to the nearest value. Because of this, C compilers generate special code for the conversion which changes the rounding mode of the CPU. Unfortunately this code causes the Pentium III and other CPUs to *flush* the execution pipelines to ensure consistent rounding, at a high performance penalty. This overhead has been found to seriously affect the performance of the tile renderer, if counteractions are not taken. Additionally proper scaling is needed to get the n fractional bits of a fixed-point number, which normally requires floating point multiplication by a constant. Floating point to fixed-point conversion is an often overlooked and serious performance bottleneck, e.g. the Silicon Graphics GTX [4] dedicated an entire geometry processor just to perform floating point to fixed-point conversion (see the figure on page 27), while the InfiniteReality [158] includes dedicated ASIC support for this.

To overcome these problems, Hybris exploits a property of IEEE 754 floating point numbers [7] to allow both conversion to integers *and* proper fixed-point scaling, using one floating point addition with a special value designed to extract the needed numerical range from the floating point value *and* generate the bit-pattern needed for a fixed-point value. A 32 bit IEEE 754 floating point number uses a bit-pattern where bit 31 is the sign, bits 30–23 is the exponent and bits 22–0 is the mantissa. Thus, only 23 significant bits are available. Adding 2^{23-n} to an IEEE floating point number has the effect of shifting the bit pattern of the mantissa to the right (the shift count depends on the value of the exponent) to create the bit-pattern needed for a fixed-point number with 2^n bits in the fractional part, provided that the floating point value is less than 2^{23-n}. Now bits 22-0 of the floating point value contains the bit-pattern for the fixed-point value. By masking away bits 31–23 we get the fixed-point value, including the required scaling multiplication by 2^n for free in the process. In C this conversion can be expressed as:

```
#define SCALING 12  /* 12 fractional bits */
union {
  float float_temp; /* 32-bit floating point value */
  int float_bits;   /* corresponding 32 bit-pattern */
```

```
} u;
u.float_temp = float_value + ( 1 << (23-SCALING) );
fixed_value  = u.float_bits & 0x007FFFFF;
```

A drawback is the need for a temporary value, as C cannot assign a `float` to an `int` directly without enforcing its own conversion. Another drawback is that this conversion only works for positive numbers in a limited numerical range ($0 \leq v < 2^{23-n}$), although to handle signed values an offset (2^{23-n-1}) can be added to values prior to conversion and subtracted afterwards. Alternatively we can check for the sign of the floating point number first and negate or instead subtract 2^{23-n} from the floating point number, followed by a negation of the fixed-point value. Finally, the maximum precision is strictly 23 bits. All this is can be good enough, since clipping against the outer (guard-band) boundary has been performed prior to conversion, guaranteeing the numerical range of the values. The number of bits required in the fractional part of a fixed point number depends on the desired sub-pixel accuracy. Fixed point numbers used in forward differencing interpolation loops require additional fractional guard bits based on the maximum number of iterations needed, i.e. $\log_2(\ \textit{max number of iterations}\ )$. E.g. interpolating over a range of 4096 values requires 12 guard bits.

Since floating point calculations are only needed in the transformation stage of the graphics pipeline, and since temporary transformed vertex data are stored in memory (or cache), it is possible to split the conversion into two stages masking away the temporary variable overhead. This is done by adding the bit-pattern adjustment value to the perspective transformation parameters and storing the adjusted values in the temporary screen-space transformed vertex array. When the triangle setup process reads vertex data, it simply has to mask away the exponent bits to get the final fixed-point value.

Other subtleties related to clipping, floating point and fixed-point rounding errors for screen and pixel coordinates are covered in [19, 18].

3.2.4 Back-face culling

A well known technique for reducing the computational load in polygonal computer graphics is back-face culling [63, 84]. Back-face culling is a technique to remove triangles which would eventually be covered by front-facing triangles when rendering an opaque closed-surface 3D object. A triangle is considered to be back-facing when it faces away from the viewer. To express this, a vector normal to the triangle's plane is used. The sign of the dot product of this normal vector and the vector from the view point to one of the triangle's vertices then determines whether the triangle faces towards or away from the view point. Triangle plane normal vectors suitable for back-face culling can be calculated from the cross product of two

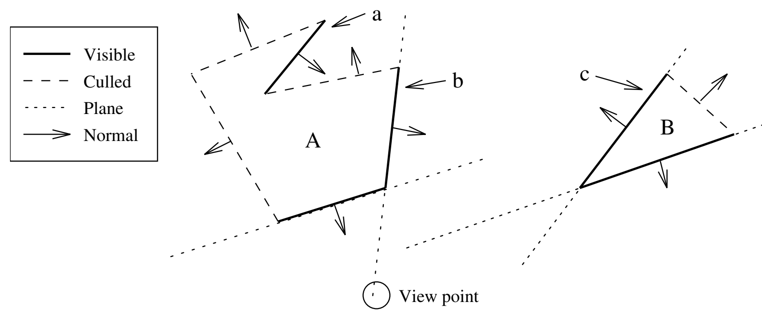

Figure 3.4: Back-face culling. The location of the viewpoint relative to the triangle's plane determines which way the triangle faces.

vectors defined by two edges in the triangle. Vertex ordering for all triangles must be consistently either clockwise or counterclockwise for this to work. Figure 3.4 shows how the triangle's plane and the viewpoint are used to perform back-face culling.

While back-face culling is conceptually simple, it is not obvious where in the graphics pipeline back-face culling should be done. One method is to calculate normal vectors on the fly from triangle edges during setup. Another might be to store pre-calculated normal vectors in the 3D object description, and transform them to view space along with vertices, or alternatively transform the view space viewing position vector back to object space using an inverse transformation, and perform the dot product test directly on untransformed triangle normals. Furthermore, it is possible to use a space-saving encoding of the triangle normal, for compact representation in object space.

In Hybris the triangle normals are stored as a quantized representation along with triangle mesh indexing values, requiring only 32 bits for storage of the quantized triangle normal. The normal vector is quantized to 8 bits for each of the 3 coordinates plus an extra 8 bit parameter to enable the triangle plane equation to be represented. This has the advantage that a triangle can be back-face culled before even looking at one of its vertices. As the viewpoint can be transformed to object space and be evaluated in the plane equation, then depending on the sign of the result we know whether the triangle is back-facing. A margin for error is allowed to compensate for quantization noise, causing some borderline cases to be classified as front-facing even though they really are back-facing. This quantization method has proven to work well in Hybris.

Other normal quantization methods exist, e.g. [43] which uses quantized spherical coordinates to represent normals using an 18-bit encoding, decoded through a

2000 entry look-up table. An interesting alternative is to encode a triangle normal as a 16-bit index to a bit mask table [240], which must be updated every frame to match the viewing direction. Using a table look-up and a logical AND operation for each triangle it determines whether a triangle is back-facing, although the method has a large margin for error since it depends on the view direction rather than the view point, culling only up to 40% of the triangles.

In comparison our quantization method requires slightly more bits (24 for the normal or 32 for a plane equation), is much simpler to implement, and does not require any table look-up although three 8 bit multiplications are needed.

An alternative method for back-face culling is to calculate the triangle normal after perspective projection to 2D screen coordinates, allowing us to simplify the calculations as the sign of the z component of this triangle normal tells us which way the triangle faces. While simple and cheap, this test must be done quite late in the geometry processing pipeline, at least after perspective projection, clipping and transformation, limiting the efficiency of back-face culling. Still, as this method does not require storage of triangle normals it is more suitable for dynamically changing triangle data.

In the quantized normal back-face culling, borderline back-facing triangles wrongly classified as front-facing are later removed in the triangle setup stage where the triangle normal vector is calculated from screen-space transformed vertices. This normal vector is also needed later for plane equation based calculation of triangle colour shading and z-depth gradients for rendering. Back-face culling using the quantized normal vector method improves speed by culling away triangles before fetching vertices. In the current software implementation of Hybris a small performance increase is observed (about 7% for the Bunny).

Quantization for compression of the source data may also be applied to the vertices, as described in [43, 221]. However this type of compression is only an advantage if the vertex geometry processor is implemented in hardware, or if the model is to be compressed for internet transmission. A software geometry processor must use the CPU's floating point units for speed, and would be slowed down by decompression.

3.2.5 Hierarchical back-face culling

The quantized back-face culling method was developed further to enable hierarchical back-face culling which allows trivial rejection of entire groups of triangles. A prerequisite for this is the partitioned object database described later in this chapter. Since vertex data is not needed for back-face culling we can completely avoid transforming vertex data until at least one triangle in the current dataset is classified as front-facing. If all triangles are back-facing, no vertices are transformed and the

Figure 3.5: Partitioned model of the Stanford Bunny. Each colour represents one partition of 500 triangles. Hierarchical back-face culling makes it possible to reject an entire partition of triangles before screen-space transformation.

object is trivially rejected. Note that this will only work for objects with a certain degree of coherency in the position and direction of triangles. For typical closed manifold objects this is not typically the case. However when using a partitioned object database many smaller groups of triangles are formed, where all triangles in an object partition may be facing in approximately the same direction. While this method requires testing all the quantized normals of an object partition to possibly cull it, a substantial speedup is possible. Testing on the Stanford Bunny shows an additional speedup of approximately 8% when hierarchical back-face culling is used on a partitioned bunny. Figure 3.5 illustrates the topology of the partitions. In figure 3.4 we may think of the dashed lines as such object partitions.

An alternative approach to implement hierarchical back-face culling is a method which uses visibility cones, i.e. a range of normal vector directions forming a cone that can be represented in a compact way (height and diameter at base), representing the range of normal vectors of the partition.

Similarly to hierarchical back-face culling, we can also perform hierarchical object visibility culling to improve clipping performance. If some partitions of an object are completely outside the view volume we can trivially cull them. Similarly if a partition is completely inside the view-volume we can trivially accept it, allowing us to bypass view-volume clipping completely when processing the object partition. Only those object partitions straddling the boundary are candidates for clipping. In order to quickly determine if an object partition is inside, outside or straddling, the object partition is embedded in a *bounding sphere* which allows a simple test to determine visibility.

3.2.6 Pixel addressing rounding rules

To rasterize or scan-convert a triangle to a rectangular grid of pixels, we need to determine exactly which pixels are affected by it. This is important in order to avoid empty holes between connecting triangles and to avoid overlap of connecting triangles.

Converting floating point or fixed-point pixel coordinates to integer screen pixel coordinates requires rounding of the coordinates. The pixel rasterization rounding rules described and analyzed in [149, 230] were used in previous implementations of the Hybris architecture. For horizontal spans given real valued starting and end points x_{start} and x_{end} these rounding rules are:

- round x_{start} up
- round x_{end} down
- if the fractional part of x_{end} is 0 then subtract 1 from it

the rounded versions of x_{start} and x_{end} now defines the range of pixels to be drawn, including both the start and end values. Using the rounded versions, if $x_{start} > x_{end}$ then we can safely cull the span. Otherwise $n = x_{start} - x_{end} + 1$ is the number of pixels to draw.

In C these rounding rules can be expressed for fixed-point numbers represented as integers scaled by `SCALE` bits like this:

```
/* xstart, xend rounding */
rxstart = xstart>>SCALE;
rxend   = xend>>SCALE;
rxstart+= (((unsigned int)
            (-(xstart&((1<<SCALE)-1))))>>31);
rxend  -= (((unsigned int)
            ((xend&((1<<SCALE)-1))-1))>>31);
```

In comparison, DirectX [148] and OpenGL [165, 204] rounds both start and end values down and does not draw the pixel at the end value. Pixel centers in OpenGL are defined to be at half-integer coordinates, but at integer coordinates in DirectX (equivalent to adding 0.5 to coordinates before rounding). The slightly more complex pixel rasterization rounding rules from Watt [230] were used in Hybris, where they resulted in an image shifted relative to OpenGL rounding, in effect skipping the pixel at the start value instead. Thus it is an advantage to adopt the OpenGL rounding rules in order to get consistent results. Recent versions of Hybris use these simpler rounding rules, which were found to work just as well

as the more complex rounding rules previously used. As a side notice Watt's new book [229] now also follows the simpler rules, noting that the choice is arbitrary. In C the simpler rules can be expressed as:

```
/* xstart, xend rounding */
rxstart = xstart>>SCALE;
rxend   = xend>>SCALE;
```

The rounded versions of x_{start} and x_{end} now defines the range of pixels to be drawn from the start but *excluding* the end value. Using the rounded versions, if x_{start}=x_{end} then we can safely cull the span. Otherwise $n = x_{start} - x_{end}$ is the number of pixels to draw. Figure 3.6 shows an example of applying these pixel rounding rules.

3.2.7 Sub-pixel triangle culling

The pixel rounding rules described above gives us an additional possibility to allow culling (or zero-sizing) of very small triangles, which is a big advantage when rendering complex models with many small triangles. In effect, the rounding rules will remove triangle edges and spans which have both starting and ending points within the same pixel, as neighboring edges and spans will cover these pixels. If we apply the rounding rules to a triangle's bounding box *prior* to rasterization we are now able to cull very small triangles before sending them to rasterization, where it would be culled on a span-by-span basis anyway. A sub-pixel triangle culling test checks the rounded horizontal and vertical coverage of the bounding box, and if one or both of these are zero the triangle is culled. Figure 3.6 shows how a small triangle is culled using sub-pixel culling, note how the surrounding triangles merge to form a less complex surface.

An experiment to extend this mechanism to cull long and thin diagonal triangles with a larger bounding box, based on geometric area calculations, proved to be unreliable as holes would sometimes appear in the surface at the points where spans cross pixel boundaries.

However, bounding box sub-pixel triangle culling has proved to boost the efficiently of Hybris greatly, since the culling can be performed during triangle setup *before* sending the triangle node packet to the bucket sorted triangle heap, saving valuable bandwidth. Without this technique designs relying on the PCI bus, such as the FPGA implementation, would suffer from even more severe bandwidth limitations. Sub-pixel culling is an important technique in Hybris as is improves the scalability of a parallel implementation by reducing the communication bandwidth requirements.

If sub-pixel rasterization is desired in order to improve rendering quality by super-sampling anti-aliasing, we can interpret this as rendering to a higher reso-

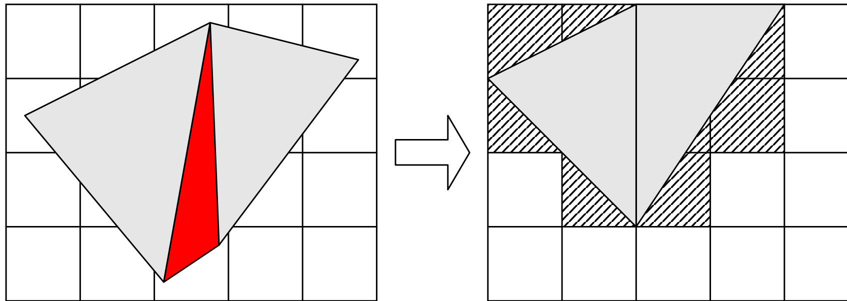

Figure 3.6: How the pixel addressing rounding rules and sub-pixel culling affect triangles. On the left side are three triangles *before* rounding. On the right side the triangle coordinates have been rounded to pixel coordinates. Note that the small center triangle has been culled due to sub-pixel culling. The remaining triangles are rendered as the dashed pixels, with no gaps or overlap between triangles.

lution pixel grid. Sub-pixel culling will be less efficient as the triangle bounding box must fit inside a sub-pixel in order to be culled. Further optimizations for the anti-aliasing case are possible though, as e.g. a sub-pixel triangle completely within a screen pixel but larger than a sub-pixel will only contribute to one screen pixel, which leads to an algorithm which reduces the triangle information to a point primitive based on the triangle's bounding box. This remains as a topic for future study.

As an interesting observation about using rounding for sub-pixel triangle culling is that it can be viewed as a very simple and quite efficient *dynamic* and *view-dependent* mesh simplification method. This should be compared to other more complex methods such as mesh optimization [93] and progressive view dependent meshes [90].

3.2.8 Sub-pixel shading correction

Because of the pixel addressing rounding rules unfortunate shading artifacts may occur. This may happen when rounding snaps a triangle edge from one pixel to the next. In our case where triangles are rendered by drawing a sequence of horizontal spans, some discontinuity in the shading values between two scanlines may be observed, most notably when long thin triangles are oriented nearly vertically on the screen. In Watt [230] this effect is described as a problem occurring only for texture mapping. However the effect on interpolative shading is also painfully visible in certain cases, as seen in figure 3.7. Depth interpolation is not affected

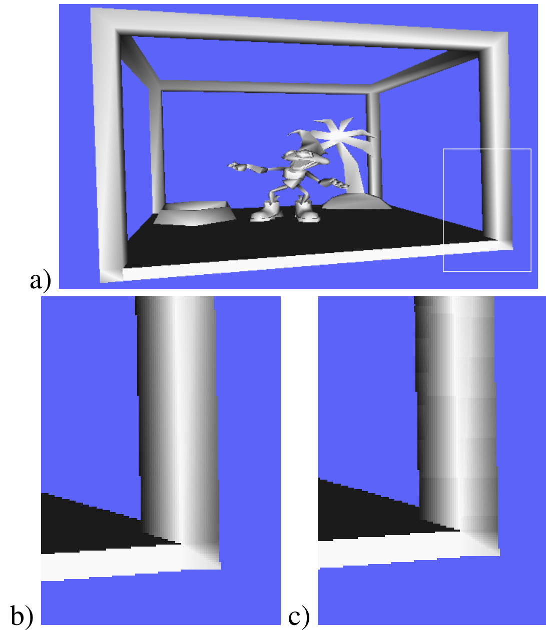

Figure 3.7: Sub-pixel shading correction. a) A frame from a VRML animation. b) Zoom in on boxed region showing the result when sub-pixel shading correction is applied. c) Without correction.

as adversely, as the effect is only visible as a slightly ragged edge in scenes with surface intersections.

To calculate the sub-pixel shading correction we need both the original and rounded pixel coordinates to correct the start value used for incremental interpolation. Since we round down, this has the effect of shifting the resulting image up and left. This corresponds with moving the pixel sample point from the center to the bottom right corner. Thus sub-pixel correction at the scanline level must move the shading start value to the *right*. The starting value can be expressed as:

$$p_{start} = p_{left} + k\frac{dp}{dx} \tag{3.1}$$

for left-to-right incremental interpolation. The parameter k is expressed as:

$$k = 1 - (x_{left} - rx_{left}) \tag{3.2}$$

to reflect sampling at the *right* side of the pixel. In C this can be expressed using fixed-point scaled integers as:

```
/* Sub-pixel shading correction at left edge */
shade+=dsdx-(((xleft-(rxleft<<SCALE))*dsdx)>>SCALE);
```

Unfortunately sub-pixel shading correction may cause overflow in incremental interpolated shading when rendering very small triangles. Overflow wrap-around causes artifacts visible as black or white pixels at triangle edges. One viable solution to this problem is to use saturation arithmetic for interpolation to avoid overflow wrap-around problems, which can be implemented in hardware. For software, saturation arithmetic is available in the Intel MMX instruction set extension.

3.2.9 An alternative: Point rendering

Recently rendering algorithms based on point primitives rather than triangles have gained some interest as evidenced by recent papers at Siggraph 2000, Surfels [181] and QSplat [193]. An overview of point sample rendering can be found in [77].

Rendering using point primitives involve representing surfaces as surface elements (e.g. Surfels) and then *splatting* them onto the screen using perspective projection. This can be quite efficient, e.g. QSplat is being used to quickly render the very complex models of the Digital Michelangelo 3D scanning project [127]. An advantage of point rendering is that it only requires a database of unconnected points to represent the 3D model. In comparison a triangle mesh 3D model must also specify how vertices are connected to form triangles. The Stanford Bunny and Buddha [208] models we use for testing were originally laser-scanned as a collection of points, which were later connected to form a triangle mesh using an algorithm described in [40].

A point or "Surfel" surface element is a geometric rendering primitive which contains the following information:

- Position in 3D object space.
- Orientation, e.g. by a normal vector to define a surface.
- Surface properties (color and texture parameters).

As an experiment a point renderer was implemented and integrated into Hybris. Since we use a differential expression in the triangle node packets, it is possible to also express axis-aligned boxes and trapezoids using the same protocol. This allows us to use the *unmodified* triangle rendering back-end to render points by in effect representing a point as an infinitely tall triangle bounded in the y-range, by assigning zero valued slopes for the span edge interpolations. This approach of using a variable sized point primitive for splatting is similar to the approach in QSplat. The implementation in Hybris also allows dynamic selection between

whether a sub-mesh should be rendered as a triangle mesh or as a point set, based on the relative distance from the viewer.

Since point rendering should be more efficient for dense datasets than triangle rendering it was expected that this would speed up rendering. Surprisingly, rendering speed with points were in many cases found to be *slower* than with triangle rendering, especially if the point size is large. Unfortunately the points must be quite large to avoid holes between them. Large point sizes lead to considerable amounts of pixel overdraw, slowing down the rendering back-end. Also, the bucket sorting overlap factor for large points is higher than for triangles. Small points caused other performance problems, as an object with a small projected area sends all its point to the back-end for rendering, mainly because no reliable rounding rules for dynamic sub-pixel rounding and point culling were found. A reason for this is that no connectivity information exist in the point database as it does for a triangle mesh. In comparison, triangle rendering can exploit dynamic sub-pixel triangle culling to speed up rendering. The QSplat renderer solved this problem by pre-calculating a hierarchy of point data sets, switching to a lower detail level when the area is small or when faster rendering is needed. Hybris would be able to use a similar technique through its implementation of the VRML LOD (Level Of Detail) node.

Even though some performance problems with point rendering were discovered, further optimization of the technique is possible in Hybris. However the fundamental problems with overdraw, bucket overlap, a high required sample rate and a low resulting image quality (because large pixels protrude outside the object boundary) causes some scepticism against point rendering. Some of the nice advantages of point rendering which were explored with Surfels are integration of texture colors into the point primitives to eliminate texture map lookup. This technique is also applicable to triangle rendering, simply by adding colors at the triangle vertices which correspond to texture map values. Now if the triangles are small enough, the result is indistinguishable from traditional texture mapping, without the problems associated with point rendering. As a curiosity it is also possible to use an enhanced “relief” texture map with depth information to render three dimensional objects, see [170].

Since point rendering has a tendency to “grow” the edges of an object because of the size of the points, it might be necessary to trim the border. One possibility is to do it by adapting the silhouette clipping method described in [197] for this purpose. Although it is designed for polygon rendering, the silhouette edge database may also be applied to point rendering to allow rendering using fewer larger points which are protruding over the silhouette edge, and then applying silhouette stencil clipping to mask away pixels outside the object silhouette boundary.

3.2.10 Half-plane edge functions

While Hybris relies on forward differencing along triangle edges, it uses plane equations to represent shading gradients inside the triangles. Plane equations may also be used for representing the triangle edges.

Rendering using half-plane edge functions have been used earlier by the Pixel-Planes architecture [64, 65], which implements a massive processor-per-pixel VLSI array to simultaneously evaluate a linear function (3.3) at every pixel.

$$F(x,y) = Ax + By + C \tag{3.3}$$

The VLSI array allows Pixel-Planes to render shaded triangles in time *independent* of the triangle area, although the processor utilization for small triangles is low. Equation (3.4) below shows how an edge function can be expressed using this type of linear equation.

Another interesting hardware rendering architecture described in [235] implements what is essentially a 32x16 pixel tile-based and serialized version of the Pixel-Planes architecture, which uses sequential direct evaluation of linear functions for each pixel. The PowerVR [188] architecture possibly also uses linear function evaluation, but little is known about the microarchitecture of the commercial PowerVR architecture other than a marketing name; "infinite planes".

An alternative algorithm for half-plane edge function based rasterization using forward differencing is covered in Pineda's paper [184], and has been explored further in [199] as well, and implemented in the Neon [140, 141, 139] and in the Silicon Graphics RealityEngine [5].

Based on [184, 139], an edge function $E(x,y)$ for an edge from (x_0,y_0) to (x_1,y_1) can be described as

$$\begin{aligned} \Delta x &= x_1 - x_0 \\ \Delta y &= y_1 - y_0 \\ E(x,y) &= (x - x_0)\Delta y - (y - y_0)\Delta x \end{aligned} \tag{3.4}$$

This edge function is zero for (x,y) coordinates that fall exactly on the edge, positive for coordinates on the "right" side and negative for coordinates on the "left" side, defined by the direction of the $(\Delta x, \Delta y)$ vector. Figure 3.8 shows how the three directed edges of a triangle form three edge functions (3.4). For points inside the triangle, all three edge functions are positive, while outside the triangle one or more edge functions are negative. For points on the edge, one or more edge functions are zero.

While the edge functions may be evaluated directly for each pixel, Pineda [184] also described an incremental algorithm, based on the following difference equa-

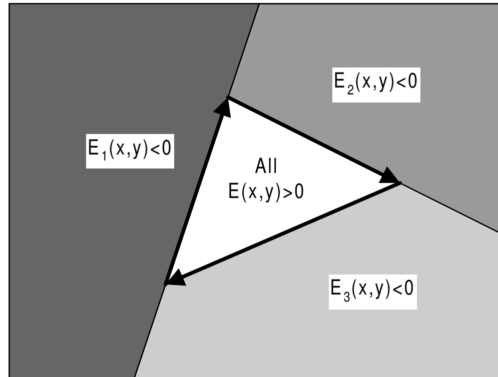

Figure 3.8: A triangle described by three half-plane edge functions.

tions:

$$\begin{aligned} E(x+1,y) &= E(x,y)+\Delta y \\ E(x-1,y) &= E(x,y)-\Delta y \\ E(x,y+1) &= E(x,y)-\Delta x \\ E(x,y-1) &= E(x,y)+\Delta x \end{aligned} \tag{3.5}$$

Using these stepping rules and given starting values for the three edge functions e.g. at one of the triangle vertices, we can traverse the pixels covered by the triangle using a suitable traversal pattern which iterates until one of the edge functions changes sign. This is essentially a variant of the flood fill algorithm.

Plane equations

As a related technique, linear functions can be used for direct evaluation for interpolation of shading, depth, texture and other rendering parameters across a triangle. The parameter p at (x,y) can be represented as:

$$p = Ax + By + C \tag{3.6}$$

The linear coefficients A,B,C can be derived from the three points (x_0,y_0,p_0), (x_1,y_1,p_1), (x_2,y_2,p_2) by solving this set of linear equations for (A,B,C)

$$\begin{bmatrix} p_0 \\ p_1 \\ p_2 \end{bmatrix} = \begin{bmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \end{bmatrix} \tag{3.7}$$

Now, by using linear plane equations to interpolate parameters we gain the advantage that direct evaluation of the interpolation can be done at any point using (3.6).

The linear equation (3.6) can also be expressed using a normal vector notation, as described in [84]. Given the three vertices $v_0 = (x_0, y_0, p_0)$, $v_1 = (x_1, y_1, p_1)$, $v_2 = (x_2, y_2, p_2)$, we form the two vectors connecting vertex v_0 to v_1 and v_1 to v_2. From this we can calculate the cross product which defines the normal vector $N = (v_1 - v_0) \times (v_2 - v_1)$

$$\begin{bmatrix} N_x \\ N_y \\ N_p \end{bmatrix} = \begin{bmatrix} x_1 - x_0 \\ y_1 - y_0 \\ p_1 - p_0 \end{bmatrix} \times \begin{bmatrix} x_2 - x_1 \\ y_2 - y_1 \\ p_2 - p_1 \end{bmatrix} \tag{3.8}$$

The plane equation is defined as:

$$N_x(x - x_0) + N_y(y - y_0) + N_p(p - p_0) = 0 \tag{3.9}$$

Isolating p gives the linear equation for p:

$$p = -\frac{N_x(x - x_0) + N_y(y - y_0)}{N_p} + p_0 \tag{3.10}$$

The derivatives of (3.10) are useful as they can be used for forward differencing evaluation of interpolation. The derivative along the x-axis is

$$\frac{dp}{dx} = -\frac{N_x}{N_p} \tag{3.11}$$

and along the y-axis

$$\frac{dp}{dy} = -\frac{N_y}{N_p} \tag{3.12}$$

note that both of these derivatives remain constant within a single triangle. p can be any parameter which can be linearly interpolated, e.g. colors (r, g, b, a), depth (z) or surface texture parameters (u, v, w). However this interpolation technique applies *only* to triangles, since three points specify a plane. Quads and larger polygons cannot be used with this technique without triangulation as no uniquely defined plane may exist.

Plane equations are used in the *Hybris* renderer to simplify parameter interpolation for triangles. The x-axis derivative (3.11) is used for forward differencing of parameters within each span and is calculated only once per triangle, reusing the same normal vector that is used for back-face culling in 2D screen space.

3.2.11 Packet data format for a triangle node

The tile renderer back-end expects a description of the triangles it has to render for each tile. A data packet protocol for transmitting and storing the information required to express a triangle is needed. To keep things simple the packet only defines a single triangle. Although using individual triangles may be redundant it keeps the design simple, as on-the-fly repartitioning of triangle strips and meshes into tiles is considered to be too complex (for now). Thus a multi-triangle packet format will not be studied. Some other architectures, e.g. InfiniteReality and Neon [158, 141] allow triangle strips, but they do not have to deal with the added complexity of tile partitioning. Single triangle packets may be compared to fixed-length instructions in RISC[1] architecture design philosophy, while multi-triangle triangle mesh or triangle strip packets may be compared to CISC.

There are several possible options for defining the packet data format, three of which will be identified and discussed below.

Raw. Stores start and end coordinates for each edge, i.e. the three vertices of each triangle are stored, possibly rounded to integer or sub-pixel coordinates. For each vertex, triangle parameter values are also stored, e.g. colours, texture coordinates and depth.

Differential. Stores only a differential expression of the triangle. Starting values for incremental linear interpolation of the edges as well as parameters along the edges are stored, along with differentials to add at each step in forward differencing interpolations.

Plane Equations. Stores half-plane edge functions describing the three edges of the triangle, as well as plane equations for each of the triangle's parameters.

Each of these three methods have advantages and disadvantages, depending on how we choose to partition the rendering architecture.

Raw triangle description

The first method, *Raw*, is conceptually the simplest as the three vertices of a triangle can be stored and transmitted virtually unchanged, except for some rounding and packing depending on the desired bit-count. When a raw triangle is received it is necessary to set-up the interpolation parameters, which involves a division for each edge to calculate edge slopes. A traditional full-frame rendering engine only reads each packet once, so it may not matter much if these divisions are done before

[1] RISC: Reduced Instruction Set Computer, CISC: Complex Instruction Set Computer, see [174].

or after transmission of the triangle. However a tile-based renderer must perform these edge slope divisions for every tile the triangle overlaps, making this approach questionable for tile engines. An advantage is good compression properties, as the vertex data size can be reduced by simple quantization.

Systems known to use Raw transmission of triangles include the Neon [140, 141], which allows quantization of data down to 12 bytes per vertex using fixed-point notations, resulting in triangle packet sizes down to 36 bytes in the simplest case. This allows transmission of up to 2.6 million triangles/s using 32 bit PCI. Other examples are [116, 115] and the Silicon Graphics GTX [4, 63] which internally uses a three-step decomposition into first raw triangles then raw edges and finally raw spans, while the SGI InfiniteReality [158] uses raw triangles (or triangle strips) directly on its internal bus.

Differential triangle description

The second method, *Differential*, performs the triangle set-up divisions required for edge slope calculations before storing the data in the triangle node packet. This approach is tied to an implementation which uses slope based interpolation either by direct evaluation or by forward differencing. A naive implementation would need to store interpolation start values for the first vertex of each edge, and compute parameter interpolation slopes for each span from the interpolated parameter values at the edges. Examples of the differential method are found in the 3Dlabs GLint, the 3dfx Voodoo Graphics accelerator and other early PC graphics accelerators without hardware support for triangle set-up. Most other graphics architectures transfer raw triangles, even though they use differential expressions internally, requiring them to do triangle set-up every time the triangle is read. Since a global framebuffer architecture only reads a triangle node once this can be an advantage.

Since we choose to support only triangles, parameters can be expressed by plane equations as discussed earlier. By storing the x-axis derivative of each parameters plane equation we can avoid a division per span. In addition this allows us to skip interpolation of parameters along edges at the end of the spans, or both at the beginning and end of spans if the y-axis plane equation derivative is also stored, although this will complicate incremental evaluation. A better approach is to allow an adaptive incremental interpolation direction for interpolating along spans. Since one side of a screen-projected triangle will always have only one edge, while the two other edges will be on the other side (one might be a top/bottom edge), we can start span interpolation from the side of the triangle which has one edge. The advantage is that parameter interpolation start values and slopes are only required for one edge, saving both space and calculations, but the span interpolator must allow both left-to-right and right-to-left incremental evaluation, and a method

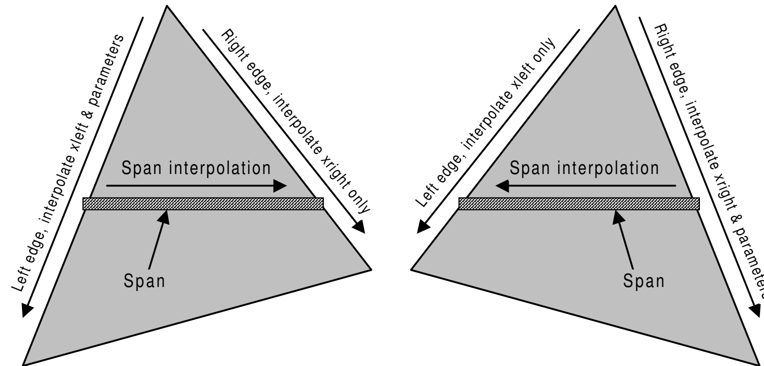

Figure 3.9: Adaptive bidirectional incremental interpolation for triangle rendering. Left: Left-to-right incremental interpolation. Right: Right-to-left incremental interpolation.

for identifying the direction is needed. Figure 3.9 shows how the adaptive bidirectional incremental interpolation method works. A future implementation of the adaptive interpolation direction method might extend the concept to allow both top-to-bottom and bottom-to-top scanline interpolation.

By storing differential expressions we can reduce the number of necessary operations per tile, as triangle set-up per tile would be magnified by the bucket sorting overlap factor. Combined with the optimizations above, the differential method is judged to be a good data format for the triangle heap.

To summarize, the triangle heap node packet data format in Hybris uses differential expressions, some of which are based on plane equations. There seems to be a tradeoff between computing triangle setup before or after sending the triangles over a redistribution network in a parallel renderer, depending on whether we perform set-up once before writing the node or possibly multiple times after reading the node in the tile renderers. The number of writes vs. the number or reads ratio influences this, and depends on the bucket sorting overlap factor.

As an example, the Hybris dual CPU parallel software implementation uses two writers and two readers, where each reader may read a node multiple times depending on the overlap. As the triangle set-up calculations are stored and reused this architecture works well in software. The hardware tile renderer back-end implementations benefit from not having to do triangle setup in the back-end, simplifying their design greatly.

Plane equation triangle description

The last method, *Plane Equations*, performs evaluation of the parameters required to express edge functions and plane equations prior to storage. The edge functions and plane equations are subsequently used to render the triangle using either direct evaluation or an incremental algorithm such as Pineda's [184]. An advantage of using edge and plane equations is that their parameters can be calculated completely without use of divisions. Since edge and plane equation parameters can be calculated directly from the raw triangle vertices, implementations tend to use on-the-fly plane equation setup from a raw triangle description. The example from before, Neon, transmits raw triangles but uses edge function setup and incremental evaluation internally. The design in [235] also transmits raw triangles, although it also uses internal buffering of plane equation parameters.

The best match for the plane equation description model is the SGI RealityEngine [5] which transmits plane equation packets on its internal triangle bus. However this approach was abandoned in the InfiniteReality [158] which transmits raw vertices on its internal redistribution bus, forming triangles at the receiving end by interpreting the vertex stream as triangles or triangle strips. Pixel-Planes [64, 65] first evaluates the plane equation parameters and then transmits them to an array of pixel processors, each of which evaluate the plane equation in parallel. PowerVR [188] is suspected to transmit plane equations, although nothing concrete about this has been published.

3.3 Object partitioning

Object-parallel rendering of 3D objects requires that the objects are distributed across all the processing elements of the renderer. A straightforward method for doing this is by using round-robin distribution of individual triangles. While this method is quite simple and achieves good load balancing with equal sized triangles, it has to treat each triangle as an individual unit. Since triangles in a 3D triangle mesh object often share vertices with their neighbors, chopping an object into individual triangles removes the advantage of shared vertex processing. Figure 3.10 shows how a vertex can be shared by multiple triangles. A central vertex in a triangle mesh is typically shared by six triangles, while a vertex located on the edge of a mesh is shared by fewer triangles.

To maintain vertex sharing after partitioning an object, we need a partitioning algorithm which is able to group triangles which share vertices. By using sufficiently large partitions, good vertex sharing can be achieved.

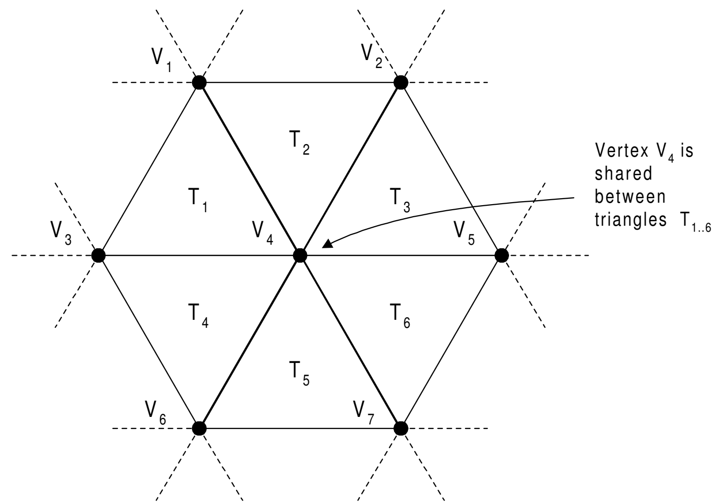

Figure 3.10: Example of vertex sharing in a triangle mesh. Each central vertex is typically shared by six triangles.

3.3.1 Triangle strips

To improve vertex sharing *triangle strips* [152] have traditionally been used in graphics hardware [158, 140]. Triangle strips provide a means for specifying a sequence of connected triangles. Figure 3.11 shows an example of a triangle strip, where the triangles $T_1 ... T_5$ are specified by the vertex sequence $v_1, v_2, v_3, v_4, v_5, v_6, v_7$. Triangle i in this sequence is specified by v_i, v_{i+1}, v_{i+2}. Some triangle strip implementations include a *swap* bit to allow reusing a vertex by swapping v_i & v_{i+1} as shown with v_5 in figure 3.11. Modern APIs such as OpenGL and DirectX chose not to implement the swap bit and requires v_5 to be specified multiple times. Closely related to a triangle strip is the *triangle fan* which shares the first vertex among all triangles, using the sequence v_0, v_{i+1}, v_{i+2}.

Without swapping, a long triangle strip (or triangle fan) requires

$$\lim_{n \to \infty} \frac{3 + (n-1)}{n} = 1 \tag{3.13}$$

down to one vertex per triangle. Because of this vertex reuse and the architectural simplicity of only buffering the two previous vertices, triangle strips and fans are a popular feature in many hardware renderers. However even in their optimal case they only share half as many vertices as in a triangle mesh, meaning that every vertex in the triangle mesh must be specified at least twice. The *generalized triangle strip* [43] extends the swap bit idea and uses additional bits to index further back

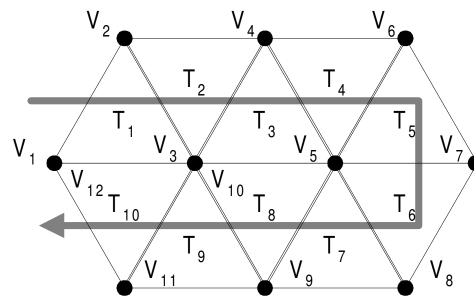

Figure 3.11: Example of a triangle strip. The grey arrow shows the triangle rendering sequence T_1 ... T_{10}. Each vertex is typically only shared by three triangles. Note that e.g. T_1 and T_{10} cannot share vertices. v_5 is a *swap* vertex in this example.

in a longer vertex reuse buffer, allowing better vertex sharing. This work is further extended in [33] to improve geometry compression. In [92] generalized triangle strips are explored in order to improve vertex cache locality for modern graphics accelerators.

Algorithms for automatic generation of triangle strips from a triangle mesh are described in [152, 92]. These strip generation algorithms may be used to partition a triangle mesh into a set of triangle strips for distribution over parallel renderers. However, optimal triangle strip generation algorithms are NP-complete [152] and furthermore the strips generated have widely varying lengths.

3.3.2 Indexed triangle meshes

A popular method for sharing vertices in a triangle mesh is the indexed triangle mesh. The indexed triangle mesh uses two arrays: The first array (a vertex buffer) contains all the shared vertices, while the second array (index buffer) describes each triangle using three indexes into the vertex array. The VRML `IndexedFaceSet` node [28] is a typical structure used for representing a triangle mesh. In an indexed triangle mesh each vertex can be shared by many triangles (figure 3.10), up to six depending on the mesh topology. This implies that the number of vertices needed per triangle goes towards $1/2$ as the mesh size increases, meaning that each vertex may be specified only once. An exception to this rule is the vertices at mesh edges, where a vertex may be used for only one triangle.

Unfortunately, each triangle may index vertices randomly anywhere in the vertex array. Because of this it can be difficult to say anything about the cacheability of vertex data, as well as predict anything about the spatial- and data-coherence of

how the triangles in the object are rendered to the screen.

While the indexed triangle mesh provides the optimal reuse of vertices, a large object may be difficult to accommodate in a graphics system. To minimize the random access properties it would be useful to determine local groups of neighboring triangles in order to partition the triangle mesh.

3.3.3 Triangle mesh partitioning with MeTiS

Rather than reinvent the wheel and implement our own mesh partitioning algorithm, the excellent general purpose free graph partitioning package MeTiS (version 4.0) [113] can be used. The graph multilevel multi-constraint graph partitioning algorithms implemented in MeTiS are presented in [114]. MeTiS supports k-way mesh partitioning while minimizing the connectivity of subdomains. While originally intended to provide data partitioning for finite element or finite volume methods, it it also applicable for partitioning the dense triangle meshes used in computer graphics today. Graph partitioning algorithms and their uses in scientific computing are covered in [201, 202].

When working on a triangle mesh, MeTiS only needs a list of the nodes (vertex index numbers) used by each element (triangles). Since each vertex is only specified by an index value, this means that the mesh partitioner does not know about position of each vertex, nor does it know anything about the size of each triangle. Because of this, MeTiS works best with triangle meshes of approximately equal sized triangles. This is not a huge problem since the trend in computer graphics is heading towards denser triangle meshes. For our application MeTiS uses a balanced k-way partitioning algorithm which minimizes the number of edges that straddle partitions (edge-cut), which helps to optimize vertex re-use within each partition and forms nice small regular sized partitions.

After an object-partitioning has been determined, it is quite straightforward to build a set of 3D objects, one for each partition, while duplicating only those vertices needed across partition boundaries. This approach allows up to two times better vertex reuse ratio than normal triangle strips. The algorithm complexity is lower than generalized triangle strip generation algorithms which must also determine a triangle rendering sequence [92, 43, 21].

By selecting an appropriate partition size it is possible to achieve good locality of reference to vertices, while maintaining a high vertex reuse ratio. Parameters such as processor data cache size, desired parallelism, communication cost as well as data dependencies such as triangle size distribution all influence the perfect partition size. Using a partition size of 500 triangles gives a data cache memory footprint of about 8 kbytes, assuming each vertex uses 32 bytes and is reused 6 times. In a software implementation for e.g. a Pentium III PC this allows temporary trans-

formed vertex data to remain in cache memory until they are needed for triangle setup.

In figure 3.12 two versions of the Stanford Buddha (from [208]) were partitioned using MeTiS with a 500 triangle partition size. Figure 3.13 is a zoom-in showing the area relationship between a 32x32 pixel tile and the rendered triangle partitions. Note how several partitions may fit entirely within one tile. This improved coherence between triangle partitions and tiles helps maintain cache data coherence and minimize memory paging. In this sense, triangle mesh partitioning is in fact a sort-first architecture, if complete sorting is not required. As Hybris uses tiled sort-middle later in the pipeline, mesh partitioning is quite useful in the architecture as a complete partition may fit in one tile. For a hardware implementation of the Hybris front-end pipeline, mesh partitioning allows a dramatic size reduction of the temporary transformed vertex array buffer. Rather than needing to accommodate the entire transformed vertex array of the object, possibly millions of vertices, we now only need a vertex buffer large enough for one object partition, in this case about 8 kbytes. A hardware implementation of the front-end graphics pipeline may use on-chip memory for this buffer.

The actual size of the vertex array depends on the level of vertex sharing present in the original unpartitioned 3D object. In the optimal case each vertex is used by 6 triangles, giving twice as many triangles as vertices in each partition. However if the original object is a pathological collection of unconnected triangles and no vertex sharing is available, the number of vertices will be three times the number of triangles. Either the vertex buffer must be designed to handle this worst case or further sub-partitioning for this worst case must be made. The 8 kbytes buffer size mentioned earlier assumes good vertex sharing, which is also why the the partition size is 500 triangles and not 512, compensating for lower sharing at the partition boundary.

As a consequence of partitioning the object, and reducing the size of the vertex buffers, we can also reduce the size of vertex indexes. Since we know the maximum number of vertices, we can trim the number of bits per vertex index in the triangle description. Since three vertex indexes are needed, going from 32 bits to 10 bits per vertex index potentially reduces the size of the index array by 68%.

Recently other researchers are beginning to use MeTiS in computer graphics, e.g. [111, 112] uses MeTiS for preprocessing to enable efficient mesh geometry compression.

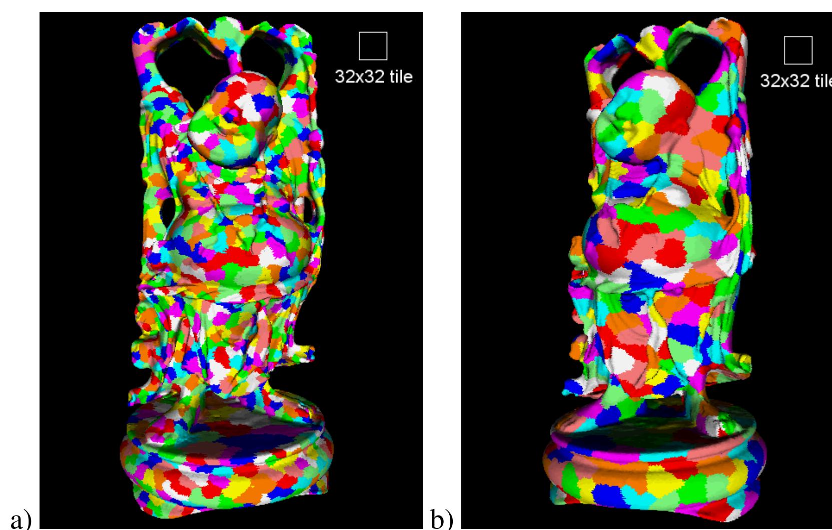

Figure 3.12: Two renderings of the Stanford Buddha, showing the result of MeTiS based object partitioning using a partition size of 500 triangles. To the left is the full Buddha model of 1,087,716 triangles and to the right is a decimated version with 293,232 triangles. The white square shows the size of a 32x32 pixel tile.

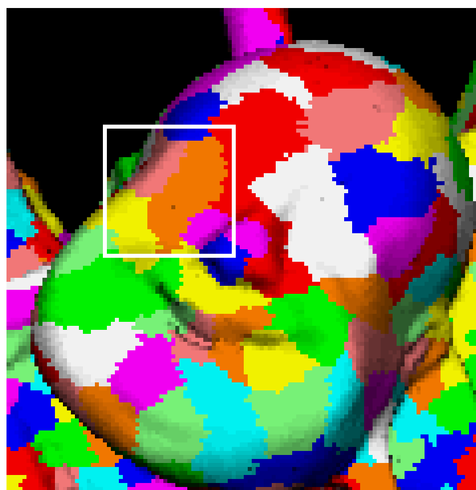

Figure 3.13: Zoom in on the left part of figure 3.12. The square represents a tile of 32x32 pixels. Notice how several partitions fit entirely inside the tile.

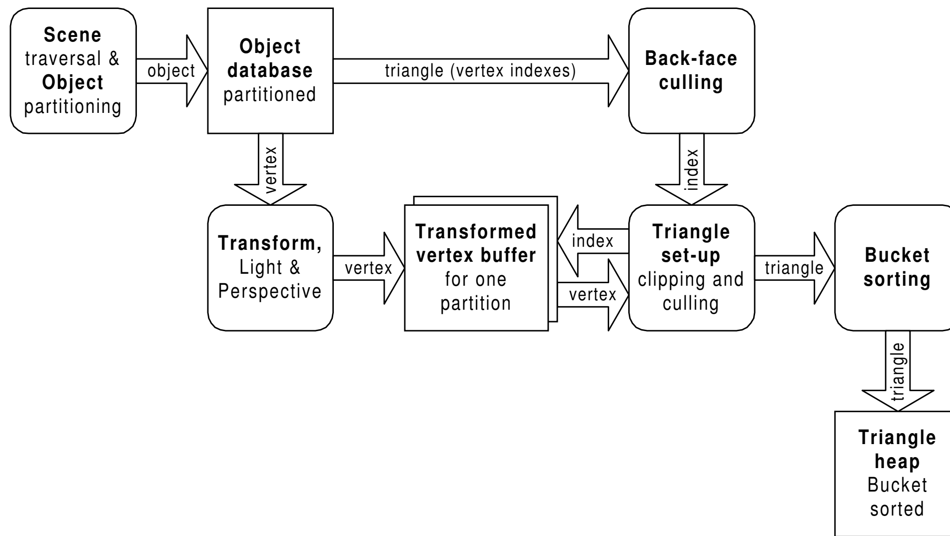

Figure 3.14: Front-end graphics pipeline, or geometry engine. One object partition is processed at a time. A transformed vertex buffer allows optimal vertex re-use within one partition. The triangles that survive clipping and culling are bucket sorted and sent to the triangle heap.

3.4 Partitioned object-parallel renderer front-end

Many of the techniques described earlier in this chapter are implemented in the Hybris architecture. This section describes the partitioned object-parallel front-end graphics pipeline in the Hybris architecture.

The front-end graphics pipeline traditionally performs geometric transformation, lighting, clipping and perspective projection. If implemented in hardware the front-end graphics pipeline is often referred to as a “geometry engine” or “hardware T&L unit”. The operations performed in the front-end creates screen-space vertices and graphics primitives ready to be rendered in the back-end graphics pipeline, which renders triangles to pixels in an image-parallel graphics architecture. Thus the front-end performs the work before sort-middle redistribution in a parallel graphics architecture.

While a single front-end rendering pipeline by itself is conceptually the same whether the object database is partitioned or not, some additional optimizations such as reduced storage and indexing range as well as improved data locality for caching becomes possible with partitioning. Additionally a partitioned object database allows parallelism by processing multiple object partitions in parallel us-

ing multiple front-end rendering pipelines. Figure 3.14 presents an overview of a single front-end rendering pipeline, which is able to process one partition at a time. A hardware implementation may use a double buffered vertex buffer to allow pipelined processing of several partitions. Figure 3.15 shows how two geometry engines are used to render two object partitions in parallel, where each pipeline processes its own set of object partitions, in this case odd or even. Using two triangle heaps can improve performance, e.g. in dual CPU implementations by allowing the two separate CPU caches to operate without invalidating each other.

Some operations that are performed in the front-end pipeline operate on a per-vertex basis (transformation, lighting and projection) while other operations (back-face culling, clipping and triangle set-up) operate on a per-triangle basis. For simplicity the *only* graphics rendering primitive represented in Hybris is *triangles*. By using triangles, other more complex graphics primitives (e.g. quads, polygons, quadric surfaces, Bézier patches, NURBS surfaces and subdivision surfaces) can be represented by subdividing them into a set of triangles. Triangle strips can also be subdivided into individual triangles for simplicity. Other graphics primitives such as lines and points are also representable as triangles. Lines are represented as a rectangle using two long thin triangles. Points are represented as a square by two small triangles. An alternative technique to allow points to be directly expressed and rendered as "tweaked" triangles is described in section 3.2.9.

While the object partitioned rendering front-end depends on a higher level rendering control program to manage the object partitions, the front-end pipeline does not need to be concerned by this, and only has to deal with the individual partitions.

If an object partition is determined to be visible (i.e. located somewhere in the view volume, not occluded by other objects and not back-facing) then it is sent to the rendering front-end. The first process in the front-end is an object-space to world-space to screen-space transformation of each vertex. The transformation is performed in one step using a matrix representation of the combined object-space to screen-space transformation. After transformation of the vertex coordinates the vertices are lit using a lighting model.

The lighting model relies on vertex normals which are pre-computed and stored in the vertex records. These vertex normals are represented in object-space coordinates. To avoid transforming the vertex normals to world-space in order to perform lighting calculations, the relevant world-space lighting parameters are transformed from world-space to object-space *before* processing the object partition. This allows lighting calculations to be performed in object-space. Perspective projection is necessary to determine the screen-space pixel coordinates of the vertices, and is performed after transformation and lighting. All of these steps can be performed in a single combined transformation and lighting datapath, and is expressed in C using a single loop.

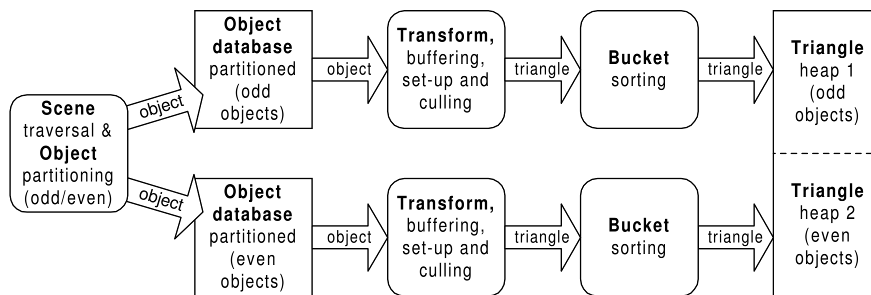

Figure 3.15: Parallel front-end rendering pipeline. Each geometry engine works on its own set of objects, in this example odd or even numbered object partitions. Two triangle heaps are used to improve performance.

After these steps, the vertices are ready for triangle set-up, except for one special case, *clipping*. If necessary, a triangle is clipped against the view volume using the clipping algorithm described earlier.

3.5 Triangle setup and bucket sorting

Triangle setup in a sort-middle architecture can be performed either before or after bucket sorting. Selecting whether to use one or the other sequence depends on several factors, such as the level of parallelism in front- and back-end as well as load balancing between front- and back-ends. Since bucket sorting with a triangle heap is a synchronization barrier between the front- and back-end rendering pipelines, the choice of where to place triangle set-up is effectively whether it should part of the front-end or the back-end. Currently Hybris performs triangle set-up before bucket sorting and after transformation and clipping in the front-end, and as such is located in the front-end.

Figure 2.12 and 2.13 from chapter 2 (page 34 and 34) illustrates how the triangles are mapped to the bucket sorted triangle heap using a two dimensional hash table. The hash table is simply a 2D array of pointers, one for each tile, which point into the triangle heap. The triangle heap itself optimizes the dynamic assignment of memory locations to triangle nodes by managing a linked list of triangle heap tile buffers for each tile. Figure 3.16 shows an example of how the tile buffers are managed in the bucket sorted triangle heap. Each of these buffers are the size of an SDRAM page, typically 4 kbytes, and are also aligned in memory to fit in one SDRAM page, optimizing memory access within one tile. Each buffer holds a header (number of triangles and a pointer to the next buffer) and 63 triangle

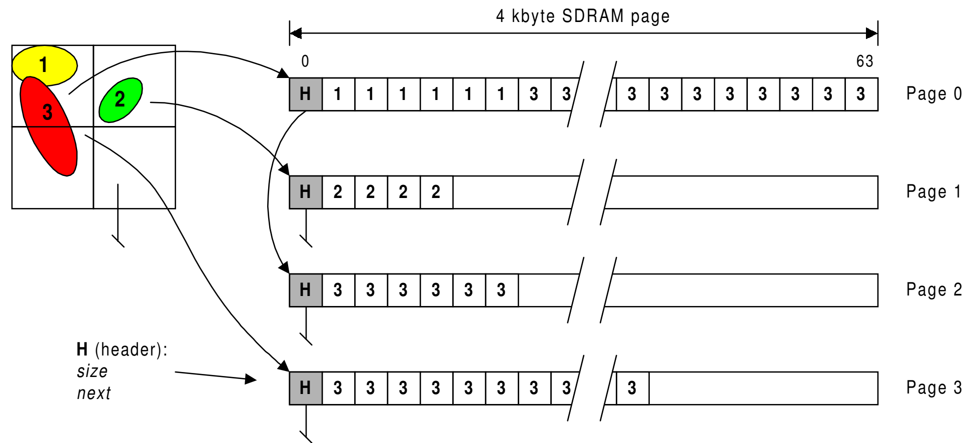

Figure 3.16: Memory management of triangle nodes in the bucket sorted triangle heap. Triangles are allocated in buffers of 63 triangles. A linked list of buffers are used, if space for more triangles are needed in a bucket. In this example the triangles of object 1 go to the first buffer, page 0. Object 2 is placed in page 1. The triangles of object 3 overlapping the upper left tile fills up page 0 and is continued in page 2. The triangles overlapping the lower left tile goes to page 3. The lower right tile is empty and nothing is allocated for it.

description nodes, i.e. 64 bytes per triangle node. If a tile buffer is full and more storage is needed, a new buffer is allocated from the heap and linked into the linked list of triangle buffers. Note that actual implementations also use a *last* pointer to quickly locate the currently active triangle buffer for a given bucket. This triangle heap management technique has proved to work well in practice, and can be implemented in software as well as for hardware using SDRAM memory directly, as described in [71, 72]. Each buffer also provides a nice communication block size for transmission to rendering hardware, e.g. over the PCI-bus to the FPGA implementation of the tile renderer back-end.

The triangle heap node packet data format is a collection of interpolation start values, slopes and plane equation differentials organized using a compact 64 byte (512 bit) C data structure. The data structure is shown in figure 3.18.

Figure 3.17 shows the different cases that the triangle setup process must consider when calculating slopes and differentials for generating this triangle node. Because of the adaptive bidirectional span interpolation (see figure 3.9) we only store parameter interpolation values for one start edge. In the tile renderer, the span interpolation direction is determined from the triangle node by the compar-

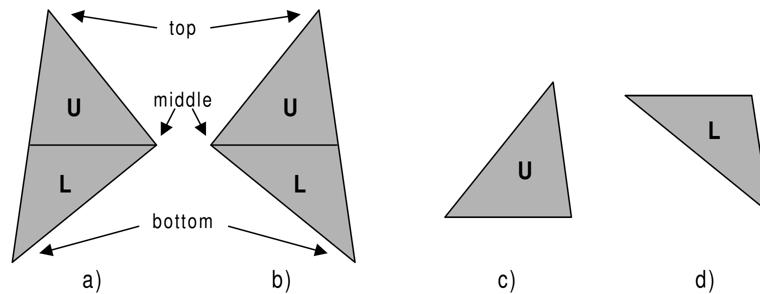

Figure 3.17: Four triangle cases considered in the triangle setup stage. a) Left-to-right triangle with upper and lower trapezoids. b) Right-to-left triangle with upper and lower trapezoids. c) Upper trapezoid only triangle. d) Lower trapezoid only triangle.

```
struct THnode {
  int z;        // depth interpolation start value
  int z_inc;    // depth interpolation slope for start edge
  int dzdx;     // depth differential for span interpolation
  int xleft;    // x start value for left edge
  int xright;   // x start value for right edge
  int xleft_inc;    // x slope for left edge
  int xright_inc;   // x slope for right edge
  int xmiddle; // x start value for middle edge
  int xmiddle_inc; // x slope for middle edge
  int s;        // color interpolation start value
  int s_inc;    // color interpolation slope for start edge
  int drdx;     // color differential for span interpolation
  int y1;       // start scanline
  int y2;       // middle scanline
  int ye2;      // end scanline
  int pad;      // pad to 64 bytes size
};
```

Figure 3.18: Triangle node C data structure. Each triangle node in the triangle heap is stored as a packet with this data structure.

ison `if (xleft_inc < xright_inc)` as left-to-right, else the direction is right-to-left.

This data structure for the triangle node uses `int`'s for all the parameters, which is convenient in a C program, although they actually represent fixed-point values. However many of the parameters may be reduced in size to create an even more compact representation, depending on the precision needed for rendering. In order to allow interpolation over a sufficiently large screen area, we need 12 fractional bits in the fixed-point representation of the slopes and differentials.

Internally in the FPGA hardware implementation [207] the triangle node is represented by 348 bits (44 bytes), by using 20.12 fixed-point for depths, 12.12 for x values, 8.12 for colors, and 11 bit integers for y values. A total of 333 bits (42 bytes). The remaining 15 bits are used to identify which tile the triangle belongs to.

However, we can do even better than that, since we do not absolutely need *all* the extra fractional resolution to specify interpolation start values, we may trim those to e.g. 4 fractional bits for x subpixel precision, and no fractional bits for colors. For depth we need about 24 bits [229] so this can also be trimmed. Applying these observations, we can save another 44 bits, reducing the total to 289 bits. Finding one more bit to kill, we can reduce the total to 288 bits (36 bytes), the same size as the Neon's smallest packed raw triangle format [141]. In comparison the early 3dfx Voodoo graphics processor used 144 bytes per triangle.

Performing triangle set-up *before* bucket sorting has the advantage of enabling exact bucket sorting, which can minimize the bucket overlap factor for large triangles, as discussed in chapter 2. In the current Hybris architecture though, bucket sorting is done using the triangle bounding box as key, placing a triangle into all buckets touching the triangle bounding box. Since Hybris performs triangle set-up before bucket sorting, exact bucket sorting may be used in the future to reduce the overlap factor for large triangles covering many tiles.

Alternatively, triangle set-up may be performed *after* bucket sorting. This has the advantage that set-up calculations are being performed in the back-end pipeline, benefiting from any available parallelism. According to [116], moving calculations from the front-end to the back-end will also help reduce latency. Some drawbacks of set-up after bucket sorting is that exact bucket sorting becomes more difficult to implement, although the method described in chapter 2 and [169] is applicable. Another difficulty is that a completely different triangle data format is needed in the bucket sorted heap. The workload of triangle set-up will also be duplicated by every tile worker rendering a part of the same triangle in a parallel tile rendering back-end.

In the next chapter we will look at how multiple bucket sorted triangle heaps are used to implement a scalable graphics system (figure 4.3), which allows multiple

object-partitioned geometry engines to work in parallel in the front-end pipeline and multiple tile engines to work in parallel in the back-end pipeline.

3.6 Tile-based image-parallel renderer back-end

The tile-based rendering back-end in *Hybris* is designed to simplify calculations and localize memory references. A small tile shaped region of pixels are directly addressable in a small virtual local framebuffer. The size of this region must be small enough to fit in on-chip RAM or in CPU cache but large enough to prevent too much overlap. (See the discussion of bucket sorting overlap in section 2.4). For input data, the tile renderer needs to be given the data relevant for one tile, which is the contents of one bucket in the bucket sorted triangle heap. The result after rendering all triangles from the bucket is a 32x32 pixel tile which must be placed at the correct location in the full-screen framebuffer.

3.6.1 Pipeline stages

The tile rendering engine's functional specification is written in C and is expressed with nested loops. The loops handle initialization of per triangle parameters, iteration over the number of spans in the triangle with setup of span parameters, and finally iteration over the number of pixels in each span where color and depth values are evaluated and assigned. From an architectural point of view, the nested loops can conveniently be thought of as a four stage pipeline. The first pipeline stage *Setup Triangle* calculates per triangle parameter setup based on the current tile parameters and the input. This involves adjusting the forward differencing y-interpolation start values to match the range of scanlines in the tile. This is done using scissoring in the vertical direction and direct evaluation as described in section 3.2.2, which requires a multiplication per parameter. Setup Triangle also determines the span interpolation direction, left-to-right or right-to-left.

Results from Setup Triangle are then forwarded to the next pipeline stage *Draw Triangle*. Draw Triangle iterates over the active scanlines in the triangle for the current tile, while updating y-interpolation parameters using forward differencing. At each scanline a span of pixels is determined, which is sent to the next stage, *Setup Span*. Setup Span handles per span parameter setup, which involved adjusting forward differencing x-interpolation start values to match the horizontal range of pixels in the tile. This also involves a scissoring operation, but this time for the horizontal interpolation direction. For each span, data is forwarded to the *Draw Span* stage. Draw Span draws the pixels in the span by iterating over the number of active pixels in the span. To draw pixels, Draw Span uses a 32x32 pixel local

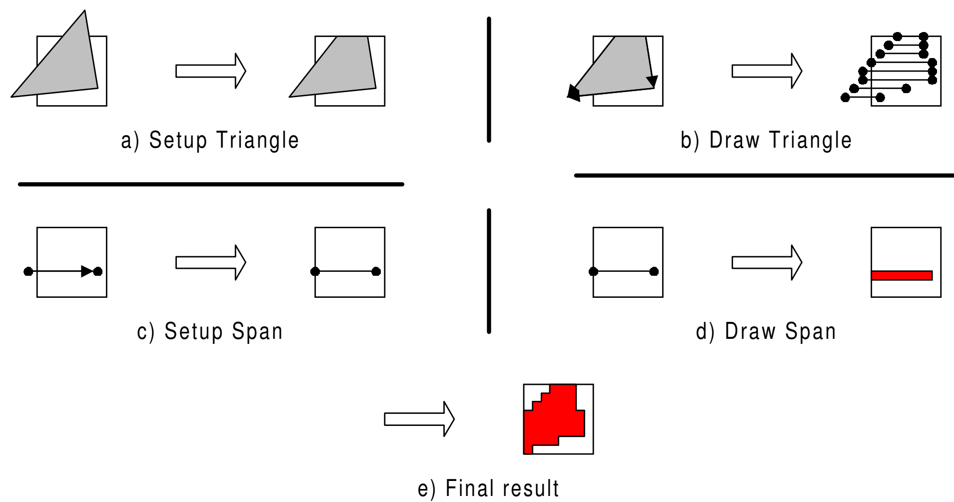

Figure 3.19: Processes in the tile rendering engine back-end pipeline. a) Setup Triangle adjusts y-parameters to fit tile. b) Draw Triangle iterates over the active scanlines, generating spans. c) Setup Span adjusts x-parameters to fit tile. d) Draw Span iterates over the active pixels in the span, performing per-pixel shading and depth testing. e) Final result for drawing one triangle.

memory buffer, from which colour and depth values are read, modified and written back. Note that if needed this triangle drawing algorithm can also be used to draw points, lines and trapezoids.

An overview of the processes involved for scan converting a triangle is shown in figure 3.19. In a hardware implementation it is convenient to combine the Draw Triangle and Setup Span stages, as some control logic can be combined. Some FIFO buffering between the stages is useful for load balancing the pipeline. This is because the Draw stages which involve iteration have a widely varying execution time which depends on the area of the triangle. For each span the number of drawn pixels may vary between 0 and 32, and for each triangle the number of active spans may also vary in the range 0–32. Figure 3.20 shows an architectural overview of the tile rendering engine back-end pipeline, with FIFO buffers placed between iterating pipeline stages to help average out any load imbalances.

For hardware implementation the two local tile pixel buffers are double buffered using a 2x2 crossbar switch to allow one buffer to be used for rendering while another can be used for I/O operations. Additionally they use a dual ported data interface to accommodate a pipelined read-modify-write cycle using one port for reading and another for writing. In the I/O operation mode the dual ported data

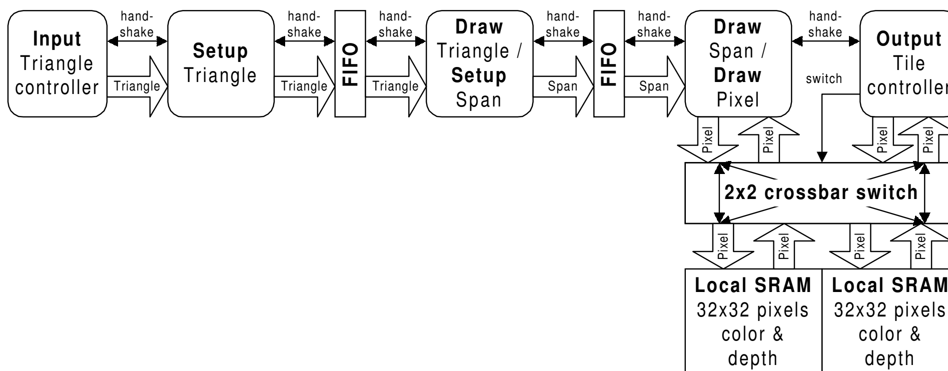

Figure 3.20: Architectural overview of the tile rendering engine back-end pipeline. FIFO buffers are placed between iterating pipeline stages to help average out load imbalances. The double buffered tile buffer allows the tile engine to render one tile while the previously rendered tile is being copied to the global framebuffer.

interface is used to read data for output while writing is used for resetting the pixels values.

3.6.2 Load balancing the back-end rasterization pipeline

While load balancing a pipeline using FIFO buffers is a reasonable approach, it may be quite expensive since a FIFO buffer requires memory proportional to how deep it is. Furthermore, a FIFO buffer is only useful for localized re-balancing of the pipeline as it can only hide the extra processing time for a few large triangles if smaller triangles also exist.

The pipeline stages of the rendering engine back-end must be balanced so on average they perform their tasks in equal time. Assuming each stage is driven by the same clock signal then the span drawing time, span setup time, triangle rasterization time and triangle setup time should be equal. This may be difficult to achieve as very small triangles are needed to allow the span and pixel loops to finish in one clock cycle.

For larger triangles where pixel processing time requires more time than setup there are two options: We can slow down the setup stages e.g. with serial multipliers and/or we can speed up pixel rasterization by parallelizing it. Slowing down the setup stages is not very desirable as the case for small triangles will be slowed down. Speeding up rasterization sounds better, as is reduces the time required for larger triangles, although the drawing time for rasterizing small single pixel trian-

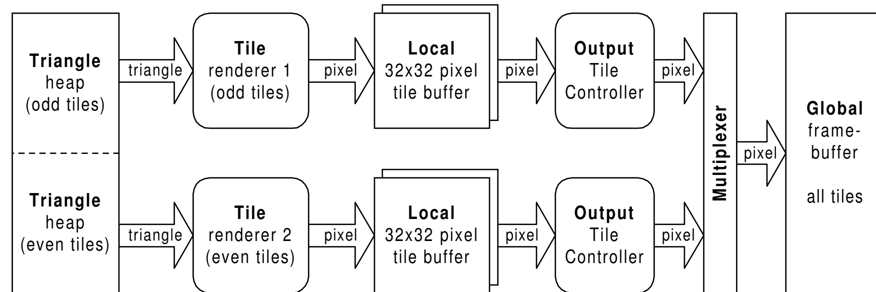

Figure 3.21: Parallel tile rendering. Two tile rendering engines shares the workload of rendering a complete frame. Each tile engine renders its own set of tiles (e.g. odd or even) independently of the other tile engine.

gles will not be improved.

3.6.3 Parallel tile rendering

Parallel tile rendering is possible by having multiple tile rendering engines, each one rendering its own set of tiles independently of the others. Figure 3.21 shows how two tile renderers work on their own set of tiles. For example an odd/even tile distribution can be used to form a checkerboard pattern for the workload distribution (see figure 4.3 on page 107). This kind of sort-middle redistribution of triangles requires that each tile rendering engine reads data from all triangle heaps. However, as each tile renderer only reads data for its own set of tiles, the triangle heaps can be organized to support this. Triangles for a given tile are stored in the same bucket using a list of buffers mapped to SDRAM pages (see figure 3.16). As an SDRAM chip [147] has four banks of pages, up to four pages may be open for access at the same time, which can be used to address up to four triangle heap buffers without excessive SDRAM page swapping overhead.

3.6.4 Image composition of tiles

When rendering of a tile has completed, the tile must be sent to the framebuffer in order to build a complete image, one tile at a time. If multiple parallel tile renderers are used to render different tiles, a multiplexer is used to collect the results. However we might also want to implement a full image composition sort-last style parallel architecture, e.g. by assigning two tile renderers to the *same* tile. Figure 3.22 shows an example of this. Each of these overlapping tile renderers must render a different set of triangles (e.g. odd/even). After they complete rendering of

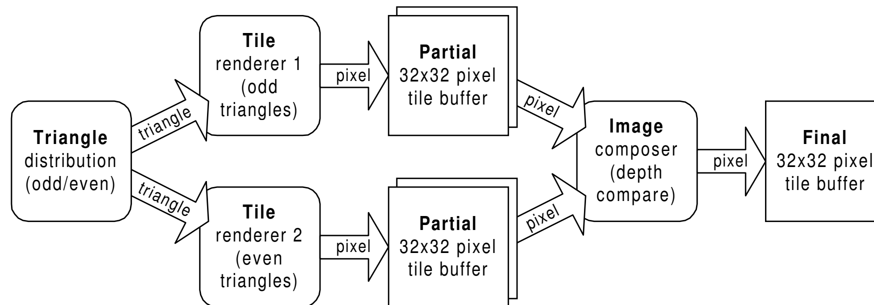

Figure 3.22: Image composition of tiles. Two tile rendering engines share the workload of rendering one tile. Tile renderer 1 renders odd triangles and tile renderer 2 renders even triangles. The image composer compares the depths of overlapping pixels and stores the nearest in the final tile.

the two sub-images, a sort-last image composition network as described in chapter 2 can be applied to compose the two tile images into a final tile, to be stored in the framebuffer. This image composition network can be implemented efficiently on-chip in a hardware implementation. A larger number of partial images may be combined into one image by using either a binary tree [153] or a linear pipeline [156] of image composers. For image composition to work efficiently it needs a large enough number of triangles per tile to keep all tile renderers busy. Also, a uniform triangle size distribution is needed for load balancing. Note that image composition does not handle transparent surfaces correctly, because it compares pixels and not fragments.

Combining parallel tile rendering and image composition

A sort-middle image-parallel tile renderer can include image composition sort-last parallel rendering on a per-tile basis by substituting each tile renderer with two partial tile renderers. Each of the tile renderers then uses image composition of the partial tiles when storing the final tiles in the global framebuffer. An example of such a hybrid sort-middle /sort-last parallel tile renderer is shown in figure 3.23. A scanline renderer (i.e. one pixel high tiles) version of this hybrid parallel architecture was presented earlier in [89].

3.6.5 Interleaved pixel parallel back-end rasterization pipeline

The tile renderer can be parallelized in various ways by combining multiple tile renderers in a sort-middle or sort-last architecture, or possibly a hybrid as shown

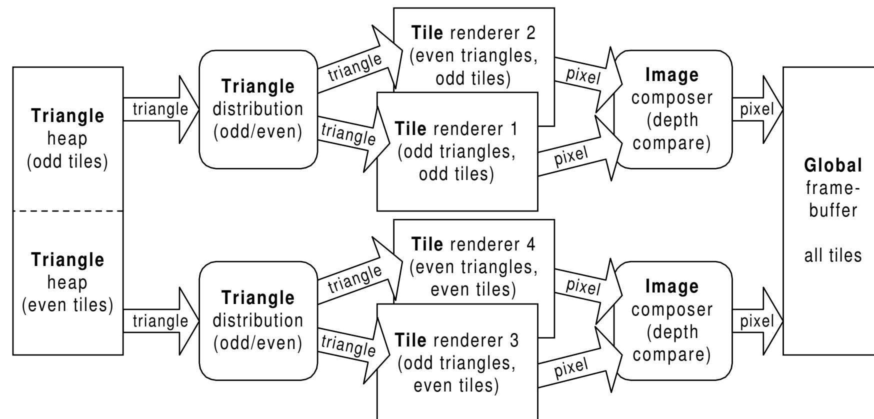

Figure 3.23: Combining parallel tile rendering and image composition to create a 4-way hybrid sort-middle / sort-last parallel renderer.

earlier. However, it is also possible to look within the tile renderer itself, and apply parallelism on a lower pixel-parallel level.

Looking at other parallel architectures, the Pixel-Planes [65] architecture is interesting as it can rasterize a single triangle in constant time *independent* of its size. Constant time triangle rasterization is accomplished with a processor-per-pixel array. While guaranteeing constant time execution, unfortunately the processor array is not well utilized when drawing small triangles.

A viable architecture for flattening the triangle size dependent differences in execution time in the tile renderer back-end is an interleaved span *and* pixel processing architecture. Probably the best known example which does this, is the Silicon Graphics architecture as described in [4], see also the discussion of figure 2.9 on page 27. The back-end of the SGI GTX architecture uses 20 interleaved span processors organized in a 5x4 array, where the draw triangle[2] process is interleaved over 5 span processors which each handle every fifth scanline (in the GTX scanlines are oriented vertically). Each span processor is interleaved over 4 image engines which each handle every fourth pixel. In effect this means that in order to keep the renderers busy, triangles must be larger than a certain minimum size (5x4 pixels), smaller triangles will leave some of the processors idle. Also, all the SGI architectures (GTX [4], RealityEngine [5], InfiniteReality [158]) apply the interleaving scheme over the entire framebuffer, which makes rendering of large triangles very fast, but it also makes rendering of very small triangles very ineffi-

[2]The GTX architecture was really designed to process generic polygons, where each polygon is decomposed into trapezoids in the setup stage.

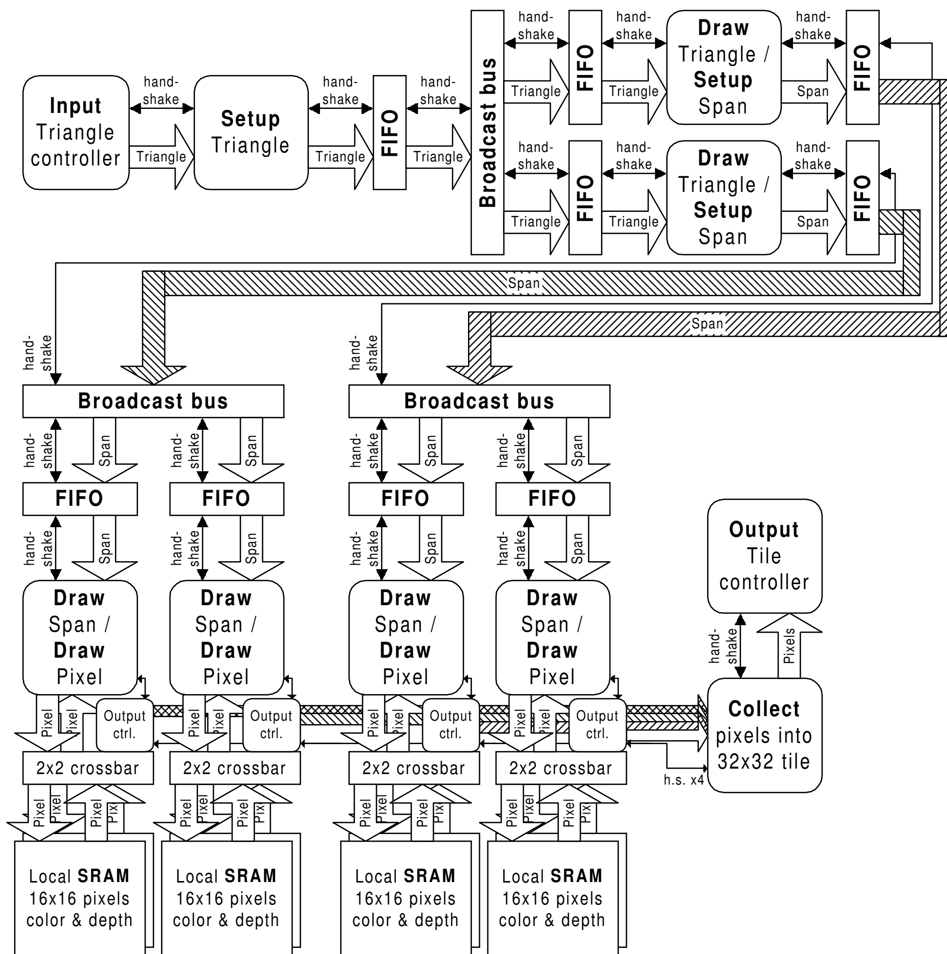

Figure 3.24: Architectural overview of the 2x2 pixel interleaved configuration of a parallelized tile rendering engine back-end.

cient. The InfiniteReality uses a framebuffer tiling pattern to make texture mapping more efficient.

Applying this interleaving scheme to the tile rendering engine back-end makes it possible to better balance the pipeline and improve the overall throughput. Since the tile engine keeps all memory on-chip the cost of interleaving the memory subsystem is very low. A viable configuration would be a 2x2 interleaved architecture where the same triangle is broadcast to two independent Draw Triangle stages, where one handles the even scanlines and the other handles the odd scanlines. After span setup, each of the stages broadcasts the same span to two independent Draw Span stages, where one handles the even pixels and the other handles the odd pixels. Each of the Draw Span processors require access to only one fourth of the pixels in a tile, allowing subdivision of the 32x32 pixel tile into four smaller 16x16 pixel tile buffers. This subdivision increases the effective bandwidth of the tile pixel memory four times, allowing four pixel span processors to work with no performance penalty. The bandwidth for the output controller is also effectively increased four times, allowing faster initialization and output of the tile pixel memory.

Figure 3.24 shows an overview of the 2x2 pixel interleaved tile buffer architecture. Note the use of FIFO buffering before *and* after broadcasting, which will allow each stage after the broadcast to work more independently. This allows better load balancing. Adding a FIFO buffer before broadcast allows cheaper buffering for cases where the FIFO load balancing is exhausted, e.g. when a FIFO is full because one processor is delaying the processing. For this reason the FIFO buffer before the broadcast should be deeper than the buffers after the broadcast. In effect, the buffers before the broadcast help reduce load imbalance between pipeline stages, while the buffers after the broadcast help reduce load imbalance between parallel processors within a pipeline stage. This load balancing method assumes an even workload distribution.

All control signals such as a signal to identify when rendering of all triangles in the tile has finished, are passed through the pipeline and FIFOs as extra bits in the triangle and span messages.

3.6.6 Anti-aliasing for the tile renderer

A technique rapidly gaining support in modern graphics hardware is *anti-aliasing*. While it is not currently implemented in Hybris, the tile rendering engine is well suited for implementation of supersampling anti-aliasing. Since the pixels in the tile buffer can be accessed quickly, it is possible to implement e.g. 2x2 pixel supersampling using the 32x32 pixel tile buffer. When all contributing triangles have been rendered to the tile buffer, it is filtered using a 2x2 pixel box filter. The result is

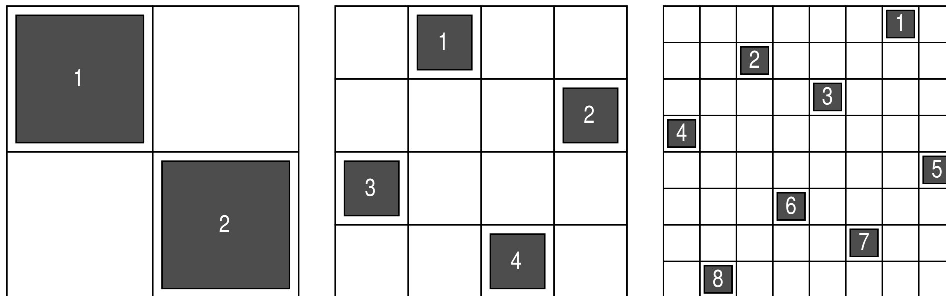

Figure 3.25: Sparse supersampling sub-pixel sample positions within a pixel. Left: 2 samples in a 2x2 grid. Middle: 4 samples in a 4x4 grid, Right: 8 samples in a 8x8 grid.

a filtered 16x16 pixel tile buffer which can be stored in the global framebuffer. This is a simple type of supersampling anti-aliasing popularly known as 4X OGSS [11] (Ordered Grid Super-Sampling). Using a tile rendering engine for implementation of supersampling anti-aliasing is very efficient in terms of bandwidth, compared to a traditional global framebuffer renderer which must store pixel and depth values in a supersampled global framebuffer, requiring more memory and memory bandwidth.

The 2x2 supersampling anti-aliasing technique fits perfectly with the previously described interleaved 2x2 pixel-parallel tile rendering engine. Rendering would be to a supersampled tile buffer with four pixel processors each generating one of the four sub-pixels. Finally a box filter would reduce the tile buffer to an anti-aliased tile by averaging the four pixels using equal weights ($1/4$).

Numerous other approaches for implementing anti-aliasing are currently emerging in recent graphics hardware. Since full supersampling with nxn sub-pixels requires n^2 samples to be processed, there might be better ways to use the high number of samples. Stochastic supersampling is a technique normally employed in ray-tracing which uses several sample points randomly placed within the area of one screen-space pixel, with a different random placement for every pixel. The benefit of this is that *noise* is added to mask the aliasing noise present in an ordered rectangular grid. While stochastic supersampling provides high image quality, it is difficult to implement efficiently.

Related to stochastic supersampling methods are *sparse* supersampling methods. As mentioned in [110] the SGI InfiniteReality [158, 136] implements the sparse supersampling method in hardware. Sparse supersampling using n selected samples placed within a nxn sub-pixel grid may look almost as good as true ordered grid supersampling using all n^2 sample points. The reason for this lies par-

tially in the way the computer screen and human eye/brain interprets the pixels. Without antialiasing, nearly vertical and horizontal edges will be affected the most by aliasing, while diagonal lines are not perceived as badly aliased. Antialiasing with ordered grid supersampling helps reduce this problem but treats all angles equally, i.e. nearly horizontal or vertical edges only benefit from 8 rows or columns in an 8x8 sub-sample array. With sparse supersampling using one sample per row and column we can achieve approximately the same result as full supersampling for those nearly horizontal or vertical edges if the subpixel samples avoid being axis aligned. This makes it possible to get n intensity steps from n sample points distributed on a nxn sub-pixel grid, while rendering nearly vertical or horizontal edges. The goal here is to approximate stochastic supersampling, by making the sub-sample distribution as "random" as possible, maintaining one sample per row and column while making sure that the samples are evenly distributed. This is important to avoid flashing of sub-pixel sized moving objects. Figure 3.25 shows some examples of sample patterns for sparse supersampling using one sample per row and column.

Returning to the 2x2 supersampling antialiasing method described earlier, we may extend it to a sparse 4x4 supersampling method, still using only 4 sub-pixel samples. The results should be an antialiased image quality closer to full 4x4 sample supersampling than the original 2x2 samples. However, extending the 2x2 sample ordered supersampling method to 4x4 sparse supersampling is not quite straightforward. One applicable method is multi-pass rendering as in [44] which uses a stochastic supersampling method by accumulating images rendered with jittered viewpoints. Since each pixel is offset by the same amount of jitter, the result is effectively the same as sparse supersampling. Note that such a multi-pass algorithm may alternatively be used to implement temporal anti-aliasing (motion blur) and field-of-view (out of focus blur) by using different camera locations and orientations while rendering each accumulated image.

In the 3dfx Voodoo 5 a different approach is used to avoid multi-pass rendering by using parallelism in the "T-Buffer™" [220] framebuffers. The T-Buffers are two or four framebuffers which can be combined by averaging during video display in a specialized video RAMDAC, explaining the 'T' in the name. The jitter offset is the same as with the multi-pass algorithm except that the sub-pixel offset is applied on screen-space coordinates just before rasterization. This method allows single-pass rendering but requires a parallel architecture with four T-Buffers and four renderers to enable four sub-sample antialiasing.

Returning to a possible implementation in Hybris, the multi-pass and parallel jitter algorithms are not well suited. This is because the tile buffering causes problems with jitter offsets, since adding a sub-pixel jitter offset to a sample might cause it to move into a neighboring tile. Fixing this problem would require over-

lapping tiles. Multi-pass viewpoint jitter is also impractical because of the virtual buffer nature of the tile buffer, as multi-pass would require the tile to be read back from the global framebuffer in order to apply a second pass, which is also prone to precision round-off errors.

A solution suitable for the tiled Hybris architecture is to render the scene at the full 4x4 supersampling resolution, and then *selectively* rasterize only the subsamples at the sparse supersampling locations. Figure 3.25 (Middle) shows which samples to select in this case. The 2x2 interleaved pixel parallel architecture in figure 3.24 would however not handle this case, as it is designed to interpolate across two scanlines, suitable for full 2x2 supersampling or just speeding up the standard non-antialiased rendering process. In order to handle 4x4 sparse supersampling the architecture must handle interpolation across four sub-pixel scanlines with variable x-axis span interpolation offsets for each sub-scanline to select the sparse sample positions. Four instances of the Draw Triangle processor would be needed rather than two. Figure 3.26 shows an architecture capable of performing 4x4 sparse supersampling, as four sub-pixel scanlines may be processed at once. This architecture is more general than figure 3.24, and can also be used to implement the previously described interleaved 2x2 pixel parallel renderer.

Other techniques similar to sparse supersampling are the four-sample RGSS (Rotated Grid Super Sampling) method used in the 3dfx Voodoo 5 [11], as well as the hybrid "Quincunx™" two-sample supersampling antialiasing / five-sample blur filter method used in the new Nvidia GeForce 3 accelerator. On the intermediate level between sparse and full supersampling is staggered grid supersampling [235] which samples half as many sub-pixels as full supersampling using a checkerboard pattern. Stochastic supersampling using several sample points randomly placed within the area of one screen-space pixel was possibly used in the GigaPixel architecture [187], although reading between the lines it was probably also using sparse supersampling.

Among other popular antialiasing algorithms for graphics hardware are the A-buffer [29] algorithms which use pixel coverage calculation to perform antialiasing with a better precision and without supersampling. Examples of the A-buffer algorithm used for tile rendering are found in [235, 10]. Other coverage-based methods include [134, 79] as well as the SPARP [124, 125] and Z^3 [110] which are efficient extensions of the subpixel bitmask based A-buffer methods described in [199, 200]. Unfortunately all these architectures have several problems with handling transparency and sub-pixel depth buffering, complicating their design and use. Supersampling handles these issues correctly and simply.

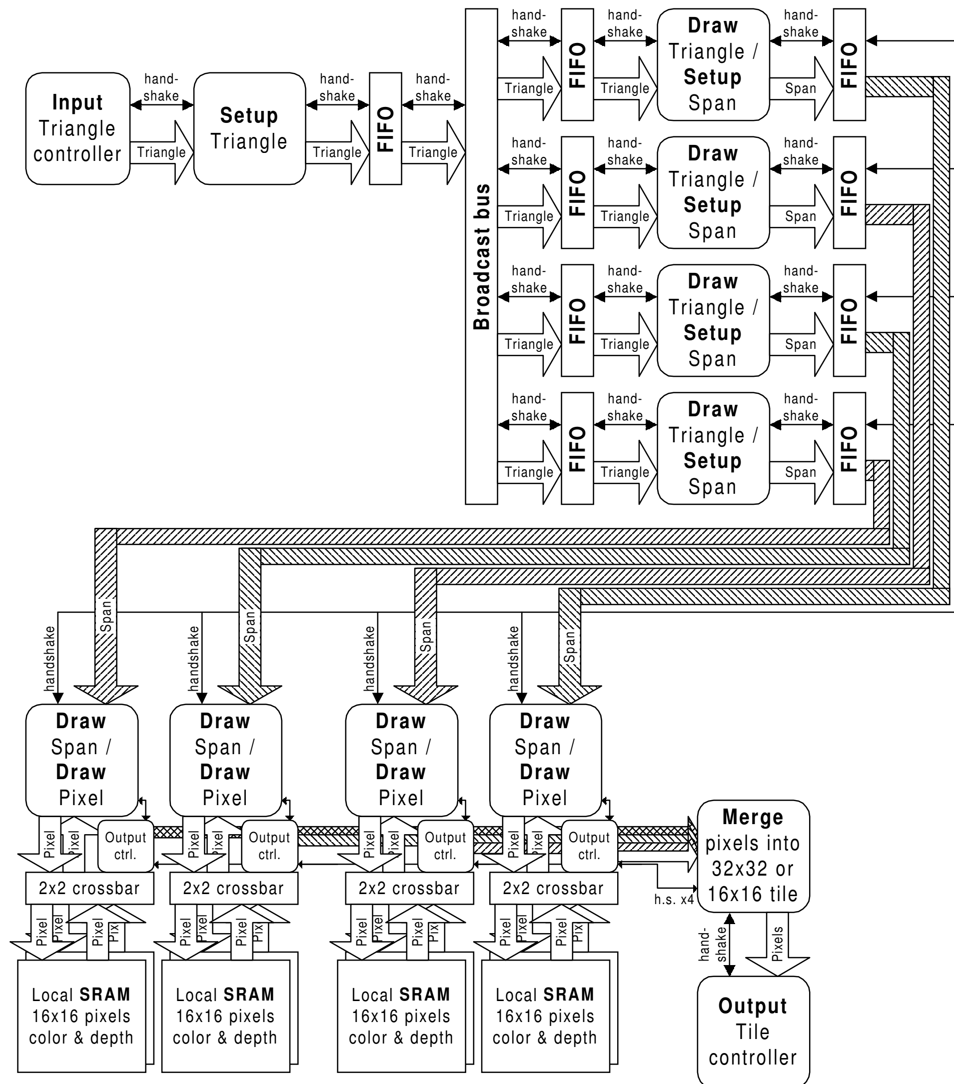

Figure 3.26: Architectural overview of a scanline interleaved pixel parallel configuration of the tile rendering engine back-end, suitable for 4x4 sparse supersampling using four sub-samples within a 4x4 sub-pixel grid.

3.7 Texture mapping

One important aspect of computer graphics currently missing in the Hybris graphics architecture is *Texture Mapping*. Texture mapping has not been implemented yet because it would complicate the design process and limit the possibilities to experiment with various details of the architecture. Another reason is that the author believes that texture mapping in Hybris can be replaced with per-vertex surface parameters, such as color. This is feasible since the triangle meshes used in computer graphics show a tendency to grow very dense, making it difficult to distinguish between a rendered image of a textured object and a per-vertex shaded object. Recent research in point rendering such as *Surfels* [181] clearly shows that this is a promising alternative to texture mapping. This colored point rendering method achieves image quality similar to rendering a dense triangle mesh of triangles each only a few pixels in size.

The problem of mapping texture map images onto colored vertices in a triangle mesh is related to the very similar problem of finding an optimal triangulation of terrain elevation maps. Such terrain map simplification algorithms are described in [91, 231, 173], and a nice example of texture to triangle mesh mapping is found in [22].

Colored vertices can also be viewed as an effective form of texture compression, given that a dense triangle mesh would be used anyway. Texture lighting, which usually requires modulation of the texture values in a pixel shader, is trivially implemented with colored vertices. However for scenes with large flat polygons texture mapped with detailed bitmapped pictures (e.g. many current computer games) the colored vertex technique is somewhat impractical as the large polygons would have to be subdivided into many small triangles in order to map the image onto colored vertices. In this case traditional texture mapping is very useful.

An implementation of texture mapping involves storing a texture map identifier as well as (u,v) texture map coordinates per vertex, just like colored vertices which stores the color as (r,g,b,α). The homogeneous texture coordinates are interpolated across the triangle just like colour and depth parameters, except that per-pixel perspective correction must be applied before looking up the colour value in the texture map. This inverse mapping requires two divisions (or one reciprocal and two multiplications) per pixel [82, 229, 236], which requires specialized division hardware to perform quickly. Alternatively perspective correction may be omitted if the triangles are very small. Related techniques for large triangles are [9] and [14].

Traditional texture mapping is also possible to implement in the Hybris architecture, e.g. by having the draw-span process query a texture map processor to get a color for the current pixel. Since we render a small tile at a time, we can use

texture caching techniques similar to those used in [100]. Note that tiling should be used for both texture map representation as well as rendering to improve texture caching, as concluded in [80]. Pseudo-tiled rasterization order [139] is used by some global framebuffer architectures such as the Neon [140] to improve texture caching. Note that texture caching is important to improve texture mapping performance as each texture mapped pixel value is usually calculated by *filtering* of several adjacent texels to get an aliasing free texture value for the pixel. A hierarchical pyramid filtering technique known as MIP-mapping is usually employed to optimize texture map filtering [82, 63, 96, 219].

By leveraging the tile-based architecture we can alternatively apply *deferred* texture mapping on a per-tile level. Deferred texture mapping is described in [44] and in the PixelFlow [155, 57] architecture. Deferred texture mapping reduces the calculations required for texture mapping by factoring them out of the inner triangle rasterization loop. Instead the renderer stores interpolated texture map coordinates at each pixel. After rasterization of *all* triangles have completed, the actual texture map lookup can begin. Because of the deferring of texture map lookups until the end of the rendering process, we avoid looking up texture map pixels for those pixels which have been overwritten while rendering images with overlapping features. This results in a potential performance improvement for scenes containing many overlapping texture-mapped surfaces, under the assumption that they are not transparent. Because of this transparency problem, deferred texture mapping suffers from the same problems sort-last image composition architectures are facing. To solve the transparency problem correctly for sort-last and deferred texture mapping architectures requires an efficient pixel fragment sorting architecture, a topic for future research.

Recent commercial implementations of deferred texture mapping and shading are found in the Giga3D [187] and PowerVR [188] graphics architectures for the PC.

Procedural textures is an alternative promising technique for applying real-time synthesized textures to objects, as storage for texture maps is completely avoided. Perlin [177] describes some useful procedures for generating colour patterns for simulating e.g. marble. A possible implementation of these techniques for hardware is described in [65]. Current graphics processors are beginning to integrate techniques similar to texture synthesis, an example is a generalization of multi-texturing called texture shaders [138] which is used e.g. in the pixel-shaders of the recent Nvidia GeForce 3 graphics processor. The trend for the future is to integrate better and more complex per-pixel shading techniques into future graphics hardware as suggested by [205]. Such techniques were previously only usable for off-line batch rendering of computer animations.

3.8 Chapter summary

In this chapter we have presented an in-depth view of the concepts involved in the design of the Hybris graphics architecture, at an abstraction level slightly above the possible implementations. The potentially available parallelism of the architecture has been described independently of any actual implementation.

The graphics architecture has taken form as an object-parallel and image-parallel architecture utilizing partitioned triangle meshes in the front-end graphics pipeline and tile-based rendering in the back-end graphics pipeline.

Further, several viable extensions for the architecture have been discussed. These extensions include antialiasing and texture mapping. Additionally some possible sub-tile interleaved pixel parallel architectures for the tile rendering back-end have been presented.

Chapter 4

Codesign for Hardware and Software Implementations

In this chapter we will explore some of the possible implementations of the scalable Hybris graphics architecture. Using codesign methods based on virtual prototyping we are able to examine several possible configurations of the architecture and then semi-automatically synthesize several possible hardware and software implementations of the architecture, using optimizing C compilers and VHDL logic synthesizers. The hardware/software partitioning is done manually before using the C compiler and VHDL synthesizer.

Two software implementations are discussed, one is a single-threaded program for a standard single processor Windows PC, and the other is a multiprocessor implementation for a Windows 2000 dual Pentium III SMP workstation demonstrating scalability. Two hardware implementations are examined. One is for a standard-cell based ASIC implementation using an STMicroelectronics 0.25 μ CMOS process. The other is an FPGA implementation for a Xilinx Virtex XCV1000 FPGA mounted on a Celoxica RC1000 reconfigurable computing platform. Furthermore some extended implementations leveraging the scalability and advanced features of the Hybris scalable graphics architecture will be discussed.

4.1 Design methodology

During the design of the Hybris architecture and its implementations, various design methods were used. During the design process a constantly mutating mental image of the processes in the architecture is maintained. This design is then manually captured into a software representation using the C or C++ language to model the processes as a sequence of loops. Since the imagined processes of the archi-

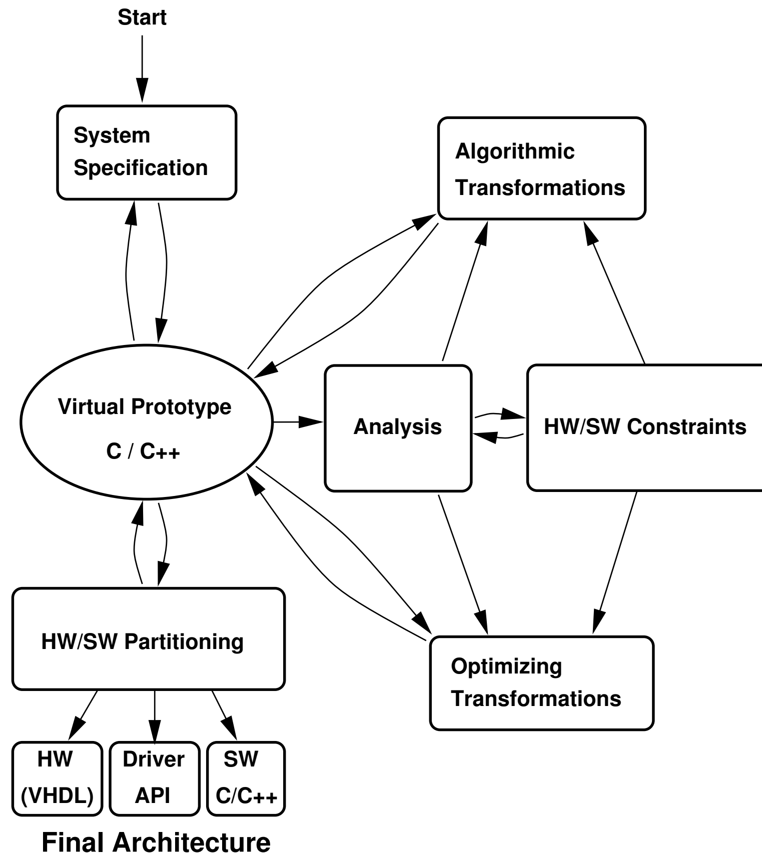

Figure 4.1: Codesign work-flow using the Virtual Prototyping design methods. Design space exploration is done using a C implementation of the architecture. The C representation is then used for HW/SW partitioning.

tecture are concurrently running tasks, capturing them in the C language enforces a serial ordering of the processes. Using this C representation, we can experiment with various loop transformations to explore different design options and different types of data buffering between loops. Finally we can use the C representation as a basis to implement optimized software and hardware implementations. This approach allows functional verification of the design ideas. Figure 4.1 illustrates this *Virtual Prototyping* design method, which was used to design and test the Hybris architecture. The paper [88] covers some aspects of virtual prototyping. Some of the virtual prototyping design techniques are loop transformations to improve memory usage efficiency, such as the loop fusion and strip mining techniques [117, 143, 228, 145, 194]. The work presented in this thesis is the result of con-

tinued improvement of the Hybris architecture using this simple C based virtual prototyping design method.

In [56] some related aspects of applying codesign for graphics hardware design are examined, using a slightly different approach to virtual prototyping based codesign. They use a “Data Flow Modelling Framework” (DFMF) and an “Algorithm Prototyping Environment” (APE) to enable the same kind of design space exploration as used for development of Hybris. Their DFMF is a C++ class library for representation of pipelines using stacks for buffering data between loops corresponding to processes in a pipeline. APE is a C++ implementation, using the DFMF classes, which is an actual “virtual” implementation of the graphics processor under development. The main motivation for this approach is the faster design and simulation cycles possible in C relative to VHDL, and the possibility to model the design at a higher level of abstraction. They also refer to one of our early papers [132], which describes the use of a C/VHDL co-simulation environment to model an architecture for a generic graphics co-processor.

4.1.1 Codesign using C language for architectural design

To give an overview of how virtual prototyping can be used to represent and explore design options, we may think of a C program manually derived from a mental image of an architecture, reflecting a serial functional description of the architecture. Program flow starts at the beginning of a pipeline of concurrently running processes. Each process may be modeled using a loop, where each loop iteration represents a datapath. During program flow, a loop may be interpreted as data parallelism if a loop iteration does not depend on a result from the previous iteration, allowing us to process each iteration in parallel. Program flow passing from one loop to a following loop in the program may be interpreted as a pipeline, or functional parallelism, allowing separate parallel processes to execute each loop. These observations allow us to transform a program into an architecture based on pipelines of processors (i.e. pipelined processor farms [62]), which resembles what happened during the design of the Hybris architecture.

When the program reflects the desired architecture, we can partition it into hardware and software components to create a combined system. This entire design process is called codesign or co-synthesis depending on the level of design tool automation. Codesign and co-synthesis in general using the Lyngby Co-Synthesis system (LYCOS) is covered in [133]. Further details about codesign and co-synthesis can be found in [119] as well as [76]. An overview of various codesign methods and systems is available in [209].

The LYCOS system uses a “Y-chart” codesign methodology to transform an application into an actual implementation by mapping it onto different components

in a pre-determined target architecture. Alternatively an "inverted Y-chart" codesign methodology starts with an application which is then successively refined into an architecture with various subsystems to be implemented.

The virtual prototyping development method that has been used to experiment with the Hybris graphics architecture shares many similarities with the *inverted* Y-chart codesign methodology, at least while operating on a sufficiently high level of abstraction. This design process is similar to the one used with the COSMOS codesign tool [227].

At some point in the codesign process the architecture needs to be implemented and we must choose between a limited number of components such as CPUs (usually we are forced to use Intel PC class CPUs) as well as other hardware components (ASIC standard-cells, FPGA prototyping boards, IP blocks), communication channels (PCI-bus) and storage (SDRAM main memory). When the derived architecture is mapped onto these components, we *flip* the Y-chart to use the regular Y-chart codesign methodology where the application, now well suited for implementation on a known target architecture, is mapped onto that target architecture to form the final implementation.

The design process we just described and used during virtual prototyping codesign and implementation of the Hybris graphics architecture, is very similar to a codesign methodology for embedded hardware-software systems which is described by Gajski in [66].

From an annotated C specification a more verbose specification of a system can be derived, allowing manual exploration of how the system can be partitioned into concurrently running processes, and how interfaces between the processes are implemented. Repeating this as a hierarchical design process we can decide how each of the processes are mapped to a machine description, either by subdividing until we arrive at a level suitable for hardware synthesis, or by stopping at a microprocessor software description of the process.

The virtual prototyping design process also allows design space exploration of interfaces in the design. The loop fusion and strip mining loop transformations applied to the design changes the structure of the processes. The interfaces appearing in the architecture reflect how the process are structured and mapped to various components of the possible target architectures. This allows us to partition the system to minimize communication bandwidth by transforming some of the bandwidth requirements to internal localized communication and computation. Communication bandwidth is often a limiting factor in digital systems design [108], and must be dealt with by designing the high-level architecture to reflect the actual implementation options.

4.1.2 Using C language for hardware description

Using a high level C description is not directly suitable for hardware description. In our case the hardware design case studies done in [130, 71] for ASIC design and [207] for FPGA design, were carried out by manual translation of the C source code into VHDL. A similar but probably more systematic design method was used in the design of the AMD K6-2 3D-Now! microprocessor [206], which was completely specified, simulated and functionally verified using a C++ program representation, but it still had to be manually translated to an HDL specification for synthesis, which required cross-verification with the C++ specification. Such a C based design process was also used during the design of the popular S3 ViRGE VX PC graphics processor [13], which was successively refined from a high-level functional architecture down to RTL level C code, only then was the VHDL code manually created.

An even more interesting example is the design method used to create the Neon graphics accelerator [141], which was completely specified, simulated and functionally verified as a C program. The interesting part is that they followed specific design rules for the C program which allowed *automatic* translation of the RTL level C specification to a RTL level Verilog HDL description synthesizable by Synopsys. This was achieved by modifying a portable C compiler `lcc` to create `c2v` which generates Verilog code for synthesis instead of machine code for a microprocessor. The C program was functionally partitioned to match the hardware architecture requirements, and an event driven simulator also written in C directly calls the C functions, also allowing the use of C debugging tools. The disadvantage of this approach is that an RTL level coding style has to be adopted for C programming.

The advantage of specifying an architecture in a programming language such as C is easier design space exploration, as well as faster simulation. Both the AMD K6-2 and the Neon design projects emphasized the importance of fast simulation greatly, as they were able to simulate their designs using multiple workstations in parallel (AMD used thousands of PCs for simulating the K6-2). Economically there is an added bonus since C simulation does not add the per-copy licensing fees of VHDL simulators. The Neon and the K6-2 are quite complex designs, containing 6.8 and 9.3 million transistors respectively.

Recently some more standardized C/C++ programming language based design methods are appearing. In [78] some aspects of using a programming language for hardware/software system design are covered, which forms the basis for recent efforts such as SystemC. SystemC [218] is a recent open system design standard for using C++ as an architecture description language, i.e. a higher level of abstraction than a typical HDL language such as VHDL. A limitation of C/C++ is that

it does not directly support features essential for low-level hardware specification such as; parallelism, reactivity (automatic response to stimuli), variable bit-length data types, multi-valued logic, communication channels (signals/wires) and time. For SystemC these shortcomings were handled by creating a C++ class library to provide the missing modeling elements without introducing proprietary extensions to the C++ language. As the SystemC class library includes a cycle-based simulation kernel, all levels of a complete system design can be simulated using an ANSI C++ compiler. Hardware synthesis from SystemC requires that the hardware parts of the system are described using a synthesizable subset of C++ similar to VHDL: i.e. no object-oriented features, no pointers, no recursion, no goto, no malloc/free, no embedded assembly code, etc. Unfortunately hardware synthesis tools for SystemC are not currently available, although manual translation to VHDL is possible.

The C language based design process applied during the virtual prototyping development of the Hybris graphics architecture is quite similar to the SystemC-based "refinement for implementation" design flow. Such a design flow, where the system architecture description is translated from higher abstraction levels to lower levels by always using SystemC representations is described in [59]. The main advantage of using a homogeneous system design language all the way down to the final optimized software and RTL hardware descriptions is easier testing by co-simulation and co-verification. Doing the same with C and VHDL is slower and possibly error prone.

Possibly SystemC might replace VHDL in the future, however current efforts with SystemC seem to indicate that the RTL level coding style will be the primary focus for current hardware design.

High-level hardware-from-C synthesis systems are beginning to appear, which work from a C program specification expressed as loop *nests*, a higher level of abstraction than RTL. A good example is the PICO-N (Program-In-Chip-Out) synthesis tool [203], which automatically partitions the design into a systolic array of customized non-programmable VLIW processors.

An interesting C based HDL is "Handel-C" [24] which is currently being promoted by Celoxica (formerly Embedded Solutions). Handel-C is basically Occam using a C-style syntax, which can be translated to ANSI C compatible code for simulation by using the Handel-C preprocessor. A logic synthesis tool is also provided, replacing VHDL synthesis tools such as Synopsys. The language introduces low-level parallelism constructs to C, using `par{}` blocks to indicate blocks of C statements to be executed in parallel. Normal C statements are executed serially in the sequence listed in the program. Logic synthesis with Handel-C interprets each C statement as a piece of combinatorial logic to be executed in one clock cycle. A sequence of C statements are sequenced by an automatically generated state machine, executing one C statement per clock cycle. Parallel C statements defined

inside `par{}` blocks are executed in parallel in one clock cycle. It is the Handel-C programmer's responsibility to make sure that each C statement is simple enough to execute within a given clock period, determining the maximum clock frequency. Pipelining can be expressed by using temporary variables between lines. In summary Handel-C simplifies many aspects of digital system design by hiding many details present in an equivalent VHDL design. Compared to RTL level VHDL design, it is difficult to know how resource sharing between sequential C statements is done.

For the design process of the Hybris graphics architecture, the virtual prototype C software representation of the architecture allows rapid experimentation with architectural concepts and implementation options. Manual translation into VHDL requires cross-verification and slows down simulations when exploring design options, as noted in [71]. The recent FPGA implementation [207] does not suffer from slow simulation speed, although logic synthesis and place and route is quite time consuming. Instead, the FPGA architecture limits the possible design space for simulation to what may fit in the FPGA.

As a curiosity, based on the ideas behind the Neon's event-based C simulation, Hybris might one day be able to simulate a high-level model of *itself* by using the event passing mechanism of its VRML engine to between VRML nodes representing different processes in the architecture. Hybris' VRML implementation is described in the next chapter.

4.2 Standard physical interfaces

While one way to solve the bandwidth problems in computer graphics is to build a fully customized graphics workstation using high bandwidth buses and interfaces everywhere, but for the unfortunate fact that it would be very expensive. Existing commodity interfaces should be used everywhere possible. In this section we will look at the available options relevant for the PC platform.

4.2.1 AGP – A fast interface for graphics

The AGP (Accelerated Graphics Port) interface [239, 104] was originally designed to allow graphics accelerators to store texture maps in the PC's main memory, reducing the total memory cost as semiconductor memory was very expensive when AGP was first introduced in 1996/97. Physically AGP is an extension of the 3.3V 32-bit 66 MHz PCI bus specification [175], but simplified to allow point-to-point connection only. It adds a demultiplexed address bus, pipelined transfers and also a double (2X) and quad (4X) data rate interface. The interface is clocked at 66

MHz and the 2X and 4X modes allow effective peak data rates up to 533 MB/s (2X) and 1066 MB/s (4X), compared to 266 MB/s for the simple 66 MHz transfer mode (1X).

Electrically the 1X transfer mode is identical to 66 MHz PCI, while the 2X transfer mode uses transfer of data on both rising and falling clock edges of an additional phase shifted 66 MHz clock. The 4X mode requires the signaling voltage lowered to 1.5V as well as two additional phase shifted and clock doubled (133 MHz) clock signals, allowing transfer of data four times (on the falling edge of each of the two 4X clocks) within one 66 MHz bus clock period of the interface. Intel is currently working on the specification of an 8X mode for the AGP interface.

For all transfers initiated from the AGP host to the AGP card, standard PCI transactions are used. AGP allows the AGP card to access the host's main memory using pipelined transactions in two different modes, "DMA" and "execute". The "DMA" mode is intended for large transactions e.g. for downloading textures, while the "execute" mode is used for random access of smaller blocks in the host's main memory. These transfers can be pipelined in a split-transaction fashion, by allowing the AGP card to issue its next data transaction while waiting for the previous transfer to complete. The AGP system logic maintains a prioritized queue of these transaction requests, divided into high-priority and low-priority subqueues. To speed up the shorter "execute" random access transfer type, an extra 8-bit SBA (Side Band Addressing) bus is used to send data transfer requests to the host. Random access within a large main memory area is complicated by the host operating system's virtual memory system as a contiguous memory area is fragmented into 4kbyte virtual memory pages. To solve this problem, the AGP host must maintain a GART (Graphics Address Remapping Table) to translate virtual memory addresses from the AGP graphics accelerator into physical main memory addresses. The AGP card's device driver must manage memory in coordination with the GART.

Today the original purpose of AGP, direct use of host memory for texture mapping in 3D graphics, has been defeated as memory costs have dropped rapidly, allowing graphics accelerators to use faster local memory directly on the graphics board. Configurations with 64 MB of 128 bit wide 230 MHz DDR SDRAM are common, allowing a far higher texture bandwidth than AGP. Instead the purpose of AGP is today being retargeted to allow faster texture map download, and more importantly faster communication of low level graphics primitives such as triangles. As the geometric detail level of 3D graphics continues to increase, more bandwidth will be needed for geometry relative to texture maps.

A problem with the AGP *port* philosophy is that systems based on AGP are difficult to scale incrementally, as only one AGP interface is available in a PC, making a dual 3D graphics card scaled configuration awkward. (Although one prototype of a twin AGP slot PC mainboard was reportedly spotted at a tradeshow

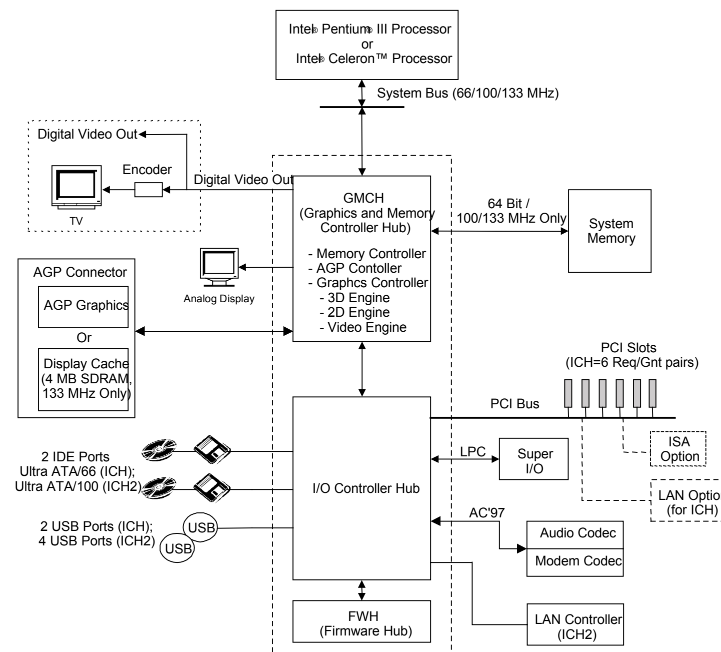

Figure 4.2: System architecture of a modern standard PC with an AGP interface and PCI bus, based on the Intel 815E chip set. (From [107].)

in 2000).

Figure 4.2 shows an example of a modern standard PC with an AGP interface and PCI bus, based on the Intel 815E PC chip set [107]. Note how the AGP interface is isolated from the PCI-bus. The 815E chip set even integrates a low-performance 3D graphics processor allowing low-cost PCs without an AGP graphics accelerator to be produced. Interestingly this approach seems to gain popularity, as recently (June 2001) Nvidia announced the *nForce* PC mainboard chip set featuring an integrated GeForce 2 MX 3D graphics processor as well as a crossbar memory controller allowing the PC to have two independent SDRAM main memory banks. While the integrated graphics processor approach eliminates the AGP bottleneck, the communication bottleneck is now moved to the CPU and main memory interface, as the graphics processor competes with the CPU for memory bandwidth. The crossbar memory architecture, which was probably inspired by the SGI O^2 workstation [118], helps to improve bandwidth.

AGP Pro

AGP Pro [105] is a simple extension of the standard AGP specification, increasing the mechanical, thermal and electrical limits. The form-factor of AGP Pro allows larger circuit boards to be used. The AGP Pro connector is physically extended to add additional power lines allowing a maximum power consumption of 110W by the AGP Pro board. In comparison AGP specifies a maximum current of 6A for the 3.3V supply, i.e. 20W.

AGP Pro is intended to make designs such as the 3Dlabs Wildcat II 5110 dual pipeline graphics accelerator possible [1].

4.2.2 PCI

PCI (Peripheral Component Interconnect) [175] is an older but more widely used bus interface. There is a high availability of design tools and prototyping boards for PCI (unlike AGP), but commonly only for the lowest-speed 33 MHz 32-bit PCI specification which has a limited peak transfer rate of 133 MB/s. The faster versions of the PCI bus feature a 66 MHz and/or 64-bit interface, but are not widely used mainly because of the availability of AGP.

The PCI bus can operate in to basic data transfer modes, the normal multiplexed address/data mode or the burst mode. When multiplexed address/data is used, the maximum bandwidth is effectively halved to 66 MB/s. Burst mode allows rapid transfer of consecutive data allowing transfers to approach the theoretical maximum bandwidth of 133 MB/s if the burst mode transfer transaction is large enough.

A convenient advantage of PCI is that prototyping boards for the PCI bus are available, such as the RC-1000PP PCI board featuring a PLX PCI 9080 PCI interface and a Xilinx Virtex XCV1000 FPGA. Unfortunately such a prototyping system is limited by the maximum speed of the PCI bus. The available driver for accessing the RC-1000PP through the PCI bus limits the bandwidth further. Unfortunately these prototyping boards are not available for the AGP port.

PCI-X

Compaq is currently promoting an extension of the PCI bus named PCI-X [34], which allows use of higher clock frequencies as well as an improved protocol specification allowing more efficient bus transfers of data.

PCI-X allows use of 33, 66, 100 or 133 MHz clocks on the bus, however the 66 MHz mode limits the bus to a maximum of four devices, the 100 MHz mode limits the bus to a maximum of two devices and the 133 MHz mode allows only one device i.e. a point-to-point configuration. From a card subsystem design point

of view PCI-X is simpler than PCI and AGP because Compaq makes an IP library available free of charge, which includes ready to use HDL designs for a PCI-X interface. However PCI-X based PCs are not yet widely available.

The main industry motivation for adopting PCI-X is to allow support of faster interfaces for Gigabit Ethernet and Ultra3 SCSI, e.g. a 4-port Gigabit Ethernet interface would fully saturate the 64-bit 66 MHz PCI bus, while a 64-bit 133 MHz PCI-X bus handles the load better.

4.3 Implementing the Hybris graphics architecture

The Hybris graphics architecture described in the previous chapter can be implemented in various ways. We need to decide what the target hardware platform is, and map the architecture onto it. In order to achieve good performance the platform must match the requirements of the architecture. In the previous chapter the architecture was described at a high enough level of abstraction so that the processes in the architecture can be mapped to many different platforms, as software on microprocessors or as dedicated processing units in an ASIC or FPGA. This platform mapping task can be considered as the last step in a hardware/software codesign design flow.

For practical and economical reasons the target platform is limited to a PC and possibly some add-on hardware on the PCI-bus. In the following we discuss benefits and shortcomings of the several possible codesign based implementations of the Hybris graphics architecture.

4.3.1 Single CPU software implementation

The simplest implementation target of the Hybris graphics architecture is a serial software implementation running on a single CPU. This is also the main implementation testbed for many of the algorithmic and architectural concepts explored during the virtual prototyping and design space exploration of the graphics architecture.

Although one method for implementation of the software version is to simply compile the virtual prototype representation of the architecture, we can also rely on platform dependent features to further optimize the implementation. This is done by mapping some of the loops to platform specific features such as the size of cache lines, type of cache line set associativity, amount of L1 and L2 cache as well as the amount and bandwidth of available main memory (e.g. SDRAM [147]). Other important features to consider are micro-architectural features of the CPU such as the performance properties of the floating point and integer execution units, as well

as how the CPU's instruction execution pipeline is affected by loops and branches. The interactions between CPU, caches and main memory lead to implementations that apply memory alignment, data blocking and main memory paging to improve dataflow. A valuable resource describing optimization issues related to the Intel Pentium III architecture is [106]. In the previous chapter many of these aspects were discussed during definition of the graphics architecture, allowing a relatively efficient implementation of the architecture for the single CPU PC environment. These system observations are also applicable for hardware designs, such as the ASIC and FPGA implementations discussed later.

For implementation on a Pentium III based PC, the data structures in the Hybris architecture are aligned and sized to fit in one or two 32-byte cache lines. A vertex fits in one cache line and a triangle node fits in two cache lines. This provides an efficient memory interface as the CPU always reads a cache line from system memory beginning on a 32-byte boundary. (A 32-byte aligned cache line begins at an address with its 5 least-significant bits zero.) A cache line can be filled from memory with a 4-transfer burst transaction. The caches do not support partially-filled cache lines, so caching even a single word requires caching an entire line. See the Intel architecture system programming guide [103] for more information. The Pentium III integrates two 16 kbyte L1 instruction and data caches and a 128–512 kbyte unified L2 cache, all caches are 4-way set associative with a 32-byte cache line size. For an overview of caches in general, see [174].

The loop fusion and strip mining data locality optimization techniques applied throughout the codesign of the Hybris graphics architecture ensures good cache utilization. E.g. the 32x32 pixel virtual local framebuffer tile uses 5 kbytes to store 8 bit color and 32 bit depth per pixel, which fits nicely into the L1 data cache, with plenty of space left for processing the 4 kbyte triangle heap buffers and other temporary data as well as space left for an extension to 24 bit color per pixel. Similarly the front-end uses an 8 kbyte temporary transformed vertex buffer which also fits in the L1 data cache. One problem in the bucket sorting stage might be the 4 kbyte memory stride between buffers in the bucket sorted triangle heap which may cause the memory start address of each buffer to map into the same cache lines. But because the caches are 4-way set associative they can manage up to four buffers mapped to the same set of cache lines at once. Because the objects are partitioned we typically only write triangles into a set of four neighboring tile bucket buffers, which matches the way the cache is managed.

In addition it can be an advantage to use inline assembly code to utilize e.g. Intel's MMX, SSE and SSE-2 SIMD vector processing extensions for the Pentium III and IV CPUs [106]. Similar extensions for the AMD CPUs are 3D-Now! and 3D-Now! Pro [206]. Other examples include the PowerPC AltiVec extensions and the UltraSparc VIS instructions. Finally special purpose processors such

as the Samsung MSP [166] implement similar vector datapaths. Unfortunately it would greatly complicate the virtual prototyping design process if these extensions were included in the virtual prototype, as they are very platform specific and not portable. To use these extensions in practice requires manual translation of the C specification to include these instructions as assembly code, e.g. by using compiler-specific intrinsic functions which cause the C compiler to emit SIMD instructions. Recently the Microsoft Visual C++ compiler has implemented preliminary support for such intrinsics, but is still in the beta testing stage. Intel's C compiler additionally supports *automatic* vectorization of the code to SIMD instructions, unfortunately the resulting code is often slower, e.g. the Hybris renderer was slowed down by about 10%. Experiments with optimizations of the Hybris software implementation using different C compilers revealed that the Microsoft Visual C++ compiler currently generates the fastest executing machine code, unfortunately the "processor-pack" upgrade patch for preliminary support of SIMD instructions breaks something in the compiler's support for C++ templates.

The Pentium III's SSE extensions additionally provide enhanced data streaming cache management techniques such as instructions for prefetch (load a cache line before it is actually needed) and non-temporal stores (store final data in main memory without also placing it in the cache). These enhancements may be useful to further optimize buffer management in Hybris for the partitioned object database, the bucket sorted triangle heap and the global framebuffer. Unfortunately an implementation using these techniques also requires manual assembly coding, leaving this as a topic for future experimentation.

The single threaded software implementation of the Hybris graphics architecture performs quite nicely on the test PC with a Pentium III 500 MHz CPU, reaching rendering performance levels up to 2,7 million triangles/s in software only. When rendering complex 3D models, this software renderer is in many cases able to out-perform a hardware graphics processor such as the Nvidia GeForce 2 GTS. Some performance benchmarks are listed at the end of this chapter.

The current software implementation is targeted for a PC running the Windows 2000 operating system, where an operating system specific interactive user interface is implemented using windowed output of the final rendered image with user feedback from mouse and keyboard input devices for manipulating the view direction, etc. Additionally the software has been encapsulated as a Java user interface component, allowing a Java application to easily integrate the Hybris software renderer. It should also be mentioned that an earlier single-threaded software implementation of Hybris has been successfully compiled with the Gnu C++ compiler `gcc` for the Linux/X-Windows operating system on a PC platform, demonstrating the portability of the architecture.

4.3.2 Multiple CPU parallel software implementation

The currently fastest working implementation of Hybris is a parallel software implementation. The parallel implementation is targeted specifically for a dual Pentium III 500 MHz PC running Windows 2000. Parallelism is achieved by using a process with two Win32 threads running in the SMP (Symmetric Multi-Processing) computing environment provided by the platform. The Hybris architecture was mapped to this programming model by utilizing the available data parallelism in the architecture, by running the graphics pipeline in both threads. Each thread runs on its own CPU and processes its own data. The object partitioned front-end pipeline was mapped onto two threads by mapping the first half of the object partitions to the first thread and the second half of the partitions to the second thread. When both threads finish processing for the front-end pipeline, a barrier synchronization point manages the threads, switching them to start working on the tile partitioned back-end pipeline. When working on the back-end pipeline, each thread is assigned a set of tiles to render. The first thread renders odd numbered tiles, while the second thread renders even numbered tiles. In effect the workload distribution forms a checkerboard pattern of tiles.

In the parallel renderer the bucket sorted triangle heap is not just used for binning the triangles into buckets for each tile. The parallel renderer also uses the bucket sorted triangle heap for workload redistribution. This is an implementation of sort-middle parallelism. Figure 4.3 shows the dataflow in the parallel renderer, using two triangle heaps. Note that the sort-middle redistribution of triangles requires that each tile rendering worker reads data from all triangle heaps. As long as all CPU's work exclusively on either the front-end or the back-end, it is not necessary to double buffer the bucket sorted triangle heaps. However if a pipeline of concurrently running front- and back-end worker processor "farms" are formed, the triangle heaps must be double buffered in order to allow both pipeline stages to work in parallel. Note that by using two triangle heaps we can improve performance in the dual CPU implementation, as it allows writes to the two separate caches to operate without invalidating cache lines in each other. A cache line is invalidated if one processor writes to a memory location cached in the other, because of the automatic cache snooping logic in a dual Pentium III system. When reading from the triangle heaps this is not a problem, as cache lines are not invalidated by reading. Further, as each tile renderer only reads data for its own set of tiles, cache performance is good even though each tile renderer must read from both triangle heaps.

In a hardware implementation we can further improve memory performance by applying smarter SDRAM memory bank management techniques unavailable in software. This is because we have no control over how the operating system's vir-

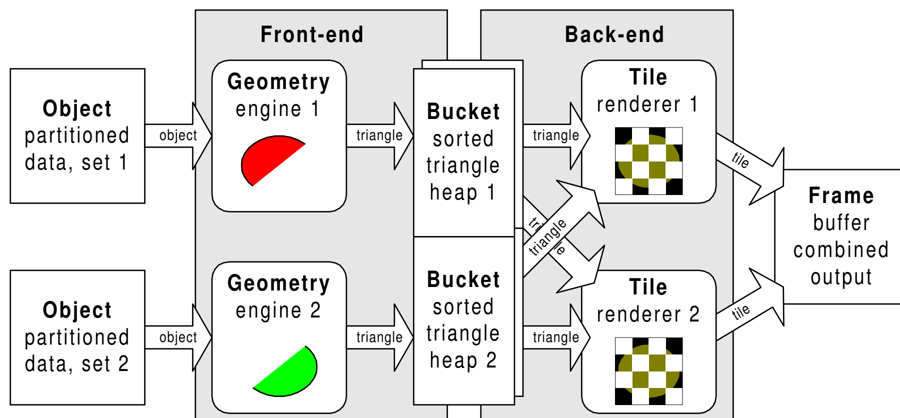

Figure 4.3: Dual CPU parallel implementation of the Hybris graphics architecture. Two sets of object partitions are processed independently by the workers in front-end pipeline and binned into tiles in two independent bucket sorted triangle heaps. In the next stage the two tile rendering workers in the back-end pipeline process tiles in parallel. Each tile renderer reads data from all triangle heaps.

tual memory management maps the 4 kbyte virtual memory pages to the physical SDRAM's four banks of 4 kbyte memory pages. For the software implementation the current approach of using a large array of 4 kbyte page aligned buffers is the best we can do to improve bank management, according to [106]. If bank management is available we can organize the triangle heap for the case shown in figure 4.3 by having geometry engine 1 write odd tile buckets to bank 0 and even tile buckets to bank 1, similarly for geometry engine 2 and bank 2 and 3. Tile renderer 1 then reads from bank 0 and 2, and tile renderer 2 from bank 1 and 3.

This parallel implementation of the Hybris architecture has proved to be very efficient, reaching a speedup close to two for a variety of scenes. This demonstrates some of the nice scalability properties achievable with implementations of the Hybris graphics architecture. Future implementations for SMP parallel processing platforms with more than two CPU's are also possible, provided that enough memory bandwidth is available for bucket sorting. As an interesting observation, this structure for the parallel renderer using a pipeline of groups of worker processors has recently been formalized as a general method for structured design for embedded parallel systems, known as *Pipelined Processor Farms* (PPF) in the new book [62]. In PPF terms, the front-end and the back-end of the parallel implementation are processor farms which together form a pipelined processor farm.

A multiprocessor platform usable for this type of pipelined multiprocessing is

the Imagine stream processor, for which a graphics renderer has been implemented in [172]. They conclude that such a parallel computing platform is very competitive to contemporary hardware graphics processors. The Imagine achieves its performance using the same design methods that have traditionally been exploited in special-purpose hardware, but without giving up programmability.

Shared memory multiprocessor architectures such as SMP are currently the best for a parallel implementation of Hybris, as the triangle heaps are used for sort-middle redistribution. Using a distributed memory parallel architecture would need a very efficient implementation of message passing, requiring efficient hardware support for good performance. The distributed memory approach is better suited for a hardware implementation where dedicated communication channels are available. Note that the programming model necessary for distributed memory parallel computing, which emphasizes data locality, is also very useful for shared memory parallel architectures, where the caches behave as distributed local memories. For a highly scaled implementation of Hybris, a shared memory multiprocessor should provide some form of multi-banked memory with an efficient communication network based on e.g. crossbar switches. This type of parallel computer architecture can be considered a hybrid of shared/distributed multiprocessing architectures.

An earlier parallel renderer implementation [85] of an early version of the Hybris architecture without object partitioning was not as successful, reaching a speed-up of only 1.6 in the best case. It was limited mainly by main memory bandwidth for the transformed vertex buffer and also attempted to run three different stages of the graphics pipeline concurrently (geometry, triangle setup and tile rendering), causing severe memory paging and poor cache utilization.

4.4 ASIC implementation

An implementation of the back-end tile rendering engine of Hybris for an ASIC was made in [71, 72]. The ASIC implementation was targeted for the STMicroelectronics HCMOS7 0.25 μ standard-cell based CMOS manufacturing process.

In order to implement the ASIC, the C source code of the reference software implementation of the Hybris architecture had to be translated to VHDL source code in a format suitable for logic synthesis. See [164] for an overview of VHDL for modeling of digital systems and [217] for a description of the synthesizable subset of VHDL implemented in Synopsys. In order to enable logic synthesis some strict coding guidelines must be followed. For logic synthesis using Synopsys Design Compiler this means that the VHDL code must be an RTL (Register Transfer Level) description of the digital circuit. This RTL description of the architecture was first derived by manually transforming the C source code into "RTL-friendly"

C code, by changing the way loop variables are used and updated. Each loop in the C program is transformed into two parts, one reflecting calculations and one reflecting loop variables. This transformation closely resembles the RTL coding style in VHDL where a loop can be expressed as two VHDL processes, one reflecting combinatorial logic (calculations) and one reflecting clocked register transfers (loop variables). Furthermore, nested loops in the C program are decomposed into individual loops. This design process is an example of the Virtual Prototyping design method discussed earlier, where a C program specification is transformed into another C program matching the implementation target architecture. From the RTL-friendly C code the RTL VHDL description is then derived by manual translation.

The tile rendering engine pipeline structure discussed in the previous chapter (see figure 3.20 page 79) was expressed in RTL VHDL code ready to be synthesized and implemented in the ASIC. FIFO buffering between the pipeline stages corresponding to the loop nesting was added to balance the workload, as the execution time for each stage is highly data dependent and can vary by many clock cycles. On-chip SRAM was used to implement both the FIFOs and the dual ported 32x32 pixel color & depth tile buffers. These SRAMs were implemented using Synopsys DesignWare SRAM's, although a real implementation should definitely use the full-custom dual ported SRAM macro-cell generators provided by STMicroelectronics, as the generic DesignWare SRAM's are slower and far less area efficient.

The FIFOs allow the implementation to balance the load across the pipeline in case one of the pipeline stages is stalling the pipeline, assuming that the average workload distribution provides a balanced workload for the pipeline. However the FIFOs use a lot of chip area without improving the maximum data throughput. A better way to utilize the chip area is to improve the load balance by creating parallel datapaths with interleaved pixel processing, as discussed in the previous chapter (see figure 3.24 page 83).

Using the RTL programming model for logic synthesis, each iteration through the calculations in the combinatorial logic process must be completed during one clock cycle. If a calculation is too complex to be performed within the desired clock frequency it can be subdivided either by using a state machine to control the dataflow and/or subdividing into more parallel processes or by pipelining the calculations.

Synopsys Design Compiler has a nice feature to aid in the design of pipelined datapaths called *register re-timing* or *register balancing* which allows the designer to add a pipeline of registers at the end of a datapath, and then tell the synthesis tool to distribute these registers across the datapath. The register re-timed pipelined datapath is functionally equivalent to the purely combinatorial datapath with an added

"delay" pipeline at the end, but it should now work correctly at a higher clock frequency. Synopsys FPGA Compiler II which is used for the FPGA implementation described later has a similar register balancing feature. In [71] the register balancing optimizer was used to create pipelined datapaths for the ASIC implementation of the tile rendering engine.

An SDRAM memory controller for the bucket sorted triangle heap was also designed for the ASIC implementation. The intention of this is to provide the prerequisites for a hardware implementation of the front-end graphics pipeline. I addition a hardware implementation of the triangle heap allows the design of a memory architecture better suited for the tile renderer. As mentioned in the description of the software implementations of Hybris, it is difficult to know how the 4 kbyte SDRAM pages are mapped to banks, because of the virtual memory manager. In hardware we have the opportunity to control this, as well as design a memory architecture suitable for implementation of a double-buffered triangle heap, allowing pipelined parallel operation of the front-end and back-end. The memory layout for the ASIC implementation's triangle heap is essentially the same as for the software implementation, i.e. triangle nodes in a linked list of 4 kbyte page aligned triangle buffers for each bucket. As the 2D bucket pointer hash table is small (40x32 pointers) and static in size, it can be located in a small on-chip buffer. The tile renderer back-end serially reads a triangle buffer from the triangle heap at a time, maximizing SDRAM performance as burst mode transfers can be used. Bandwidth problems may occur only when *writing* to the triangle heap from the front-end, as the writes require random access to memory which can cause excessive page swapping in the SDRAM memory. A write caching memory architecture is described in [72] which uses four FIFO buffers to serialize write accesses to the four banks of pages in an SDRAM [147]. This caching scheme requires consecutive writes to be evenly distributed over the four banks to keep the FIFO's balanced. In addition the current Hybris architecture has introduced object-partitioning in the front-end pipeline which further helps serialize the writes to the triangle heap, reducing the requirement to handle random writes. By organizing the tile buckets in a bank interleaved scheme so neighboring buckets are in different banks, we are able to handle bucket overlapping triangles and object-partitions efficiently.

An actual ASIC implementation of the design synthesized using Synopsys Design Compiler would need to be processed with the Cadence Silicon Ensemble ASIC design layout tool. The input to Cadence is a Verilog net-list produced with Synopsys. Cadence is then used to map the design to the ASIC standard-cell library provided by STMicroelectronics in order to perform layout with floorplanning, placement and routing. Finally the design layout tool is used to create the lithography masks required for the physical manufacturing process to produce the prototype ASIC.

ASIC Simulation

Manufacturing costs for a prototype run of an ASIC in small quantities is very high (e.g. the STMicroelectronics 0.25μ CMOS process costs about 700 euros per mm^2 for a few prototype chips, including university discount). Because of this, simulation of the ASIC design is necessary to get an idea of how the design works before actually manufacturing a chip.

Since the ASIC design was not fully completed only simulation estimates of the performance is available. From [71] simulated performance for the tile rendering back-end running at 27 MHz is approximately 16 frames/s for rendering an object with 1 million triangles, such as the Stanford Buddha. Since the ASIC implementation used floating point operations in some of the inner loops, that is the main limiting factor for the performance. As pointed out in [71] fixed-point arithmetic would be required to increase the speed of the design.

In the following section we investigate an FPGA design which implements the tile rendering back-end using fixed-point arithmetic.

4.5 FPGA implementation

Based on the work done for the ASIC implementation, an FPGA implementation [207] was made, targeted for a Xilinx Virtex XCV1000 FPGA [238], a modern FPGA which provides a design space equivalent to one million system gates. For an overview of FPGAs in general, see [26]. In order to fit the tile rendering engine onto the FPGA, several changes had to be made. One of the changes was to remove the FIFOs between the rendering pipeline stages in the ASIC design, as those large FIFOs are excessively expensive/impossible to implement in an FPGA. The area is much better spent for applying parallelism in the tile rendering engine together with some shorter FIFO buffering. Currently the FIFOs are reduced to a depth of one, i.e. replaced by simple registers. Another change to the design was to remove pipelining from the individual datapaths. While not absolutely necessary, as all the FPGA cells contain registers useful for pipelining, this made the FPGA design process much simpler and also allowed fixing some bugs from the ASIC implementation. Datapath pipelining may be re-introduced for the FPGA implementation in the future, as the Synopsys FPGA Compiler II supports the same register balancing optimization techniques used to implement pipelining for the ASIC. The synthesizable subset of VHDL implemented in the Synopsys FPGA Compiler II is described in [217].

One of the most important changes made was that the on-chip SRAM architecture for on-chip tile buffering was redesigned. The ASIC implementation used a generic Synopsys DesignWare SRAM block, which features an asynchronous

reset signal to clear the contents of the entire SRAM. While convenient, this reset behaviour adds complexity to an SRAM making it larger and slower. Further, to implement SRAM blocks in the FPGA we are forced to use the configurable blocks of dual ported SRAM present in the FPGA (known as BRAM), which do *not* feature asynchronous reset to clear the contents. VHDL components for the BRAMs can be generated by the Xilinx Core Generator's dual port SRAM generator. A properly pipelined tile buffer architecture using this more area efficient type of on-chip RAM was described in the previous chapter (see figure 3.20). Here double-buffering and cross-bar switching is used to allow multiple accesses to the dual ported 32x32 pixel tile buffers, providing an efficient tile buffer memory architecture which also allows enhanced rendering algorithms to read pixels, which is needed for alpha blending operations and other advanced rendering methods, such as those discussed in [48].

The tile depth z-buffer is now double buffered allowing it to be cleared and stored in a global depth-buffer for future use if required. In comparison, the ASIC implementation relied on asynchronous clear with a single buffered depth buffer. The software implementation of Hybris uses some of the z-buffer bits to implement "dirty" bits in order to limit the need for clearing the z-buffer every time. Similar techniques are described in [23] and reported to be used in the ATI Radeon graphics processor.

Additionally it is not possible to transfer the complete triangle heap to the FPGA, as it does not have enough on-board memory to accommodate the potentially huge triangle heap. E.g. 64 Mbytes are needed to store a triangle heap for one million rendered triangles. Because of this the PC host software for the FPGA implementation must send triangle buffers for one tile at a time to the FPGA board.

The FPGA implementation is targeted for a codesign prototyping platform based on an ordinary Pentium III PC with an FPGA prototyping board on the PCI-bus. This prototyping board is the Celoxica / Embedded Solutions Ltd. RC-1000PP PCI board featuring a PLX PCI 9080 PCI interface and a Xilinx Virtex XCV1000-6 FPGA as well as 8 Mbytes of asynchronous SRAM in four independently addressable 32 bits wide banks. Unfortunately such a prototyping system is limited by the maximum speed of the PCI bus. The available drivers for accessing the RC-1000PP through the PCI-bus limits the bandwidth further. The Xilinx Virtex FPGA architecture is described in [238], and the RC-1000PP manual [216] describes how the FPGA's pins are connected to the other resources on the prototyping board.

The design flow for the FPGA implementation is to start from a suitable reference C implementation and manually transform it into an RTL description of the design in VHDL suitable for FPGA implementation. In this particular study some of the VHDL code developed for the ASIC implementation could be reused. Synopsys FPGA Compiler II is then used to synthesize the VHDL description, targeted

for the Virtex FPGA architecture. Once synthesized an EDIF net-list is exported from Synopsys. This EDIF net-list is then imported into the Xilinx Alliance Design Manager FPGA layout tool to map the design to a particular FPGA using placement and routing to finally create an FPGA configuration bit-file. Finally, this bit-file is then used to configure the FPGA to form the designed digital system.

4.5.1 PCI bandwidth

The performance of the first FPGA implementation was severely limited by the available PCI bandwidth. The bandwidth problems were mainly caused by poor utilization of the limited communication models available in the driver for the RC-1000PP FPGA prototyping board.

The protocol used in [207] for transferring data over the PCI-bus performed a host-initiated PCI bus-master DMA transfer for each triangle node buffer in the bucket sorted triangle heap. While this type of transfer, once running, will transfer data at maximum speed over the PCI-bus, there is a relatively large overhead for initializing the data transfer. Since the blocks to be transferred are 4 kbytes or less in size, the accumulated overheads of many small transfers becomes very high. In [207] a plot of block size and measured transfer rate is presented, showing a relatively poor transfer rate of about 10 Mbytes/s for continuous transfer of 4 kbyte data blocks. This should be compared to a data transfer rate of over 100 Mbytes/s achievable for a single large block transfer. It should be noted that the actual transfer rate depends on the PCI implementation of the PC's motherboard chip-set: Observed maximum transfer rates for transfers from the PC to the PCI board varied between 50 Mbytes/s with the Intel 815E chip-set, 80 Mbytes/sec with the VIA 694 chip-set and 110 Mbytes/sec with the Intel 440BX chip-set.

According to the PLX PCI 9080 manual [185] the PCI interface on the FPGA board is also capable of performing scatter/gather or chaining mode PCI bus-master DMA transfers. Chaining mode DMA would be the ideal method for transferring data to the FPGA prototyping board, because it is able to match the data distribution in the triangle heap. The bucket sorted triangle heap contains a set of triangle node buffers for each bucket corresponding to a tile. These buffers are not necessarily placed in contiguous memory locations. However, by using chaining mode DMA it would be possible to instruct the PLX PCI 9080's DMA engine to transfer the relevant blocks. In practice this can be done by creating a linked list in either host (PC) or local (FPGA board) memory which is a list of pointers to variable sized memory blocks to be fetched. Once started, the PLX chip automatically executes the transfers by following the links in the list, avoiding the expensive overhead of starting the individual transfers from the host. Unfortunately the supplied RC-1000PP device driver API [25] for Windows only exposes a simplified "2D"

interface to this scatter/gather functionality, requiring that the memory blocks are equal in size and spaced equally in memory. Since no source code was supplied for the RC-1000PP API, adding the proper functionality would require starting from scratch with the implementation of a Windows device driver.

Another alternative data transfer method is to use memory mapped I/O. As the FPGA board's driver maps the on-board memory to a virtual address space in the host PC, the FPGA board memory can be accessed as normal memory, although at a lower speed. An implementation of the triangle data transfer based on memory mapped I/O improves the bandwidth to about 30 Mbytes/s. Although better than the original 10 Mbytes/s, it is still far from the maximum PCI bandwidth. The slow speed is caused by address/data multiplexing which halves the available bandwidth, as well as the use of individual PCI bus transfers for each data word.

The best transfer method to the FPGA board was found to be DMA transfers of large data blocks. However this requires re-organization of the 4 kbyte triangle heap memory buffers into larger 2 Mbyte memory blocks which as mentioned earlier can be transferred at the maximum transfer rate of the PCI bus, i.e. up to about 100 Mbytes/s. The re-organization causes a small overhead from memory-to-memory copying, but nevertheless this method is currently the best performing transfer method to the FPGA board, using the current drivers. For higher performance, the memory copy can be avoided by using chaining mode PCI bus-master DMA, but this requires an improved driver.

Once the data has arrived in one of the input buffer memory banks on the FPGA board itself, the tiles are rendered one tile at a time, using the buffer swapping input multiplexer described in [207] to allow PCI-bus data transfer and tile rendering to be overlapped in time. This buffering scheme is essentially an implementation of a very large input FIFO needed to maximize PCI bandwidth by using large transfer sizes. Figure 4.4 shows the input multiplexer needed to access data from one or the other bank, as well as the input triangle controller.

Note that if chaining mode DMA for transfer of smaller buffers is available, the on-chip BRAMs on the FPGA would be sufficient for input FIFO buffering, eliminating the need for external SRAM buffering. Incidentally the Virtex FPGA implementation currently has exactly 8 kbytes of unused on-chip BRAMs, just enough for two 4 kbyte input triangle buffers, usable for implementing an 8 kbyte input FIFO by utilizing the dual-ported BRAMs.

From this point on the FPGA implementation of the tile rendering engine from figure 3.20 renders the triangles in each tile, and then stores the rendered tiles in a full-frame framebuffer ready to be sent to the display. The next section discusses how the display is handled.

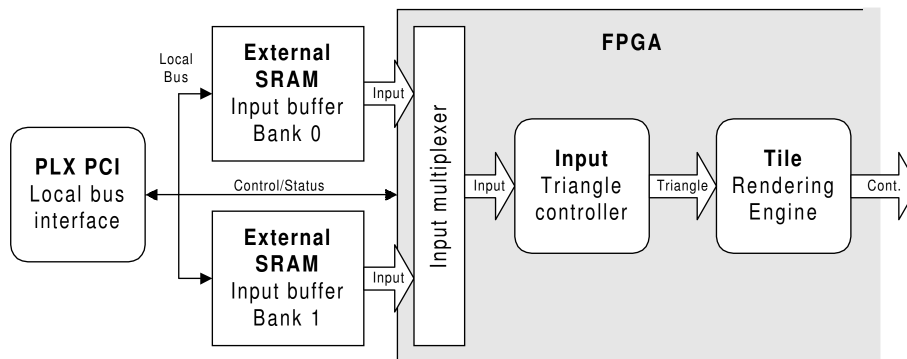

Figure 4.4: Data input for the FPGA. The input multiplexer allows the FPGA to read triangles from one bank while the PCI interface writes data to the other. The input triangle controller builds the internal representation of a triangle.

4.5.2 VGA video output

One of the major bottlenecks of the FPGA implementation in [207] is the transfer of the final rendered image back to the host PC. Since this transfer is done across the same PCI-bus as the source data transfer to the FPGA, the PCI-bus is continuously switched between sending triangle data and retrieving image data to and from the FPGA board. The required bandwidth for transferring animated images is very high when high resolution images are transferred at a high framerate. This also puts the FPGA implementation of Hybris at a disadvantage compared to other 3D graphics processors which are integrated in the video display hardware. The required bandwidth for transfer of a video image sequence can be formulated as follows:

$$B_{video} = width * height * depth * framerate \qquad (4.1)$$

where *width* and *height* specify the size of the image in pixels, *depth* is the number of bytes per pixel, and *framerate* is the number of image frames to be transferred per second. B_{video} is the bandwidth in bytes/sec. For example a low resolution image of 640x480 pixels with 8 bits/pixel at 30 frames/sec requires a bandwidth of 9.2 Mbytes/s, easily accommodated by the PCI-bus though it steals bandwidth from data transfer to the FPGA board. However the video bandwidth required for higher resolutions can be quite high. Transferring a typical image for a PC display with a size of 1280x1024 pixel in 24 bits/pixel true color at 75 Hz full frame rate requires a bandwidth of 295 Mbytes/s, about three times higher than the available bandwidth on the PCI-bus.

The best way to get rid of this extra bandwidth requirement is to integrate the 3D graphics processor in the video display hardware to get a dedicated communication channel to the display. Integration of the FPGA into a standard PC graphics adapter seems to be quite difficult, as detailed technical information about current PC graphics adapters is impossible to obtain. However a much more practical solution exists, namely to add a standard VGA (Video Graphics Array) video interface to the FPGA board itself. Since we do not want to build a PC graphics adapter from scratch involving complicated Windows drivers and VGA compatibility, we will still need a standard PC graphics adapter for Windows applications. The FPGA's VGA video interface thus only needs to interface with an analog RGB monitor. This approach for building a 3D graphics accelerator is essentially similar to the pioneering 3D graphics accelerators for the PC platform, namely the 3dfx Voodoo Graphics and Voodoo2 dedicated 3D graphics accelerators. These 3D graphics systems also required the presence of a standard Windows graphics adapter, and would switch the video output between the 3D graphics board and the 2D graphics board. The drawback of this approach is that a 3D graphics application using the dedicated 3D graphics accelerator must run in full-screen, while the 2D Windows display is not available at the same time (unless two monitors are used). If 2D Windows functionality must be integrated into the 3D hardware implementation, some of the techniques described in [186] might be useful.

Video output for a CRT (Cathode Ray Tube) based VGA monitor requires generation of five signals. Three analog colour signals for intensity of the primary colours `Red`, `Green` and `Blue`. Additionally two synchronization signals, `hsync` and `vsync` are needed for scan-line and frame synchronization. A CRT video display monitor is inherently analog and does not process digital pixel samples but continuous intensity signals. The maximum image resolution is determined by the video bandwidth and the *dot pitch* of the display tube. To display a digital image from a framebuffer on a CRT monitor the VGA video interface must serialize the pixel contents of the framebuffer into a continuous analog signal, and generate synchronization pulses between scanlines and frames which follows the VGA timing requirements. The CRT display draws an image on the screen using a raster-scan process by scanning an electron beam from left-to-right to build each scanline and from top-to-bottom to build the image from scanlines. At the end of each scanline the display must receive a horizontal synchronization signal to move (retrace) the beam back to the left edge in order to display the next scanline. Similarly the vertical synchronization signal marks the end of a video frame and causes the beam to retrace back to the top.

The human visual perception system is able to interpret the raster-scan pattern of a moving dot of light as a complete image, given a sufficiently high frame refresh rate of at least 60 Hz to avoid flickering.

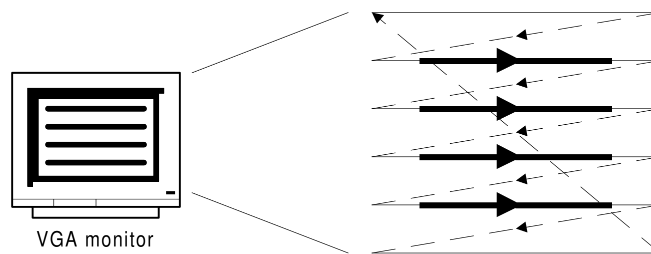

Figure 4.5: Raster-scan pattern of the electron beam in a CRT-based VGA monitor. Visible pixels are illustrated as thick lines in the raster-scan pattern. Horizontal and vertical retrace is indicated by the diagonal lines. Note the horizontal and vertical blanking intervals in the border around the visible display area.

First we consider a digital version of this video interface using a *pixel clock* as a reference for timing. The Standard VGA video signal for a 640x480 pixel display needs a 60 Hz frame refresh rate and a 31.5 kHz line rate. The visible display area is defined relative to the horizontal and vertical synchronization signals by horizontal and vertical blanking intervals. Figure 4.5 illustrates the raster-scan pattern of the electron beam and shows how the visible display area is formed. Standard VGA specifies timings for the synchronization signals and temporal placement of the visible area. In practice the VGA timings are expressed using the pixel clock to determine the duration of synchronization and blanking as a number of pixel clock cycles. E.g. to accommodate a visible area of 640x480 pixels as well as the blanking periods for synchronization requires a pixel clock frequency of 25.175 MHz to allow for a 31.5 kHz line rate with a total of 800 pixels of which 25.17 μs are the 640 visible pixels and 3.77 μs (95 pixels) form the hsync pulse in the remaining blanking interval, similarly for the 60 Hz frame rate we have a total of 525 scanlines of which 480 are visible and 2 are used for the vsync pulse.

The contents of the framebuffer, which was created using the tile rendering engine, now has to be displayed on the VGA monitor. This is done by fetching one pixel from the framebuffer per pixel clock during raster-scan of the visible area. In the FPGA implementation the framebuffer is stored in double buffered external SRAM memory, allowing the tile renderer to render one frame one tile at a time to the first buffer, while a VGA display processor reads pixels in raster-scan sequence from the second buffer. The framebuffers are stored in two independent memory banks, allowing full memory bandwidth to both the rendering and video display processors. A 2x2 crossbar switch allows switching between memory banks at any time, using tri-state buffers to allow switching between input from and output to

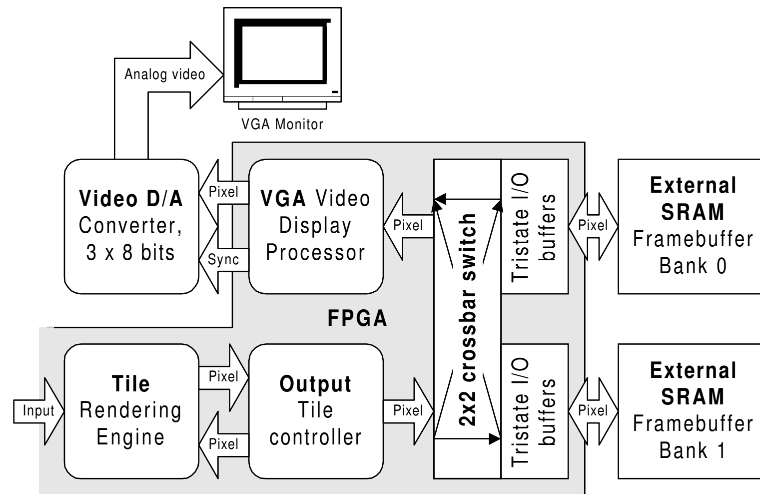

Figure 4.6: Integration of a VGA video output processor and the tile rendering engine in the same FPGA. A 2x2 crossbar switch and two external memory banks allows independent framebuffer access for the two processors.

the external SRAM banks. Figure 4.6 shows how the video output from the tile rendering engine is handled by integrating the VGA video output processor in the same FPGA.

The current FPGA implementation uses two clock domains to allow independent operation of the tile rendering engine and the video display processor. This allows setting the pixel clock to exactly the required frequency, while the tile rendering engine's clock frequency remains fully adjustable. Since the external SRAM is asynchronous, no problems are caused by switching the SRAM banks between two clock domains. If synchronous SSRAM or SDRAM is to be used in a future implementation, a more elaborate mechanism must be used to cross between clock domains.

Since the tile renderer processes one tile of 32x32 pixels at a time, the framebuffer also reflects this tiling pattern. However the video display processor requires a linear stream of pixels one scanline at a time. This is handled by reading one line of 32 pixels from each framebuffer tile intersected by the currently displayed scanline. An SRAM address calculation datapath handles this automatically. As the tiles are 32 pixels wide this framebuffer linearization method is also applicable to an SDRAM based framebuffer without too much overhead.

The video DAC (Digital to Analog Converter) used with the FPGA's video out-

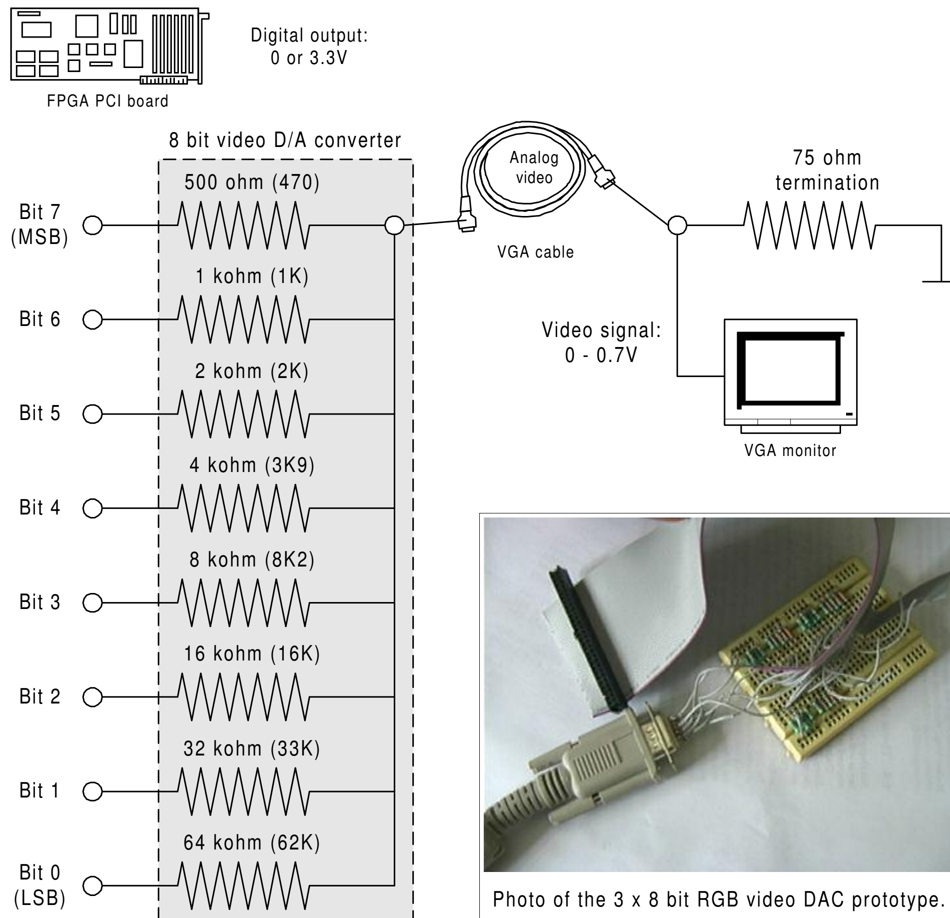

Figure 4.7: Schematics for the video D/A converter used for the FPGA implementation's VGA video output. Three of these 8 bit DACs are needed for 24 bit true colour RGB video output. In addition two TTL level signals, `hsync` and `vsync` are needed for line and frame synchronization and are connected directly to two pins on the FPGA. The photo shows the video DAC prototype which connects to the FPGA board using a ribbon cable.

put is a simple resistor network, driven directly by the output pins of the FPGA. Figure 4.7 shows the schematics for one of the three 8 bit video DACs. The resistor values are chosen to generate an analog signal voltage between 0 and 0.7V, assuming the VGA monitor provides a 75 Ω termination and the FPGA generates a 3.3V digital high voltage. By doubling the resistance for each lesser significant bit, a linear correspondence between digital pixel value and video signal voltage is achieved. The resistor values shown in figure 4.7 are the ideal values, with the closest standard resistor series values used for the prototype shown in parenthesis.

During practical testing the resistor network DAC provided a very nice and sharp picture when used in conjunction with the FPGA's output pins. The 640x480 pixel display was tested up to a pixel clock frequency of 50 MHz (corresponding to the monitor's maximum frame rate limit of 120 Hz) showing excellent image quality. A minor practical problem with "fine tuning" the resistor values in the resistor network was encountered. Since the resistor network was built using from standard series 5% accuracy resistors, it was quite difficult to balance all the resistors. Furthermore, the wiring pins on the FPGA prototyping board have a 27 Ω resistor in series which also has to be accounted for. These resistor inaccuracies cause some visible discontinuities in colour scales on the monitor, e.g. the intensity drops slightly when changing from pixel value 127 (MSB off, all other bits on) to 128 (MSB on, all other bits off) where the correct behaviour should be a slight increase in intensity. Fortunately these problems are very minor, and can be fixed by using precision resistors (1%) and trimming or simply by using a commercial video DAC or RAMDAC chip. A RAMDAC integrates a colour correction look-up table with the D/A converter to allow gamma correction and other calibrations.

4.5.3 Physical properties

The physical FPGA configuration bit-file is generated using the Xilinx Alliance Design Manager layout tool. The physical layout of the synthesized logic design is created automatically by mapping the logic to FPGA cells and performing automatic floorplanning, placement and routing of signals. The final FPGA configuration can be viewed and analyzed in the Design Manager. Figure 4.8 shows an overview of the FPGA floorplan. The floorplan shows how the cells of the FPGA are utilized for logic only and gives an idea of how much area is left for improving the design by adding more logic. The dataflow through the FPGA is predominantly from left to right. The triangle input read-ahead controller, which reads triangle nodes from the external SRAM banks connected to I/O pads on the left edge of the FPGA, is located on the left side. Triangle setup and drawing by scanline interpolation is located approximately in the middle and left region. Span setup and drawing by span interpolation is located in the middle and middle-right area. The

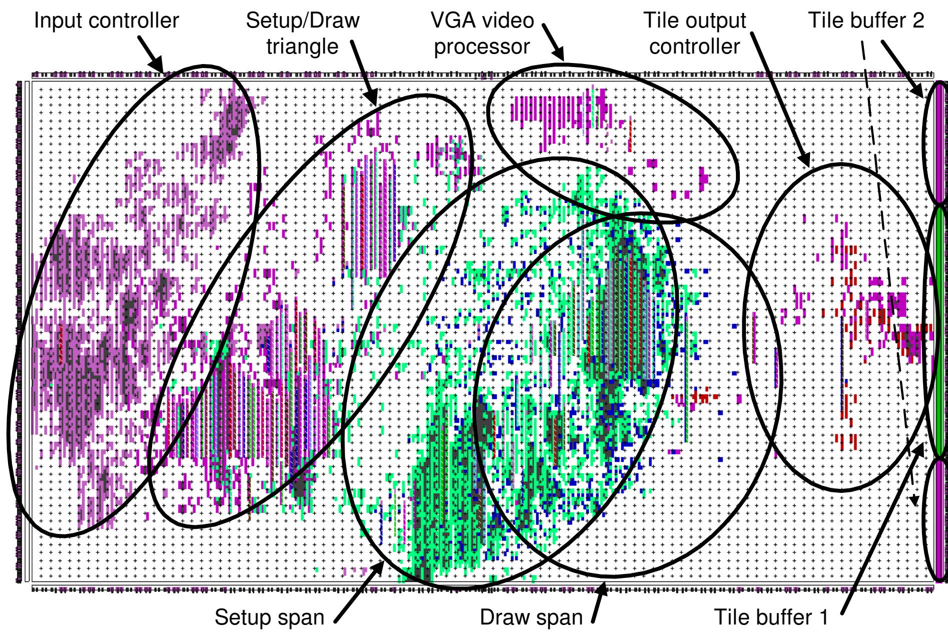

Figure 4.8: Floorplan of the FPGA. Shows how the synthesized logic is mapped and placed onto the configurable logic blocks of the FPGA.

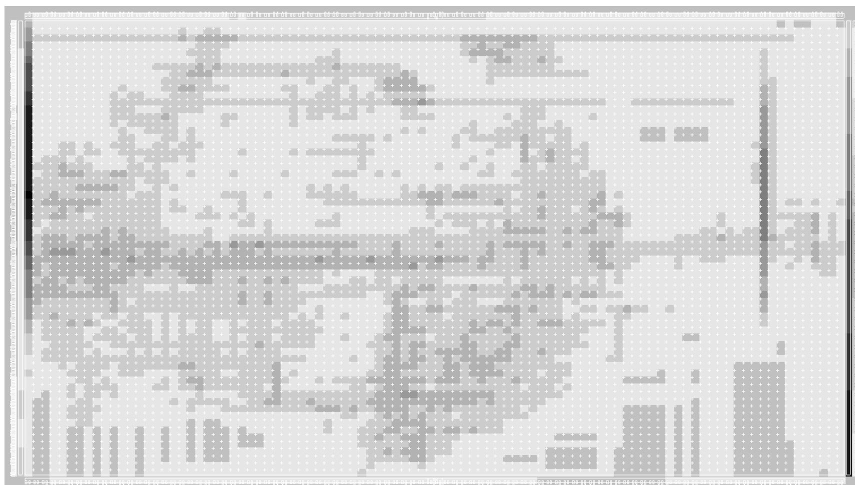

Figure 4.9: Contention map of the FPGA. Shows the total utilization for all the FPGA cells, including both logic and routing resources. Higher utilization is indicated by a darker colour, black indicates maximum utilization.

tile buffers and output controller crossbar switches are located to the right, near the BRAM tile buffers which form the right-most column of the FPGA. Finally the global framebuffer SRAM banks are connected to I/O pads on the right edge of the FPGA. The VGA video output processor is located in the upper-middle area, slightly to the right. The digital video output is connected to pads on the upper edge of the FPGA.

Figure 4.9 shows a contention map, which charts the total utilization for all the FPGA cells, including both logic and routing resources. Note how the extensive routing required to implement the triangle input controller causes high contention at the left edge of the FPGA. This is because the triangle data structure is represented internally in the FPGA as a 348 bit logic vector. Similarly some contention is caused near the BRAM tile buffers in the bottom right hand corner because of fan-out for the reset signal, which is not mapped to one of the dedicated clock nets.

Finally figure 4.10 shows the routing map, which illustrates how all the signal paths in the FPGA design have been routed. Note that the VGA video output signals for digital video are routed backwards in the FPGA, opposite of the general left-to-right dataflow direction, which may cause some routing contention. This is because the digital VGA video output pads are located on the upper edge, and the data source for the video processor is the global framebuffer connected to the external SRAM banks via pads on the right edge of the FPGA.

The FPGA implementation is currently very promising for future performance improvements. From the overview of the FPGA floorplans in figures 4.8, 4.9 and 4.10 it is clear that the Virtex XCV1000 FPGA is not yet fully utilized as a relatively large area of configurable logic blocks are currently not utilized. From the contention map we can see that there is some routing contention in the I/O border areas of the FPGA, but not in the interior. This suggests that we have plenty of space left for future improvements of the tile renderer by adding more logic.

From the Xilinx Alliance Design Manager layout tools we can get a report of the FPGA implementation's device utilization and performance statistics. This design report is listed in figure 4.11.

From these statistics we can see that only one third (31%) of the logic capacity is utilized, giving extra space for an implementation of e.g. the 2x2 interleaved pixel parallel tile renderer discussed in chapter 3, see page 83. This parallelization only adds extra logic (and wires) and does not need additional on-chip memory, although the BRAMs must be configured for a wider data interface to allow parallel data access. The BRAMs on the left edge of the FPGA may also be used to accommodate an extended tile buffer to better support colour rendering using 24 bits rather than 8 bits per pixel. However there is an additional routing delay and possibly more contention if signals are routed across the FPGA to use the left BRAMs as well as the right BRAMs for the rendering process.

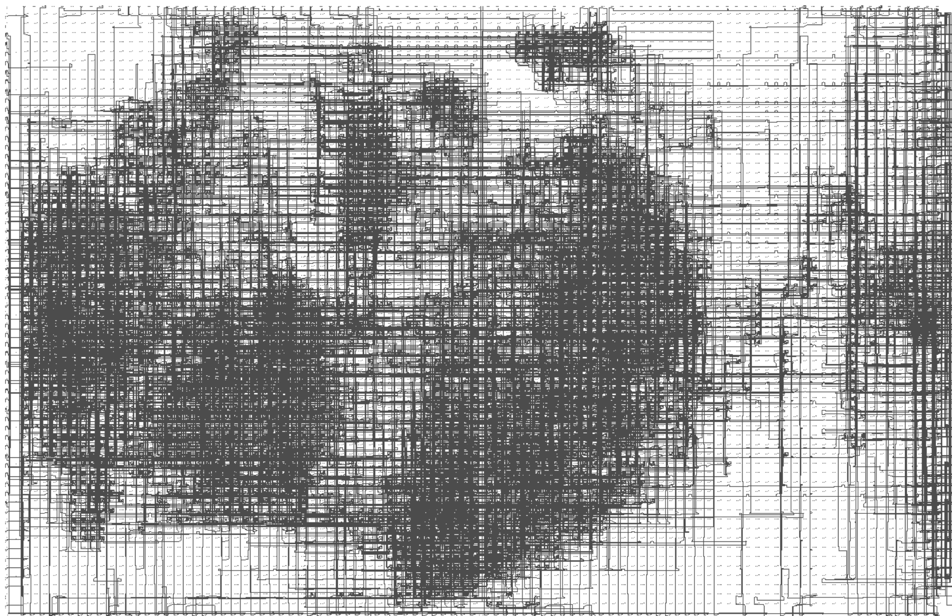

Figure 4.10: Fully synthesized graphics hardware. FPGA place & route layout for the tile rendering back-end and VGA video output mapped to a Xilinx Virtex XCV1000 FPGA.

```
Design Summary
--------------
   Number of Slices:              3,932 out of 12,288   31%
   Number of Slice Flip Flops:    2,138 out of 24,576    8%
   Total Number 4 input LUTs:     6,548 out of 24,576   26%
      Number used as LUTs:                      6,543
      Number used as a route-thru:                  5
   Number of bonded IOBs:           293 out of    404   72%
   Number of Block RAMs:             16 out of     32   50%
   Number of GCLKs:                   2 out of      4   50%
   Number of GCLKIOBs:                2 out of      4   50%

Total equivalent gate count for design:  331,682
Additional JTAG gate count for IOBs:  14,160

Design statistics:
   Minimum period:  42.252ns (Maximum frequency:  23.668MHz)
   Maximum net delay:  10.997ns
```

Figure 4.11: The FPGA implementation's device utilization and statistics, as generated by the Xilinx Alliance Design Manager layout tools when targeting the Xilinx Virtex XCV1000-6 FPGA.

Note also that only 8% of the flip-flops are utilized, meaning that any pipelining applied to the datapaths in the FPGA design is essentially free. Pipelining may allow the design to operate at a higher clock frequency. The current FPGA design operates reliably at a 25 MHz clock frequency without pipelining of the datapaths.

If we improve the internal speed of the FPGA we might run into problems with the external asynchronous SRAMs as they can be operated reliably only up to about 35 MHz from the FPGA. This was discovered during the design of the VGA video output processor, as it uses a different clock domain than the tile renderer, allowing its clock frequency to be set independently of the tile renderer's clock. However, since the tile renderer uses internal on-chip buffering for the tile-based rendering, a future design can be allowed to run on a faster clock than the external memory. If the internal processing speed can be made far higher than the external memory speed, we may use the extra computational power to improve the rendering quality e.g. by applying anti-aliasing as discussed in section 3.6.6 of chapter 3, demonstrating other aspects of the Hybris architecture's scalability properties.

Model name :	Buddha		Bunny	
Size (triangles) :	1,087,716		69,451	
Implementation :	frames/s	triangles/s	frames/s	triangles/s
New Single CPU	2.5	2,719,290	19	1,319,569
Single CPU	2.2	2,392,975	16	1,111,216
Dual CPU	3.9	4,242,092	29	2,014,079
FPGA	1	1,087,716	12	833,412
ASIC (simulated)	16	17,403,456	n/a	n/a
GeForce 2 GTS	1.6	1,740,346	20	1,389,020

Figure 4.12: Measured performance figures for different implementations of the Hybris graphics architecture. For reference the commercial Nvidia GeForce 2 GTS graphics processor is included.

4.6 Performance figures for the implementations

This section lists some comparative performance figures for renderings on the Windows 2000 based test PC which features dual Pentium III 500 MHz CPUs, the FPGA prototyping board with integrated video output and on the ASIC implementation (simulated). For reference we compare the performances of Hybris with the commercial Nvidia GeForce 2 GTS[1] 3D graphics processor.

The test objects are the Stanford Buddha and Bunny models. The Buddha is made of 1,087,716 triangles, and the Bunny consists of 69,451 triangles. Figure 4.12 lists the performance observed while rendering these objects with the different implementations of Hybris, as well as the GeForce 2 performance reference. Both frames/s and triangles/s are listed. Note that the triangle rate is calculated from the total number of triangles in the test object. The number of visible triangles is usually less than half, as it depends on dynamic back-face culling and triangle rounding.

While the dual Pentium III parallel software implementation of the Hybris graphics architecture achieves the highest performance levels, the FPGA implementation is also beginning show quite high performance levels. For the Bunny test case, the FPGA performs almost as well as the single CPU software renderer.

The dual CPU implementation demonstrates an observed speedup relative to the single CPU implementation of 1.81 for the Bunny and 1.77 for the Buddha test objects. While the observed speedup is not linear, this should be considered to be quite good, considering that some serially executed code still exist in the user input

[1] GeForce 2 GTS: See chapter 2 and `http://www.nvidia.com`

interface and display output interface, e.g. for every frame the final rendered image must be copied from main memory to the display memory on the AGP video card. This means that the actual speedup of the parallel renderer itself is higher, if we compensate for the serially executing part. From rendering of an empty scene, we get an idling framerate of about 120 Hz, representing the serially executing code in the system. Using Amdahl's Law (4.2), see [174], we can then estimate the actual speedup of the rendering code in the dual CPU implementation. Amdahl's Law states that the total execution time is the sum of the execution time of the improved (or parallelized) code and the unimproved (or serial) code:

$$t_{total} = \frac{t_{parallelizable}}{parallel\ speedup} + t_{serial} \tag{4.2}$$

which for the single CPU case looks like this:

$$t_{singlecpu} = t_{parallelizable} + t_{serial} \tag{4.3}$$

and for the dual CPU case:

$$t_{dualcpu} = \frac{t_{parallelizable}}{dualcpu\ speedup} + t_{serial} \tag{4.4}$$

The observed speedup is calculated by:

$$observed\ speedup = \frac{t_{dualcpu}}{t_{singlecpu}} \tag{4.5}$$

The estimated dual CPU parallel speedup can then be determined by:

$$dualcpu\ speedup = \frac{t_{parallelizable}}{t_{dualcpu} - t_{serial}} \tag{4.6}$$

substituting with (4.3) we get:

$$dualcpu\ speedup = \frac{t_{singlecpu} - t_{serial}}{t_{dualcpu} - t_{serial}} \tag{4.7}$$

By inserting the observed performance and the observed serial execution time in equation (4.7) we can get an estimate of the dual CPU parallel speedup. This gives the following speedup figures, based on idle performance of 120 frames/s:

Bunny: Parallel speedup: 2.07 (observed speedup: 1.81)

Buddha: Parallel speedup: 1.80 (observed speedup: 1.77)

From these figures we get the very interesting result that in the case of the Bunny test object, the dual CPU parallel renderer achieves *superlinear* speedup. This can be explained by the way the Hybris graphics architecture uses memory partitioned in small localized blocks, which improves the cache utilization. When two CPU's are used, each of them process only one half of the dataset while the total cache size is doubled. This results in a better cache utilization than with the single CPU which must use a single cache to process all data.

However, the speedup is not superlinear with the Buddha test object. A good reason for this is the very large data size of the 3D model which causes extensive main memory bandwidth contention. This is a weakness of the shared memory multiprocessor platform, as the CPU's in the dual CPU platform use the same main memory architecture as the single CPU platform. A multiprocessor computer architecture using a crossbar switched multi-banked memory architecture would help in our case, as the two worker processes in the dual CPU implementation of Hybris do not simultaneously access the same memory locations.

The ASIC implementation performs very well in simulation, but it should be noted that the input data-source and output data-sink are assumed to be ideal, i.e. they are not the bottleneck, allowing the ASIC implementation of the tile rendering engine to run at maximum speed. This may not be true in a real system, so the 16 frames/s should only be interpreted as an indication of the achievable speed for an ideal graphics system using a single tile rendering engine. Note that the actual internal triangle rate in the ASIC tile renderer is 4.2 Mtriangles/s after back-face culling and rounding. The simulated performance is for the predicted 27 MHz operational frequency. Under the realistic assumption that a final ASIC implementation can be scaled to 100 MHz [71], the internal tile rendering performance would be 59 frames/s or 15 Mtriangles/s (for a dense triangle mesh such as the Buddha). A triangle heap based on 64 bit wide SDRAM must run at least at 120 MHz to support this triangle rate (using the 64 byte triangle nodes). Given that contemporary 3D graphics processors employ 128 bit wide 460 MHz DDR SDRAM memories and operate at up to 200 MHz internal clock frequency, there is plenty of room for improvement of the ASIC implementation's performance, e.g. by parallelizing it, as such a memory subsystem would support a triangle heap bandwidth of over 100 Mtriangles/s. The task of designing a parallel front-end geometry processor capable of supplying these data rates is quite challenging, but not impossible.

The GeForce 2 GTS performance results were made using the GLView VRML viewer which was the only 3D viewing software found to be capable of handling the same test 3D models at an acceptable speed. Other viewers were found to work very slowly. It should be noted that the GeForce 2 GTS is capable of rendering at higher performance levels when the object database is small enough to fit in its vertex buffer, eliminating triangle communication overhead completely. Unfortu-

nately even the small Bunny test model is too big to fit in the GeForce 2's vertex buffer. The GeForce 2 is mainly optimized for high quality texture mapping of large triangles in 3D computer games, not millions of small un-textured triangles.

In conclusion to this performance comparison it appears that a parallel software renderer is currently the best solution for rendering very complex 3D models with millions of triangles. However the hardware renderers may catch up on this number in the future.

4.7 Prospects for future HW/SW implementations

Several different future implementations of the Hybris architecture are possible by applying hardware/software codesign to arrive at other designs possibly achieving even higher performance levels.

Some parts of the front-end graphics pipeline might well be implemented in hardware. The most likely candidate is the triangle set-up calculations which may be performed prior to bucket sorting as the last operation in the front-end or alternatively as the first operation in the back-end. The triangle set-up phase is quite computation intensive as it involves division to calculate the triangle slopes and differentials for parameters to be interpolated across each triangle.

Division is a bit tricky to implement in hardware, as e.g. Synopsys will only allows synthesis of the '+','-' and '*' operations in VHDL. To synthesize the division operation a regular structure somewhat similar to array multiplication may be used. The paper [241] presents an arithmetic unit generator for VHDL which is able to generate arithmetic units configurable for many word sizes, including division operations. This generator is best suited for ASIC technologies, however. For the FPGA the Xilinx Core Generator can generate a customizable pipelined divider suitable for our purpose. A generic description of arithmetic structures for datapaths can be found in [98].

The triangle set-up stage in Hybris was recently transformed to exclusively use integer and fixed-point arithmetic, so no complex floating point units are required. This makes the triangle set-up stage implementable for both ASIC and FPGA technologies, using the division circuits just described.

Future improved ASIC and FPGA technologies will enable more elaborate implementation of the Hybris graphics architecture. Some of the available options with denser process technologies are implementation of multiple processor elements by leveraging the scalability of the Hybris architecture, e.g. several parallel back-end tile engines and/or several parallel front-end object geometry engines. A mentioned in the previous section, the triangle heap bandwidth is not a problem as

we can use new memory technologies such as DDR[2] SDRAM. Note that a highly scaled implementation of the Hybris graphics architecture would need a crossbar switched multi-banked memory architecture to support simultaneous data streams to and from multiple parallel front-end and back-end processors.

Embedded DRAM manufacturing processes are gaining popularity in the semiconductor industry, and may be usable for on-chip buffering in future implementations of the graphics architecture. The main advantage of using embedded DRAM is a far higher on-chip memory capacity, allowing e.g. an on-chip global framebuffer. An example of a graphics processor which uses embedded DRAM to integrate the entire global framebuffer is the work-in-progress Glaze3D™ presented at [168], which integrates 72 Mbits of 512-bit wide embedded DRAM running at 150 MHz to achieve high memory bandwidth. A related example is the ATI Mobility graphics processors for notebook PCs which also integrate the framebuffer using on-chip embedded DRAM, but this time for its low-power properties as access to off-chip external memory is quite power consuming.

While the Hybris design works best with dual-ported SRAM memories in its localized buffers for high-speed parallel access, some buffers may be implementable using embedded DRAM memories. However care must be taken to properly handle memory refresh, paging and bank management if DRAM is used. An alternative to regular embedded DRAM is "1T-SRAM" [126] embedded memory, which uses a special caching and fine-grained multi-bank switching scheme to hide memory refresh delays, making a DRAM array behave as a single-ported SRAM. The 1T-SRAM IP-blocks may also be used in standard logic processes. According to [70] embedded 1T-SRAM is used in the next generation Nintendo game console, a new competitor for Sony's PlayStation 2.

In the near future Xilinx will start producing the Virtex II Pro FPGA, which includes hard-core PowerPC CPU cores embedded in the array of configurable FPGA cells. While a soft-core CPU is implementable in a normal FPGA, a hard-core CPU is far more area-efficient and faster as well. This allows very tight integration between hardware and software components in future system-on-chip (SoC) implementations using these FPGAs. The front-end geometry engine might be implemented by using a hard-core CPU closely coupled with SIMD vector coprocessors (similar to [132]) implemented in the FPGA fabric. As the Virtex II FPGAs include a large number of hard multiplier cores they can be used to perform the many multiplications needed in an implementation of the front-end pipeline.

---

[2]DDR: Double Data Rate, i.e. data transfer on both rising and falling clock edges.

4.8 Chapter summary

In this chapter some of the possible hardware/software codesign implementation options for the Hybris graphics architecture which was presented in the previous chapter have been examined. We have examined how virtual prototyping codesign using the C programming language allows the Hybris graphics architecture to be mapped to different implementation technologies, ranging from sequential software and parallel software for general purpose computers with one or more CPUs, over to more elaborate hardware/software codesign implementations. ASIC and FPGA implementations of the back-end tile rendering pipeline have been successfully implemented, although some work still remains in order to get superior performance from the FPGA hardware implementation.

Chapter 5

Interfaces and Applications

This chapter discusses how to interface with an application that uses the graphics architecture. An implementation of VRML 97 is used as a high-level interface to the Hybris graphics architecture. Finally the 3D-Med medical visualization workstation is presented as an example of an application which uses the graphics architecture.

In order to implement many of the techniques required for high performance scalable graphics, a high abstraction level of the application programming interface (API) is required. This chapter describes how VRML 97 can provide a high-level interface allowing optimizations such as automatic object partitioning, which is needed to achieve good scalability and performance in the Hybris graphics architecture. This is compared to other lower-level standard graphics API's such as OpenGL. In order to gain scalability for high performance levels it is not enough to optimize only the low level triangle rendering architecture. Optimizations on a higher level of abstraction can give good results which may even be impossible to achieve by optimizing only the lower levels of an architecture.

5.1 Virtual Reality

The highest level of abstraction in an interactive graphics system is the user interaction feedback loop. An image is presented on the screen, the user then interacts by giving feedback that affects how the next image is generated. If this feedback loop runs fast enough, and has a sufficiently short latency, the user will get the illusion of manipulating something "real". This feedback is the basis for *Virtual Reality*. A general tutorial on virtual reality concepts is found in [123].

A critical factor of the user interaction feedback loop is the rate of image display. This is measured in frames per second. At framerates less than one frame per

second (fps), interactivity suffers as the user must wait for the next image to appear. According to [152] a framerate of 6 fps is enough to get a sense of interactivity, 15 fps provides good interactivity up to around 70 fps, above which interactivity is not further improved because of limitations with displays and human perception.

Another critical factor is the *latency* of the feedback loop, i.e. the time that passes from user input until a visible response is displayed. Latency is related to framerate, as a faster framerate reduces the latency. Additionally the 3D graphics system and the user input devices also add latency to the system. Note that latency typically is in the order of a few frames, depending on the amount of pipelining in the graphics system.

The visual quality of the resulting images is also important to create a convincing illusion of a "virtual reality". Besides photorealism and nice looking images, anaglyphic 3D *stereo* images are able to give the user a sense of depth perception. An anaglyphic 3D stereo image is presented to the user by showing an image rendered from the point of view of the human user's left eye to that eye *only*, similarly for the right eye. This can create an illusion of depth similar to what is possible with stereo photography.

Several methods for separating the images for each eye are available. These include: Red/green images + red/green colour filters, colour/monochrome images + special filters, alternating images + shutter glasses, horizontal/vertical polarized light images + horizontal/vertical polarization filters, head-mounted binocular displays, fresnel-lens LCD stereo displays, etc.

These display methods all require rendering of two separate images, using a slightly different view-point for each image to simulate the distance between the human eyes. To properly align each image requires correction for the shape of the display surface as mentioned in [42]. However our experiments with stereo rendering in Hybris, using the red/green filters and shutter glasses methods, shows that the human visual system is able to compensate for these problems. Further, our experiments showed that anaglyphic 3D stereo rendering provided a greatly enhanced sense of depth in the rendered images. For example it was quite confusing to get a good idea of the three-dimensional structure of the veins in a 3D model of the liver reconstructed from a 3D ultrasound scan, but by using 3D stereo rendering it was possible to get a good impression of the three dimensional vein structure. In Hybris the 3D stereo rendering was implemented by rendering an image for the left eye, moving the location of the viewpoint horizontally, and then render the image for the right eye.

In order to implement virtual reality or any other kind of 3D visualization we need an API to define how an application should use and interface with the 3D graphics architecture. In the next section we will examine how such an interface can be made.

5.2 3D application interfaces

In general, 3D graphics interfaces fall in two categories, immediate-mode and retained-mode APIs. The main distinction between these APIs is that immediate-mode APIs specify a relatively low level of graphics primitive instructions which immediately cause rendering to the display in the same order they are executed, while retained-mode APIs allow a more abstract hierarchical 3D object structure to be specified with no guarantee about the actual rendering order of the individual graphics primitives.

5.2.1 Immediate-mode graphics interface

An immediate-mode graphics API presents an interface which allows the application to specify a graphics primitive which is then immediately rendered into the framebuffer. A subsequent API call to render a primitive can therefore assume that all pixels affected by the previous call have been rendered to the framebuffer. This imposes a strict ordering of graphics primitives. Also there is in general no way to know when a frame is finished, as it is possible to render into a framebuffer while it is being displayed.

Most commercial 3D graphics processors use a low-level immediate-mode graphics API such as OpenGL [165, 204] or DirectX [148]. Typically OpenGL is the *de facto* API for 3D computer graphics. OpenGL has evolved from Silicon Graphics' IrisGL to a portable immediate-mode graphics API suitable for many application. The primary focus for OpenGL is technical CAD applications. While many possible implementations of these interfaces exist [118], they are basically limited to a serial immediate-mode interface model.

Recently research into parallel immediate-mode graphics interfaces has been made, such as described in [99, 101] and implemented in the scalable Pomegranate architecture [51] and WireGL [27]. While such a parallel API can be scalable, it does not solve the important problem of *partitioning* the input. The application must provide a balanced number of triangles to each interface at each node.

Because such a serial immediate-mode graphics interface may limit the maximum performance of a graphics system if an application does not use the API properly, a retained-mode graphics interface may be preferred as the application can rely on the graphics system to optimize rendering.

5.2.2 Retained-mode graphics interface

A retained-mode graphics API is an interface which accepts a single description of the graphics objects to be rendered. This may be a hierarchical description

involving multiple objects, each defined in their own coordinate systems defined relative to other coordinate systems. Such an object hierarchy is known as a scene graph.

Although a retained-mode API places limitations on the flexibility of a graphics system such as by imposing fixed constructs for objects, it may provide improved performance by statically and dynamically optimizing the scene description data required for rendering.

The origins of the retained-mode API can be found in the PHIGS (Programmer's Hierarchical Interactive Graphics System) API, a standard, dynamic and interactive graphics interface, which is described in [211]. PHIGS uses a hierarchy of data structures, which are essentially classes of objects like in C++. An overview of how classes can be used to build hierarchical scene graphs in retained-mode graphics interface software is given in [74]. While the ANSI PHIGS standard is not in widespread use today, many of the ideas survive in recent retained-mode graphics systems such as SGI OpenInventor [210], the retained-mode API in DirectX [148] and SGI's IRIS Performer multiprocessor API [192]. An example of a scalable retained-mode interface is the distributed large-screen rendering system described in [196, 195].

The ISO VRML 97 standard [28] (Virtual Reality Modeling Language) which is based on many of the ideas in OpenInventor, can also be interpreted as a retained-mode scene graph graphics API. OpenWorlds [47] is such a VRML-based object-oriented scene graph API for C++. Recently Sun's retained-mode Java 3D API [212] is gaining popularity in the VRML / X3D community, making rapid development of VRML based 3D applications possible using the Java programming language [213, 61, 60].

The VRML scene graph representation used in the Hybris graphics architecture is another example of such a retained-mode graphics system. For implementation of the Hybris VRML browser, a hierarchy of VRML nodes are used, which are represented by implementing each node as a C++ class.

Other VRML browser implementations which have been described in publications include i3D [8], VRweb [183], VRwave (a Java version of VRweb) [6], as well as OpenWorlds [47] which also exposes its internal classes as a commercial retained-mode scene graph API similar to OpenInventor. These techniques are also quite usable in computer game engines [15].

Note that recently the popular immediate-mode graphics APIs OpenGL and DirectX are integrating some concepts from the retained-mode APIs, such as object representations using vertex and index buffers which allow the application to send indexed triangle meshes rather than individual triangles or triangle strips to the graphics system. A higher-level retained-mode interface can be used to encapsulate the immediate-mode API to enable automatic object partitioning and optimization.

As a historical note, the Silicon Graphics and Microsoft *Fahrenheit* API project was an optimistic attempt to merge OpenGL and DirectX into a unified retained-mode optimizing scene graph API.

5.3 The Hybris VRML engine

The Hybris graphics architecture needs an API interface in order to be usable for real applications. Since one of the requirements is that we need freedom to experiment with internal object representations, an immediate-mode API is not ideal. A retained-mode API seems like a good model for this. Further, the platform independent VRML 97 language does not even require a formal API to be specified as long as the graphics system can read the VRML scene description. Since VRML provides a very high abstraction level of the graphics interface, we gain the ability to perform any optimizations we may think of. This is why VRML 97 was chosen as the interface to the Hybris graphics architecture.

The VRML language can be used by itself to specify interactive behavior in a virtual world of 3D objects, as internal sensor nodes are able to generate events in response to user interaction as well as the flow of time. These events are routed to other nodes in the VRML scene graph which can respond by activating predetermined animations. The Hybris VRML engine implements this event routing system to allow animation and interactive manipulation of the scene. The event routing system is implemented in an efficient way to avoid excessive memory copies of event data. See [237] for further details about event routing.

A VRML extension implemented in Hybris' VRML engine is the Java EAI (External Authoring Interface) which is described in [135]. EAI specifies an API for the VRML engine with interfaces to the Java programming language. The EAI API works by exposing the VRML event routing mechanism to allow events to be sent and received to and from an "outside" Java application. The Java Native Interface (JNI) is used to implement efficient communication between the C++ based Hybris VRML scene graph software and a set of Java classes forming the EAI interface. This approach allows a minimal interface to be used for run-time program interaction with the VRML engine, and is useful for integration in e.g. a Java application such as the medical visualization workstation described later. See [86] for further information on this Java / VRML integration.

Further, Hybris' internal object-oriented C++ scene graph classes can be exposed to outside applications, providing a flexible high-level retained-mode graphics API. An example of such a graphics API architecture of an open VRML engine is found in OpenWorlds [47]. In Hybris the C++ classes which define the scene graph can be subclassed to build entirely new functional blocks for the VRML en-

gine, which can then be accessed from the VRML language as extended VRML nodes. As example of this is the `Volume` VRML node described later, which integrates direct volume rendering into VRML.

Finally it should be noted that the low-level interfaces used internally in the Hybris graphics architecture might be exposed as an immediate-mode API. If the object-partitioning system is exposed as an additional tool-kit API, the remaining part of the graphics architecture can be accessed through an implementation of e.g. OpenGL. This approach can potentially provide a way to allow existing OpenGL based applications to use the Hybris graphics architecture for rendering. Since other tile-based hardware renderers such as the PowerVR provides an OpenGL interface, this will also be possible for the Hybris graphics architecture.

Note that a complete frame of data to be rendered is buffered in the internal triangle heap of a sort-middle tile-based graphics architecture such as Hybris or PowerVR. Because of this, support for incremental updates to a display is either not supported or at least implemented in a less efficient “compatibility-mode”. Such incremental updates are typically only used for some CAD applications and the “3D Pipes” Windows screen saver. Fortunately most virtual reality type applications render complete frames at high framerates.

5.4 Introduction to visualization

Medical imaging normally involves the handling of very complex models which leads to great demands on memory and computational resources. Data sets are seldom smaller than 30-40 Mbytes. Traditionally medical imaging has been performed on large workstations, but recently PCs have become more powerful and are today quite capable of handling large data sets. The recent availability of Java and VRML raises the question of whether such high-level languages can be used for 3D medical imaging on a PC.

For volume visualization two approaches are commonly used; direct volume rendering and extracted iso-surface rendering.

5.4.1 Direct volume rendering

Direct volume rendering allows the 3D volume data to be visualized directly, which has the advantage that all data in the volume may contribute to the final image.

While direct volume rendering can be done easily by ray-tracing, this is very inefficient and slow, as each voxel may be processed several times. A more efficient algorithm traverses the volume in data or slice order to reduce redundant data accesses. A popular algorithm for doing this is the shear-warp volume rendering

algorithm [122]. Shear-warp is a two pass algorithm which decomposes the 3D viewing transformation into two simpler transformations, a shear transformation along one of the major axes of the volume, followed by a 2D warp to make the result match a 3D view transformation. A parallel version of the shear-warp algorithm is presented in [198]. In [39] a version of shear-warp for surface rendering of sparse volumes is presented.

The Cube-4 [179] volume rendering architecture developed at SUNY - Stony Brook is related to the shear-warp architecture in that it processes one slice at a time. Cube-4L [16] is an extension for perspective projection. EM-cube [171] is an efficient version of Cube-4, which is used in the commercial Mitsubishi VolumePro implementation [180]. VolumePro is able to interactively render 256^3 volumes at 30 frames/s. Other examples of direct volume rendering hardware can be found in [128, 190].

During the visit at the Visualization Lab at SUNY-SB some insight in the design of volume visualization systems was gathered. Based on this, a more efficient implementation of our shear-warp volume renderer, originally implemented by [109] and [131], was integrated into the VRML scene graph engine of Hybris. This improved integrated volume renderer, while still primitive compared to CUBE-4, now provides *interactive* direct volume rendering in software, suitable for use in the 3D-Med medical visualization workstation described later in this chapter.

As the volume renderer is integrated in the Hybris VRML engine it is also possible to display combined surface and volume renderings, although the integration is not yet optimal, as surface models are currently always rendered on top of the volume models. A very similar example of how surface and volume rendering can be combined by using VRML was published this year in [12].

Our VRML Volume rendering extension is defined as this extended VRML node prototype, which must be used in the `geometry` node placeholder field of the `Shape` node, as it is subclassed from the abstract `Geometry` node:

```
Volume {
  exposedField MFString url          []
  exposedField SFBool   intermediate FALSE
  exposedField SFBool   perspective  TRUE
  exposedField SFInt32  slice        -1    #[-1, slices-1]
}
```

The new `Volume` VRML node works exactly as any other geometry node such as `IndexedFaceSet`, and can be located anywhere in an object hierarchy. The `url` field is a URL[1] specifying where to find the volume data file, the

[1]URL: Universal Resource Locator, similar to an internet WWW link.

`intermediate` field specifies if the intermediate (sheared) image should be displayed rather than the final (sheared and warped) rendered image, `perspective` indicates whether we want a parallel- or perspective-projection rendering, and finally `slice` allows rendering of the complete volume (-1) or just one slice. The size and voxel dimensions of the volume is determined automatically from the input volume data file.

True integration of surface and volume rendering requires a hybrid volume and polygon graphics architecture capable of sorting the polygons relative to the slices in the volume in order to handle transparency correctly. This is actually a generalization of a fragment sorting graphics architecture which is required for proper handling of transparency, as discussed earlier in this thesis. A possible extension of the CUBE volume rendering hardware architecture to allow hybrid volume and polygon rendering with slice-level transparency sorting is published in [120].

Finally, general purpose 3D graphics hardware with an efficient implementation of texture mapping is increasingly becoming usable for real-time volume rendering. 3D texture-mapping can be used to render a volume by mapping a volumetric texture to a set of polygons. Note that 3D texture mapping must be implemented in hardware, which is currently only available in high-end graphics workstations such as the SGI InfiniteReality. The pixel blending operations in the texture mapping hardware can be used to implement shading and classification of the volume, by encoding the voxel gradient vectors as the RGB components and the intensity as the Alpha component in the 3D texture map. This type of volume rendering based on 3D texture mapping is presented in [67, 232, 146]. Gradient calculations must be done as a pre-processing step as the texture mapping hardware is unable to do this.

Recent advances in graphics hardware for single pass multi-texturing allows volume rendering using 2D texture mapping at high speed on standard PC-based 3D graphics processors, using an approach presented in [191]. Multi-texturing is used to dynamically interpolate between volume slices, improving the quality of the rendering. Additionally, as recent PC graphics processors such as the Nvidia GeForce has introduced per-pixel dot-product operations, this can be used to implement dynamic lighting for shaded volume rendering. Since the 2D texture map based algorithm in [191] uses a slice oriented memory layout similar to the shear-warp algorithm and CUBE, it can be implemented more efficiently than 3D texture mapping based volume renderers.

5.4.2 Surface model extraction

To visualize volume data it is also possible to extract a triangle mesh model representing an iso-surface in the volume data. This triangle mesh model is suitable for rendering with an implementation of the Hybris graphics architecture. A suitable algorithm for iso-surface extraction is the Marching Cubes algorithm [129].

A discretized version of the marching cubes algorithm, similar to the one presented in [157], was developed by [160] and is currently used in the 3D-Med workstation to extract iso-surfaces. It works by matching eight volume values (a cube) to a finite set of triangle configurations which is then adjusted to best match the location of the iso-surface, using a discretized rather than continuous set of vertex locations to speed the process. By repeating this process (marching) over the entire volume, a triangle mesh is constructed which represents an iso-surface in the volume data set.

5.5 The 3D-Med medical visualization workstation

With the current implementation of the 3D-Med medical visualization workstation, the user is able to perform three-dimensional measurements on 3D medical images in a physically correct coordinate system, while retaining the original 2D slice images from the CT scanner. Using MPR (Multi Planar Reformatting) while measuring makes it possible to compensate for misaligned scans by realigning the 3D volume to match the anatomical plane. This makes the system usable for practical clinical measurements e.g. for orthopedic surgery on femoral anatomy [50, 83]. Currently the 3D-Med system is in use at Århus Kommunehospital where it is used for surgery planning and diagnostics of acetabular dysplasia [163], prior to orthopedic surgery on the pelvis or femur.

The visualization workstation is based on the work presented in the paper [86] which describes the Java application framework for handling the raw volume data, as well as possible interfaces to 3D visualization modules based on the Hybris graphics architecture. Some of the work done in [97] as well as [222] form the concepts behind the Java software framework.

During the recent work with implementation of Hybris' Java interface, the 3D-Med prototype was extended with Java/C++ interfaces to allow a 3D region of interest subsection of the dataset to be selected. This region of interest is then used for detailed examination with our visualization tools, either as a direct 3D volume rendering visualization or by extracting an iso-surface to be viewed using the scalable Hybris 3D surface rendering hardware/software architecture. 3D viewing allows the user to get a better overview of the three-dimensional structures present

in a volumetric 3D medical image. Figure 5.1 shows a screen-shot of the working prototype 3D-Med medical visualization workstation.

All the components in the system are working prototypes developed here at CST, IMM, DTU. The 3D-Med workstation eliminates the need for expensive hardware such as Silicon Graphics visualization workstations, making it possible to perform the visualization tasks using an ordinary PC. For use in a hospital environment, the 3D-Med workstation is able to directly read DICOM / ACR-NEMA [2] formatted datasets generated by a CT scanner.

5.6 Chapter summary

This final chapter has given an introduction to graphics interface APIs in general, as well as a possible application for the graphics architecture. VRML was introduced to provide a high level of abstraction to allow the static and dynamic optimizations used in the Hybris graphics architecture. Additionally an overview of volume rendering was presented in relation to the surface rendering techniques used in the Hybris graphics architecture. As an example of a useful application of the graphics architecture, the working prototype 3D-Med medical visualization workstation was presented.

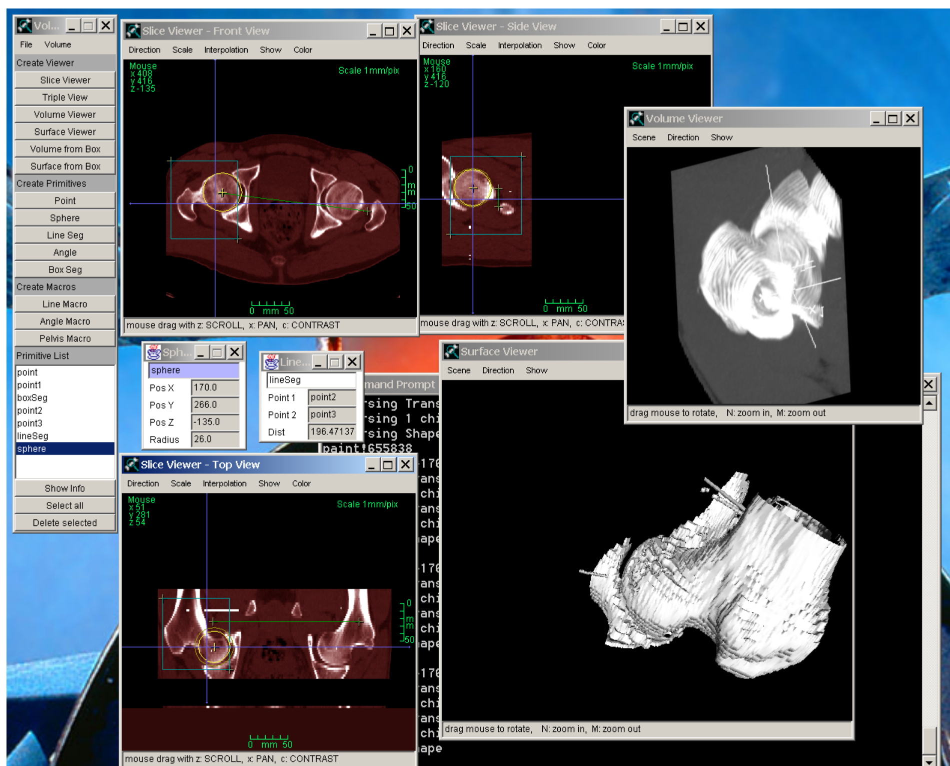

Figure 5.1: Screen-shot of the working prototype 3D-Med medical visualization workstation. In the lower right side of the screen a software implementation of the Hybris graphics architecture is interactively rendering a surface model. The surface model is extracted from the volume data displayed in the other windows. In the upper right side the shear-warp volume renderer is rendering the same sub-volume.

Chapter 6

Conclusion

During this Ph.D. thesis we have covered many aspects of how to design a generic 3D computer graphics architecture with scalability in mind. Furthermore, we have explored some of the possible options for implementation of the graphics architecture in both hardware and software. In the following a summary of the work will be presented, along with suggestions for future research.

6.1 Summary

In chapter two, we presented the background for parallel 3D computer graphics architectures with a special focus on scalability. State of the art in current scalable commercial rendering architectures were discussed. From the available research it seems that a combination of parallel rendering techniques is a good method for achieving scalability. We shape the Hybris graphics architecture around a primarily sort-middle architecture based on image-parallel subdivision of the screen into many small square tiles mapped to virtual local framebuffers. For each tile, bucket sorting and buffering of work is used to load balance the jobs across virtual processors, each optimized for rendering one small square tile. In addition a partial sort-first architecture using object-parallel subdivision of the 3D model input data looks promising. The input data is split into many small sub-objects to distribute work over several geometry processors while maintaining data coherence. Finally sort-last is used to assemble the final image from tiles. Image composition of overlapping tiles might be useful in order to allow the architecture to scale even further, if correct handling of transparency is not an issue.

In chapter three, we designed an architecture based on the observations made in chapter two. This architecture is described at an abstraction level slightly above the possible implementations. The potentially available parallelism of the architecture

has also been described independently of any actual implementation. We also presented an in-depth view of the concepts involved in the design of the Hybris graphics architecture. The graphics architecture has taken form as an object-parallel and image-parallel architecture utilizing partitioned triangle meshes in the front-end graphics pipeline and tile-based rendering in the back-end graphics pipeline. Further, several viable extensions for the architecture have been discussed. These extensions include antialiasing and texture mapping. Additionally some possible sub-tile interleaved pixel parallel architectures for the tile rendering back-end have been presented.

In chapter four, we examined some of the possible hardware/software codesign implementation options for the Hybris graphics architecture, which was presented in the previous chapters. We have examined how virtual prototyping codesign using the C programming language allows the Hybris graphics architecture to be mapped to different implementation technologies, ranging from sequential software and parallel software for general purpose computers with one or more CPUs, over to more elaborate hardware/software codesign implementations. ASIC and FPGA implementations of the back-end tile rendering pipeline have been successfully implemented, although some work still remains in order to get superior performance from the FPGA hardware implementation.

Finally, chapter five gave an introduction to graphics interface APIs in general. VRML was introduced to provide a high level of abstraction to allow the static and dynamic optimizations used in the Hybris graphics architecture. Additionally an overview of volume rendering was presented in relation to the surface rendering techniques used in the Hybris graphics architecture. As an example of a useful application of the graphics architecture, the working prototype 3D-Med medical visualization workstation was presented.

In conclusion to this work, it can be said that the virtual prototyping C programming language based codesign method has proved to be very successful in terms of portability. We have demonstrated how the virtual prototype can be transformed into various hardware and software implementations, using a series of successive (but manual) transformations to match a given target architecture closely enough to allow C compilers and VHDL logic synthesis tools to handle the final design steps automatically.

Finally the performance of the Hybris implementations has proved to be very good, in many cases out-performing commercial 3D graphics processors such as the Nvidia GeForce 2. The dual CPU implementation demonstrates some of the scalability inherent in parallel implementations of the Hybris graphics architecture. Additionally the dual CPU implementation demonstrates the best performance (about 4 Mtriangles/s) currently achieved by a working implementation of Hybris.

6.2 Future work

As the Hybris graphics architecture is designed at a portable implementation independent level, there are many possible future implementation options for the architecture. In addition, the architecture itself can be extended in many ways to allow more advanced types of rendering.

Currently, one of the most interesting options for future work is an extension of the FPGA implementation to include the interleaved pixel parallel architectures described in chapter 3. As mentioned in chapter 4, this extension can very likely be implemented for the current FPGA prototyping platform, as the FPGA is not yet fully utilized. As as extension to this pixel parallel implementation, supersampling anti-aliasing using a four pixel box filter should be relatively straightforward to implement in the FPGA. Further, the sparse supersampling techniques should be investigated. Future FPGA prototyping platforms might allow implementation of parallel tile rendering configurations, as well as implementation of the front-end and SDRAM interfaces for the triangle heap.

Furthermore it is believed that the FPGA implementation can be improved enough to outperform as least the single CPU software implementation, up to the upper limit imposed by the maximum bandwidth of the PCI-bus. By using 64 bytes per triangle transferred to the back-end tile renderer, we can transfer up to 1.2 million triangles/s over the PCI-bus, assuming a bandwidth of 80 Mbytes/s is sustainable. Because of back-face culling and triangle rounding, this translates into an application rendering performance 2-4 times higher, depending on the object.

In order to allow proper handling of order-independent transparency, per-pixel fragment sorting should be investigated as this seems to be a promising future area of research. Current graphics architectures do not in general implement fragment sorting and relies on the application to manage transparency depth sorting prior to rendering.

The parallel software implementation of Hybris is another area for future research, as it would be very interesting to see how the Hybris architecture performs on larger multiprocessing platforms than the current dual Pentium III PC. The current prospects for scalable performance looks very promising, provided that a suitable multiprocessing platform with a crossbar switched memory architecture is available. Furthermore, a future area of research would be to construct a performance estimation model for such multiprocessor implementations. A highly parallel software implementation has the potential to out-perform most hardware graphics processors. In relation to software implementations, it would be interesting to utilize recent vector processor extensions such as the Pentium IV's SSE-2 extensions.

Bibliography

[1] 3Dlabs. "Wildcat: New Parascale Architecture". Technical report, 3Dlabs, 2001. http://www.3dlabs.com/product/technology/parascal.htm.

[2] ACR-NEMA. *DICOM: Digital Imaging and Communications in Medicine*, 2000. Specifications and Working Groups, `http://medical.nema.org`.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[4] K. Akeley and T. Jermoluk. "High-Performance Polygon Rendering". *SIGGRAPH Proceedings*, pages 239–246, August 1988.

[5] Kurt Akeley. "RealityEngine Graphics". *SIGGRAPH Proceedings*, pages 109–116, August 1993.

[6] Keith Andrews, Andreas Pesendorfer, Michael Pichler, Karl Heinz Wagenbrunn, and Josef Wolte. "Looking Inside VRwave: The Architecture and Interface of the VRwave VRML97 browser". *Virtual Reality Modeling Language Symposium*, pages 77–82, February 1998.

[7] ANSI/IEEE. *IEEE Standard for Binary Floating-Point Arithmetic*, 1985. ANSI/IEEE Standard 754-1985.

[8] Jean-Francis Balaguer and Enrico Gobbetti. "i3D: A High-Speed 3D Web Browser". *Virtual Reality Modeling Language Symposium*, pages 69–76, 1995. `http://www.crs4.it/~3diadm`.

[9] B. Barenbrug, F. J. Peters, and C. W. A. M. Van Overveld. "Algorithms for Division Free Perspective Correct Rendering". *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 7–13, August 2000.

[10] Anthony C. Barkans. "High Quality Rendering Using the Talisman Architecture". *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 79–88, August 1997.

[11] Kristof Beets and Dave Barron. "Super-Sampling Anti-Aliasing Analyzed". *Beyond3D & 3dfx*, 2000. `http://www.beyond3d.com`.

[12] Johannes Behr and Marc Alexa. "Volume Visualization in VRML". *Virtual Reality Modeling Language Symposium*, pages 23–27, 2001.

[13] Phil Bernosky and Scott Tandy. "Bringing Workstation Graphics Performance to a Desktop Near You: ViRGE VX". In *Hot Chips 8, A Symposium on High-Performance Chips*. S3 Inc., August 1996. `http://www.hotchips.org`.

[14] Gary Bishop and David M. Weimer. "Fast Phong Shading". *SIGGRAPH Proceedings*, pages 103–106, August 1986.

[15] Lars Bishop, Dave Eberly, Turner Whitted, Mark Finch, and Michael Shantz. "Designing a PC Game Engine". *IEEE Computer Graphics and Applications*, pages 46–53, January/February 1998.

[16] Ingmar Bitter and Arie Kaufman. "A Ray-Slice-Sweep Volume Rendering Engine". *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 121–130, August 1997.

[17] James F. Blinn. "Jim Blinn's Corner: A Ghost in a Snowstorm". *IEEE Computer Graphics and Applications*, pages 79–84, January/February 1998.

[18] James F. Blinn. "Jim Blinn's Corner: W Pleasure, W Fun". *IEEE Computer Graphics and Applications*, pages 78–82, May/June 1998.

[19] Jim Blinn. *Jim Blinn's Corner: A Trip Down the Graphics Pipeline*. Morgan Kaufmann Publishers, 1996.

[20] Jim Blinn. *Jim Blinn's Corner: Dirty Pixels*. Morgan Kaufmann Publishers, 1998.

[21] Alexander Bogomjakov and Craig Gotsman. "Universal Rendering Sequences for Transparent Vertex Caching of Progressive Meshes". *Proceedings of Graphics Interface*, June 2001.

[22] Georges-Pierre Bonneau and Alexandre Gerussi. "Level of Detail Visualization of Scalar Data Sets on Irregular Surface Meshes". *Visualization*, pages 73–77, October 1998.

[23] Kellogg S. Booth, David R. Forsey, and Allan W. Paeth. “Hardware Assistance for Z-Buffer Visible Surface Algorithms”. *IEEE Computer Graphics and Applications*, pages 31–39, November 1986.

[24] Matthew Bowen. *Handel-C Language Reference Manual*. Embedded Solutions Limited, 1998. Version 2.1.

[25] Matthew Bowen. *RC1000-PP Software User Guide*. Embedded Solutions Limited, 1998. Version 1.10.

[26] Stephen Brown and Jonathan Rose. “Architecture of FPGAs and CPLDs: A Tutorial”. Technical report, Department of Electrical and Computer Engineering, University of Toronto, 1997.

[27] Ian Buck, Greg Humphreys, and Pat Hanrahan. “Tracking Graphics State for Networked Rendering”. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 87–95, August 2000.

[28] Rikk Carey, Gavin Bell, and Chris Marrin. *VRML97, The Virtual Reality Modeling Language*. VRML Consortium, `http://www.vrml.org/technicalinfo/specifications/vrml97`, 1997. International Standard ISO/IEC 14772-1:1997.

[29] Loren Carpenter. “The A-Buffer, an Antialiased Hidden Surface Method.”. *SIGGRAPH Proceedings*, pages 103–108, July 1984.

[30] Milton Chen, Gordon Stoll, Homan Igehy, Kekoa Proudfoot, and Pat Hanrahan. “Simple Models of the Impact of Overlap in Bucket Rendering”. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 105–112, August 1998.

[31] Tzi-Cker Chiueh. “Heresy: A Virtual Image-Space 3D Rasterization Architecture”. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 69–77, August 1997.

[32] Tzi-Cker Chiueh and Wei-Jen Lin. “Characterization of Static 3D Graphics Workloads”. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 17–24, August 1997.

[33] Mike M. Chow. “Optimized Geometry Compression for Real-Time Rendering”. *IEEE Visualization ’97*, 1997.

[34] Compaq. “PCI-X: An Evolution of the PCI Bus”. Technical report, Compaq Computer Corporation, September 1999.

[35] Robert L. Cook, Loren Carpenter, and Edwin Catmull. “The Reyes Image Rendering Architecture”. *SIGGRAPH Proceedings*, pages 95–102, July 1987.

[36] Michael Cox and Narendra Bhandari. “Architectural Implications of Hardware-Accelerated Bucket Rendering on the PC”. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 25–34, 1997.

[37] Thomas W. Crockett. “Parallel Rendering”. *Encyclopedia of Computer Science and Technology*, 34:335–371, 1996.

[38] Thomas W. Crockett and Tobias Orloff. “A MIMD Rendering Algorithm for Distributed Memory Architectures”. *Symposium on Parallel Rendering*, pages 35–42, November 1993.

[39] Balázs Csébfalvi, Andreas König, and Eduard Gröller. “Fast Surface Rendering of Volumetric Data”. *WSCG’2000, The 8th International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media*, February 2000.

[40] Brian Curless and Marc Levoy. “A Volumetric Method for Building Complex Models from Range Images”. *SIGGRAPH Proceedings*, pages 303–312, August 1996.

[41] Nell Dale and Susan C. Lilly. *Pascal Plus Data Structures, Algorithms, and Advanced Programming*. D. C. Heath and Company, second edition, 1988.

[42] Michael Deering. “High Resolution Virtual Reality”. *SIGGRAPH Proceedings*, pages 195–202, July 1992.

[43] Michael Deering. “Geometry Compression”. *SIGGRAPH Proceedings*, pages 13–20, August 1995.

[44] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. “The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics”. *SIGGRAPH Proceedings*, pages 21–30, August 1988.

[45] Michael F. Deering and Scott R. Nelson. “Leo: A System for Cost Effective 3D Shaded Graphics”. *SIGGRAPH Proceedings*, pages 101–108, August 1993.

[46] Michael F. Deering, Stephen A. Schlapp, and Michael G. Lavelle. "FBRAM: A New Form of Memory Optimized for 3D Graphics". *SIGGRAPH Proceedings*, pages 167–174, July 1994.

[47] Paul J. Diefenbach, Prakash Mahesh, and Daniel Hunt. "Building OpenWorlds". *Virtual Reality Modeling Language Symposium*, pages 33–38, February 1998.

[48] Paul Joseph Diefenbach. *Pipeline Rendering: Interaction and Realism through Hardware-based Multi-pass Rendering*. PhD thesis, Computer and Information Science, University of Pennsylvania, 1996.

[49] Tom Duff. "Compositing 3-D Rendered Images". *SIGGRAPH Proceedings*, pages 41–44, July 1985.

[50] N. Egund and J. Palmer. "Femoral Anatomy Described in Cylindrical Coordinates Using Computed Tomography". *Acta Radiologica Diagnosis*, 25(Facs. 3), 1984.

[51] Matthew Eldridge, Homan Igehy, and Pat Hanrahan. "Pomegranate: A Fully Scalable Graphics Architecture". *SIGGRAPH Proceedings*, pages 443–454, July 2000.

[52] David Ellsworth. "A Multicomputer Polygon Rendering Algorithm for Interactive Applications". *Symposium on Parallel Rendering*, pages 43–48, October 1993.

[53] David A. Ellsworth. "A New Algorithm for Interactive Graphics on Multicomputers". *IEEE Computer Graphics and Applications*, pages 33–40, July 1994.

[54] David Allan Ellsworth. *Polygon Rendering for Interactive Visualization on Multicomputers*. PhD thesis, Computer Science, University of North Carolina at Chapel Hill, 1996.

[55] Evans & Sutherland Computer Corporation. *Freedom 3000 Technical Overview*, October 1992.

[56] Jon P. Ewins, Phil L. Watten, Martin White, Michael D. J. McNeill, and Paul F. Lister. "Codesign of Graphics Hardware Accelerators". *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 103–110, August 1997.

[57] John Eyles, Steven Molnar, John Poulton, Trey Greer, Anselmo Lastra, Nick England, and Lee Westover. "PixelFlow: The Realization". *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 57–68, August 1997.

[58] D. Field. "Incremental Linear Interpolation". *ACM Transactions on Graphics*, 4(1):1–11, January 1985.

[59] Alessandro Fin, Franco Fummi, Maurizio Martignano, and Mirko Signoretto. "SystemC: A Homogeneous Environment to Test Embedded Systems". *CODES Symposium on Hardware/Software Codesign*, pages 17–22, April 2001.

[60] David Flanagan. *Java Foundation Classes, A Desktop Quick Reference*. O'Reilly & Associates, Inc., first edition, September 1999.

[61] David Flanagan. *Java in a Nutshell, A Desktop Quick Reference*. O'Reilly & Associates, Inc., third edition, November 1999.

[62] Martin Fleury and Andrew Downton. *Pipelined Processor Farms: Structured Design for Embedded Parallel Systems*. John Wiley and Sons, Inc., April 2001.

[63] Foley, van Dam, Feiner, and Hughes. *Computer Graphics, Principles and Practice*. Addison-Wesley, second edition, 1990.

[64] Henry Fuchs, Jack Goldfeather, Jeff P. Hultquist, Susan Spach, John D. Austin, Frederick P. Brooks, John G. Eyles, and John Poulton. "Fast Spheres, Shadows, Textures, Transparencies and Image Enhancements in Pixel-Planes". *SIGGRAPH Proceedings*, pages 111–120, 1985.

[65] Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories". *SIGGRAPH Proceedings*, pages 79–88, July 1989.

[66] Daniel D. Gajski and Frank Vahid. "Specification and Design of Embedded Hardware-Software Systems". *IEEE Design and Test of Computers*, pages 53–67, Spring 1995.

[67] Allen Van Gelder and Kwansik Kim. "Direct Volume Rendering with Shading Via Three-Dimensional Textures". *Symposium on Volume Visualization*, pages 23–30, October 1996.

[68] Nader Gharachorloo, Satish Gupta, Erdem Hokenek, Peruvemba Balasubramanian, Bill Bogholtz, Christian Mathieu, and Christos Zoulas. "Subnanosecond Pixel Rendering with Million Transistor Chips". *SIGGRAPH Proceedings*, pages 41–49, August 1988.

[69] Nader Gharachorloo and Robert F. Sproull. "A Characterization of Ten Rasterization Techniques". *SIGGRAPH Proceedings*, pages 355–368, July 1989.

[70] Peter N. Glaskowsky. "MoSys Explains 1T-SRAM Technology, Unique Architecture Hides Refresh, Makes DRAM Work Like SRAM". *Microprocessor Report*, 13(12), September 1999.

[71] Thomas Gleerup. "ASIC for 3D Graphics Pipeline Back-End". Master's thesis, Information Technology, Technical University of Denmark, January 1999.

[72] Thomas Gleerup, Hans Holten-Lund, Jan Madsen, and Steen Pedersen. "Memory Architecture for Efficient Utilization of SDRAM: A Case Study of the Computation/Memory Access Trade-Off". *CODES 2000 Workshop on Hardware/Software Codesign*, pages 51–55, May 2000.

[73] H. Gouraud. "Continuous Shading of Curved Surfaces". *IEEE Transactions on Computers*, C-20(6):623–629, June 1971.

[74] Eric Grant, Phil Amburn, and Turner Whitted. "Exploiting Classes in Modeling and Display Software". *IEEE Computer Graphics and Applications*, pages 13–20, November 1986.

[75] Ned Greene, Michael Kass, and Gavin Miller. "Hierarchical Z-Buffer Visibility". *SIGGRAPH Proceedings*, pages 231–238, August 1993.

[76] Jesper N. R. Grode. *Component Modeling and Performance Estimation in Hardware/Software Codesign*. PhD thesis, Information Technology, Technical University of Denmark, March 1999.

[77] J. P. Grossman and William J. Dally. "Point Sample Rendering". In *Rendering Techniques '98*, pages 181–192, Vienna, Austria, June 1998. Proceedings of the 9th Eurographics Workshop on Rendering, Springer-Verlag.

[78] Rajesh K. Gupta and Stan Y. Liao. "Using a Programming Language for Digital System Design". *IEEE Design and Test of Computers*, pages 72–80, April–June 1997.

[79] Paul Haeberli and Kurt Akeley. "The Accumulation Buffer: Hardware Support for High-Quality Rendering". *SIGGRAPH Proceedings*, pages 309–318, August 1990.

[80] Ziyad S. Hakura and Anoop Gupta. "The Design and Analysis of a Cache Architecture for Texture Mapping". *24th International Symposium on Computer Architecture*, 1997.

[81] Chandlee B. Harrell and Farhad Fouladi. "Graphics Rendering Architecture for a High Performance Desktop Workstation". *SIGGRAPH Proceedings*, pages 93–100, August 1993.

[82] Paul S. Heckbert. "Survey of Texture Mapping". *IEEE Computer Graphics and Applications*, pages 56–67, November 1986.

[83] K. L. Hermann and N. Egund. "CT Measurement of Anteversion in the Femoral Neck.". *Acta Radiologica*, 1997.

[84] Hans Holten-Lund. "Fast Rendering Techniques for Real-Time 3D Image Synthesis in an Interactive Environment". Master's thesis, Information Technology, Technical University of Denmark, August 1995.

[85] Hans Holten-Lund. "Implementation of a Parallel 3D Graphics Engine". Technical report, Information Technology, Technical University of Denmark, 1999.

[86] Hans Holten-Lund, Mogens Hvidtfeldt, Jan Madsen, and Steen Pedersen. "VRML Visualization in a Surgery Planning and Diagnostics Application". *Web3D-VRML 2000 Fifth Symposium on the Virtual Reality Modeling Language*, pages 111–118, February 2000.

[87] Hans Holten-Lund and Jacob Lildballe. "Raytracing På Transputere". Technical report, Grafisk Kommunikation, DTU, February 1995.

[88] Hans Holten-Lund, Martin Lütken, Jan Madsen, and Steen Pedersen. "Virtual Prototyping, a Case Study in Dataflow Oriented Codesign". *NORCHIP '98 Proceedings*, pages 222–229, November 1998.

[89] Hans Holten-Lund, Jan Madsen, and Steen Pedersen. "A Case Study of a Hybrid Parallel 3D Surface Rendering Graphics Architecture". *SASIMI '97 Workshop on Synthesis and System Integration of Mixed Technologies*, pages 149–154, December 1997.

[90] Hugues Hoppe. "Progressive Meshes". *SIGGRAPH Proceedings*, pages 99–108, August 1996.

[91] Hugues Hoppe. "Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering". *Visualization*, pages 35–42, October 1998.

[92] Hugues Hoppe. "Optimization of Mesh Locality for Transparent Vertex Caching". *SIGGRAPH Proceedings*, pages 269–276, August 1999.

[93] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. "Mesh Optimization". *SIGGRAPH Proceedings*, pages 19–26, August 1993.

[94] Greg Humphreys, Ian Buck, Matthew Eldridge, and Pat Hanrahan. "Distributed Rendering for Scalable Displays". *Proceedings of Supercomputing*, 2000.

[95] Greg Humphreys and Pat Hanrahan. "A Distributed Graphics System for Large Tiled Displays". *IEEE Visualization*, pages 215–224, 1999.

[96] Tobias Hüttner and Wolfgang Straßer. "Fast Footprint MIPmapping". *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 35–44, August 1999.

[97] Mogens Hvidtfeldt. "Handling of Surface Models in 3D Medical Applications". Master's thesis, Information Technology, Technical University of Denmark, March 1998.

[98] Kai Hwang. *Computer Arithmetic, Principles, Architecture and Design*. John Wiley and Sons, Inc., 1979.

[99] Homan Igehy. *Scalable Graphics Architectures: Interface & Texture*. PhD thesis, Computer Science, Stanford University, May 2000.

[100] Homan Igehy, Matthew Eldridge, and Pat Hanrahan. "Parallel Texture Caching". *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 95–106, August 1999.

[101] Homan Igehy, Gordon Stoll, and Pat Hanrahan. "The Design of a Parallel Graphics Interface". *SIGGRAPH Proceedings*, pages 141–150, July 1998.

[102] Tsuneo Ikedo and Jianhua Ma. "The Truga001: A Scalable Rendering Processor". *IEEE Computer Graphics and Applications*, pages 59–79, March/April 1998.

[103] Intel Corporation. *Intel Architecture Software Developer´s Manual, Volume 3: System Programming Guide*, 1997. Order number: 243192.

[104] Intel Corporation. *Accelerated Graphics Port Interface Specification*, May 1998. Revision 2.0.

[105] Intel Corporation. *A.G.P. Pro Specification*, July 1998. Revision 0.9.

[106] Intel Corporation. *Intel Architecture Optimization Reference Manual*, 1999. Order number: 730795-001.

[107] Intel Corporation. *Intel 815 Chipset: Graphics Controller. Programmer's Reference Manual (PRM)*, July 2000.

[108] Dan C. R. Jensen, Jan Madsen, and Steen Pedersen. "The Importance of Interfaces: A HW/SW Codesign Case Study". *CODES/CASHE '97 Workshop on Hardware/Software Codesign*, pages 87–91, March 1997.

[109] Kurt Jensen. "Volume Rendering Using VLSI". Master's thesis, Information Technology, Technical University of Denmark, February 1997.

[110] Norman P. Jouppi and Chun-Fa Chang. "Z^3: An Economical Hardware Technique for High-Quality Antialiasing and Transparency". *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 85–93, August 1999.

[111] Zachi Karni and Craig Gotsman. "Spectral Coding of Mesh Geometry". *SIGGRAPH Proceedings*, pages 279–286, July 2000.

[112] Zachi Karni and Craig Gotsman. "3D Mesh Compression Using Fixed Spectral Bases". *Proceedings of Graphics Interface*, June 2001.

[113] G. Karypis and V. Kumar. *MeTiS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices. Version 4.0*. Computer Science, University of Minnesota, September 1998. http://www-users.cs.umn.edu/~karypis/metis/.

[114] George Karypis and Vipin Kumar. "Multilevel Algorithms for Multi-Constraint Graph Partitioning". Technical Report 98-019, University of Minnesota, Department of Computer Science / Army HPC Research Center, May 1998. http://www.cs.umn.edu/~karypis.

[115] Michael Kelley, Kirk Gould, Brent Pease, Stephanie Winner, and Alex Yen. "Hardware Accelerated Rendering of CSG and Transparency". *SIGGRAPH Proceedings*, pages 177–184, July 1994.

[116] Michael Kelley, Stephanie Winner, and Kirk Gould. "A Scalable Hardware Render Accelerator using a Modified Scanline Algorithm". *SIGGRAPH Proceedings*, pages 241–248, July 1992.

[117] Ken Kennedy and Kathryn S. McKinley. "Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution". In *Languages and Compilers for Parallel Computing*, pages 301–321. Springer-Verlag, 1993.

[118] Mark J. Kilgard. "Realizing OpenGL: Two Implementations of One Architecture". *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 45–55, August 1997.

[119] Peter Voigt Knudsen. *Techniques for Co-Synthesis*. PhD thesis, Information Technology, Technical University of Denmark, March 1999.

[120] Kevin Kreeger and Arie Kaufman. "Hybrid Volume and Polygon Rendering with Cube Hardware". *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 15–24, August 1999.

[121] Kubota Pacific Computer Inc. *Denali Technical Overview*, March 1993.

[122] Philippe Lacroute and Marc Levoy. "Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation". *SIGGRAPH Proceedings*, pages 451–458, July 1994.

[123] L. Casey Larijani. *The Virtual Reality Primer*. McGraw-Hill, 1994.

[124] Jin-Aeon Lee and Lee-Sup Kim. "Single-Pass Full-Screen Hardware Accelerated Antialiasing". *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 67–75, August 2000.

[125] Jin-Aeon Lee and Lee-Sup Kim. "SPARP: A Single Pass Antialiased Rasterization Processor". *Computers & Graphics*, 24:233–243, 2000.

[126] Wingyu Leung, Fu-Chieh Hsu, and Mark-Eric Jones. "The Ideal SoC Memory: 1T-SRAM™". Technical report, MoSys Inc., 2000. `http://www.mosys.com`.

[127] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan Shade, and Duane Fulk. "The Digital Michelangelo Project: 3D Scanning of Large Statues". *SIGGRAPH Proceedings*, pages 131–144, July 2000.

[128] Barthold Lichtenbelt. "Design of a High Performance Volume Visualization System". *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 111–119, August 1997.

[129] William E. Lorensen and Harvey E. Cline. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm". *SIGGRAPH Proceedings*, pages 163–169, July 1987.

[130] Martin Lütken. "Hardware implementation of 3D graphics pipeline". Master's thesis, Information Technology, Technical University of Denmark, August 1997.

[131] Jakob Winther Madsen. "VLSI Implementering af Volumenbilledvisualisering". Master's thesis, Information Technology, Technical University of Denmark, February 1998.

[132] Jan Madsen and Jens P. Brage. "Codesign Analysis of a Computer Graphics Application". *Design Automation for Embedded Systems*, 1(1-2):121–145, January 1996. Kluwer Academic Publishers.

[133] Jan Madsen, Jesper Grode, Peter Knudsen, Morten E. Pedersen, and Anne Haxthausen. "LYCOS: The Lyngby Co-Synthesis System". *Design Automation for Embedded Systems*, 2(2):195–235, 1997.

[134] Abraham Mammen. "Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique". *IEEE Computer Graphics and Applications*, pages 43–55, July 1989.

[135] Chris Marrin. *External Authoring Interface Reference*. Silicon Graphics Inc., November 1997. `http://www.web3d.org/WorkingGroups/vrml-eai/history/eai_draft.html`.

[136] Brian McClendon and John Montrym. "InfiniteReality™ Graphics - Power Through Complexity". In *Hot Chips 8, A Symposium on High-Performance Chips*. Silicon Graphics, August 1996. `http://www.hotchips.org`.

[137] Ray McConnell. “Massively Parallel Computing on the FUZION Chip”. In *Invited Speaker at SIGGRAPH/Eurographics Workshop on Graphics Hardware*. PixelFusion Ltd, August 1999.

[138] Michael D. McCool and Wolfgang Heidrich. “Texture Shaders”. *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 117–126, August 1999.

[139] Joel McCormack and Robert McNamara. “Tiled Polygon Traversal Using Half-Plane Edge Functions”. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 15–21, August 2000.

[140] Joel McCormack, Robert McNamara, Christopher Gianos, Larry Seiler, Norman P. Jouppi, and Ken Correll. “Neon: A Singe-Chip 3D Workstation Graphics Accelerator”. *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 123–132, August 1998.

[141] Joel McCormack, Robert McNamara, Christopher Gianos, Larry Seiler, Norman P. Jouppi, Ken Correll, Todd Dutton, and John Zurawski. “Neon: A (Big) (Fast) Singe-Chip 3D Workstation Graphics Accelerator”. *Compaq WRL Research Report 98/1*, July 1999.

[142] Joel McCormack, Ronald Perry, Keith I. Farkas, and Norman P. Jouppi. “Feline: Fast Elliptical Lines for Anisotropic Texture Mapping”. *SIGGRAPH Proceedings*, pages 243–250, August 1999.

[143] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. “Improving Data Locality with Loop Transformations”. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[144] Robert McNamara, Joel McCormack, and Norman P. Jouppi. “Prefiltered Antialiased Lines Using Half-Plane Distance Functions”. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 77–85, August 2000.

[145] Nimrod Megiddo and Vivek Sarkar. “Optimal Weighted Loop Fusion for Parallel Programs”. *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1997.

[146] Michael Meißner, Ulrich Hoffmann, and Wolfgang Straßer. “Enabling Classification and Shading for 3D Texture Mapping Based Volume Rendering Using OpenGL and Extensions”. *IEEE Visualization ’99 Proc.*, 1999.

[147] Micron Technology Inc. *Synchronous DRAM Data Sheet, 64Mb SDRAM*, 1998. rev. 10/98.

[148] Microsoft. *DirectX 8.0 SDK*, 2000. `http://www.microsoft.com/directx`.

[149] David Anthony Paul Mitchell. *Fast Algorithms and Hardware for 3D Computer Graphics*. PhD thesis, Computer Science, Sheffield University, July 1990.

[150] Tulika Mitra and Tzi-Cker Chiueh. "Dynamic 3D Graphics Workload Characterization and the Architectural Implications". *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture on MICRO-32*, pages 62–71, November 1999.

[151] Søren A. Møller. "Memory and Data Structures in 3D Medical PC-Workstation.". Master's thesis, Information Technology, Technical University of Denmark, August 1997.

[152] Tomas Möller and Eric Haines. *Real-Time Rendering*. A K Peters, 1999.

[153] Steven Molnar. "Combining Z-Buffer Engines for Higher-Speed Rendering". In *Advances in Computer Graphics Hardware III*, pages 171–182. Proceedings of the 1988 Eurographics Workshop on Graphics Hardware, Eurographics Seminars, 1988.

[154] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. "A Sorting Classification of Parallel Rendering". *IEEE Computer Graphics and Applications*, 14(4):23–31, July 1994.

[155] Steven Molnar, John Eyles, and John Poulton. "PixelFlow: High-Speed Rendering Using Image Composition". *SIGGRAPH Proceedings*, pages 231–240, July 1992.

[156] Steven Edward Molnar. *Image-Composition Architectures for Real-Time Image Generation*. PhD thesis, Computer Science, University of North Carolina at Chapel Hill, 1991.

[157] C. Montani, R. Scateni, and R. Scopigno. "Discretized Marching Cubes". *IEEE Visualization*, pages 281–287, 1994.

[158] John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal. "InfiniteReality: A Real-Time Graphics System". *SIGGRAPH Proceedings*, pages 293–302, August 1997.

[159] Steve Morein. "ATI Radeon HyperZ Technology". In *Invited Speaker at SIGGRAPH/Eurographics Workshop on Graphics Hardware*. ATI, August 2000.

[160] Lars Bo Mortensen. "VLSI Til Generering Af 3D-Overflademodel". Master's thesis, Information Technology, Technical University of Denmark, February 1997.

[161] Carl Mueller. "The Sort-First Rendering Architecture for High-Performance Graphics". *Symposium on Interactive 3D Graphics*, 1995.

[162] Carl Mueller. "Hierarchical Graphics Databases in Sort-First". *IEEE Symposium on Parallel Rendering*, pages 49–57, 1997.

[163] S. B. Murphy, P. K. Kijewski, M. B. Millis, and A. Harless. "Acetabular Dysplasia in the Adolescent and Young Adult". *Clinical Orthopaedics and Related Research, (261)*, December 1990.

[164] Zainalabedin Navabi. *VHDL Analysis and Modeling of Digital Systems*. McGraw-Hill, Inc., 1993.

[165] J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide*. Addison-Wesley, 1993.

[166] L. T. Nguyen et al. "Multimedia Signal Processor (MSP) Summary". In *Hot Chips 8, A Symposium on High-Performance Chips*. Samsung, August 1996. `http://www.hotchips.org`.

[167] Satoshi Nishimura and Tosiyasu L. Kunii. "VC-1: A Scalable Graphics Computer with Virtual Local Frame Buffers". *SIGGRAPH Proceedings*, pages 365–372, August 1996.

[168] Petri Nordlund. "Glaze3D™". In *Invited Speaker at SIGGRAPH/Eurographics Workshop on Graphics Hardware*. Bitboys Oy, August 1999.

[169] Marc Olano and Trey Greer. "Triangle Scan Conversion Using 2D Homogeneous Coordinates". *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 89–95, August 1997.

[170] Manuel M. Oliveira, Gary Bishop, and David McAllister. "Relief Texture Mapping". *SIGGRAPH Proceedings*, pages 359–368, July 2000.

[171] Rändy Osborne, Hanspeter Pfister, Hugh Lauer, TakaHide Ohkami, Neil McKenzie, Sarah Gibson, and Wally Hiatt. "EM-Cube: An Architecture for Low-Cost Real-Time Volume Rendering". *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 131–138, August 1997.

[172] John D. Owens, William J. Dally, Ujval J. Kapasi, Scott Rixner, Peter Mattson, and Ben Mowery. "Polygon Rendering on a Stream Architecture". *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 23–32, August 2000.

[173] Renato Pajarola. "Large Scale Terrain Visualization Using the Restricted Quadtree Triangulation". *Visualization*, pages 19–26, October 1998.

[174] David A. Patterson and John L. Hennessy. *Computer Organization & Design, The Hardware/Software Interface*. Morgan Kaufmann Publishers, 1994.

[175] PCI Special Interest Group. *PCI Local Bus Specification*, June 1995. Revision 2.1.

[176] Torben Yde Pedersen. "Implementering Af CPU Kort Til PCI Bussen". Master's thesis, Information Technology, Technical University of Denmark, April 1997.

[177] Ken Perlin. "An Image Synthesizer". *SIGGRAPH Proceedings*, pages 287–296, July 1985.

[178] Kenneth Haldbæk Petersen. "Analyse Og Design Af PCI-Bus Kort". Master's thesis, Information Technology, Technical University of Denmark, September 1996.

[179] H. Pfister and A. Kaufman. "Cube-4 – A Scalable Architecture for Real-Time Volume Rendering". *Volume Visualization Symposium*, pages 47–54, October 1996.

[180] Hanspeter Pfister, Jan Hardenbergh, Jim Knittel, Hugh Lauer, and Larry Seiler. "The VolumePro Real-Time Ray-Casting System". *SIGGRAPH Proceedings*, pages 251–260, August 1999.

[181] Hanspeter Pfister, Matthias Zwicker, Jeroen Van Baar, and Markus Gross. "Surfels: Surface Elements as Rendering Primitives". *SIGGRAPH Proceedings*, pages 335–342, July 2000.

[182] Bui Tuong Phong. “Illumination for Computer Generated Pictures”. *Communications of the ACM*, 18(6):311–317, June 1975.

[183] Michael Pichler, Gerbert Orasche, Keith Andrews, Ed Grossman, and Mark McCahill. “VRweb: A Multi-System VRML Viewer”. *Virtual Reality Modeling Language Symposium*, pages 77–85, 1995.

[184] J. Pineda. “A Parallel Algorithm for Polygon Rasterization”. *SIGGRAPH Proceedings*, pages 17–20, August 1988.

[185] PLX Technology, `http://www.plxtech.com`. *PLX PCI 9080 Data Book*, January 2000. Version 1.06.

[186] Curtis R. Priem. “Developing the GX Graphics Accelerator Architecture”. *IEEE Micro*, pages 44–54, February 1990.

[187] Scott Pritchett. “Giga3D Architectural Advantages”. `http://www.gigapixel.com`, GigaPixel, November 1999.

[188] “PowerVR”. `http://www.powervr.com`, 1996. NEC/VideoLogic (today: STMicroelectronics and Imagination Technologies).

[189] Jan Dueholm Rasmussen. “Implementering Af PCI-Kort Med FPGA-Kredse”. Master’s thesis, Information Technology, Technical University of Denmark, April 1997.

[190] Harvey Ray and Deborah Silver. “The RACE II Engine for Real-Time Volume Rendering”. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 129–136, August 2000.

[191] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. “Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage Rasterization”. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 109–118, August 2000.

[192] John Rohlf and James Helman. “IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics”. *SIGGRAPH Proceedings*, pages 381–394, July 1994.

[193] Szymon Rusinkiewicz and Mark Levoy. “QSplat: A Multiresolution Point Rendering System for Large Meshes”. *SIGGRAPH Proceedings*, pages 343–352, July 2000.

[194] Rizos Sakellariou. *On the Quest for Perfect Load Balance in Loop-Based Parallel Computations*. PhD thesis, Computer Science, University of Manchester, 1998.

[195] Rudrajit Samanta, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. "Hybrid Sort-First and Sort-Last Parallel Rendering with a Cluster of PCs". *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 97–108, August 2000.

[196] Rudrajit Samanta, Jiannan Zheng, Thomas Funkhouser, Kai Li, and Jaswinder PalSingh. "Load Balancing for Multi-Projector Rendering Systems". *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 107–116, August 1999.

[197] Pedro V. Sander, Xianfeng Gu, Steven J. Gortler, Hugues Hoppe, and John Snyder. "Silhouette Clipping". *SIGGRAPH Proceedings*, pages 327–334, July 2000.

[198] Kentaro Sano, Hiroyuki Kitajima, Hiroaki Kobayashi, and Tadao Nakamura. "Parallel Processing of the Shear-Warp Factorization with the Binary-Swap Method on a Distributed-Memory Multiprocessor System". *Parallel Rendering Symposium*, pages 87–94, October 1997.

[199] Andreas Schilling. "A New Simple and Efficient Antialiasing with Subpixel Masks". *SIGGRAPH Proceedings*, pages 133–141, July 1991.

[200] Andreas Schilling and Wolfgang Straßer. "EXACT: Algorithm and Hardware Architecture for an Improved A-Buffer". *SIGGRAPH Proceedings*, pages 85–91, August 1993.

[201] Kirk Schloegel, George Karypis, and Vipin Kumar. "Graph Partitioning for High Performance Scientific Simulations". In J. Dongarra et al., editors, *To Be Included in: CRPC Parallel Computing Handbook*. Morgan Kaufmann, 2000. Available online: `http://www-users.cs.umn.edu/~karypis/publications/partitioning.html`.

[202] Kirk Schloegel, George Karypis, and Vipin Kumar. "A Unified Algorithm for Load-Balancing Adaptive Scientific Simulations". *Supercomputing 2000*, 2000.

[203] Robert Schreiber, Shail Aditya, B. Ramakrishna Rau, Vinod Kathail, Scott Mahlke, Santosh Abraham, and Greg Snider. "High-Level Synthesis of Nonprogrammable Hardware Accelerators". Technical Report HPL-2000-31,

Computer Systems and Technology Laboratory, HP Laboratories Palo Alto, May 2000.

[204] M. Segal and K. Akeley. *The OpenGL® Graphics System: A Specification*. OpenGL Architectural Review Board – ARB, 1992. `http://www.opengl.org`.

[205] Hans-Peter Seidel and Wolfgang Heidrich. “Hardware Shading: State-of-the-Art and Future Challenges”. In *Keynote Speaker at SIGGRAPH/Eurographics Workshop on Graphics Hardware*. Max-Planck-Institut für Informatik, Saarbrücken, Germany, August 2000.

[206] Bruce Shriver and Bennett Smith. *The Anatomy of a High-Performance Microprocessor, A Systems Perspective*. IEEE Computer Society, 1998.

[207] Henrik Ahrendt Sørensen. “SoC Design-Eksperimenter med en Stor FPGA”. Master’s thesis, Information Technology, Technical University of Denmark, February 2001.

[208] Stanford Computer Graphics Laboratory. *The Stanford 3D Scanning Repository*. `http://www-graphics.stanford.edu/data/3Dscanrep`.

[209] J. Staunstrup and W. Wolf, editors. *Hardware/Software Co-Design, Principles and Practice*. Kluwer Academic Publishers, 1997.

[210] Paul S. Strauss and Rikk Carey. “An Object-Oriented 3D Graphics Toolkit”. *SIGGRAPH Proceedings*, pages 341–349, July 1992.

[211] David Suey, McDonnell Douglas, David Bailey, and Thomas P. Morrissey. “PHIGS: A Standard, Dynamic, Interactive Graphics Interface”. *IEEE Computer Graphics and Applications*, pages 50–57, August 1986.

[212] Sun Microsystems. *Java 3D™ API 1.3 Specification Alpha*, 2001. `http://java.sun.com/products/java-media/3D`.

[213] Sun Microsystems. *The Source for Java™ Technology*, 2001. `http://java.sun.com`.

[214] Ivan E. Sutherland and Gary W. Hodgman. “Reentrant Polygon Clipping”. *Communications of the ACM*, 17(1):32–42, January 1974.

[215] Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. “A Characterization of Ten Hidden-Surface Algorithms”. *Computing Surveys*, 6(1):1–55, March 1974.

[216] Charles Sweeney and Bill Blyth. *RC1000-PP Hardware Reference Manual*. Embedded Solutions Limited, 1998. Version 2.1.

[217] Synopsys, Inc. *FPGA Compiler II / FPGA Express VHDL Reference Manual*, May 1999. Version 1999.05.

[218] Synopsys Inc., CoWare Inc., Frontier Design Inc. et al. *Functional Specification for SystemC 2.0*, January 2001. `http://www.systemc.org`.

[219] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. "The Clipmap: A Virtual Mipmap". *SIGGRAPH Proceedings*, pages 151–158, July 1998.

[220] Gary Tarolli. "Real Time Cinematic Effects on the PC – The 3dfx T-Buffer™". In *Invited Speaker at SIGGRAPH/Eurographics Workshop on Graphics Hardware*. 3dfx Interactive, August 1999.

[221] Gabriel Taubin and Jarek Rossignac. "Geometric Compression Through Topological Surgery". Technical report, IBM Research technical report number RC-20340, 1996.

[222] Kim Theilgaard. "Medicinsk Visualiseringssystem". Master's thesis, Information Technology, Technical University of Denmark, 1997.

[223] Jay Torborg and James T. Kajiya. "Talisman: Commodity Realtime 3D Graphics for the PC". *SIGGRAPH Proceedings*, pages 353–363, August 1996.

[224] Neil Trevett. "GLINT Gamma: A 3D Geometry and Lighting Processor for the PC". In *Hot Chips 9, A Symposium on High-Performance Chips*. 3Dlabs, August 1997. `http://www.hotchips.org`.

[225] Neil Trevett. "Challenges & Opportunities for 3D Graphics on the PC". In *Keynote Speaker at SIGGRAPH/Eurographics Workshop on Graphics Hardware*. 3Dlabs, August 1999.

[226] Steve Upstill. *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley, 1989.

[227] C. A. Valderrama, M. Romdhani, J. M. Daveau, G. Marchioro, A. Changuel, and A. A. Jerraya. "COSMOS: A Transformational Co-Design Tool". In J. Staunstrup and W. Wolf, editors, *Hardware/Software Co-Design, Principles and Practice*, pages 307–357. Kluwer Academic Publishers, 1997.

[228] Akiyoshi Wakatani and Michael Wolfe. "Effectiveness of Message Strip-Mining for Regular and Irregular Communication". *Proc. PDCS*, October 1994.

[229] Alan Watt. *3D Computer Graphics*. Addison-Wesley, third edition, 2000.

[230] Alan Watt and Mark Watt. *Advanced Animation and Rendering Techniques, Theory and Practice*. ACM Press, 1992.

[231] Henrik Weimer, Joe Warren, Jane Troutner, Wendell Wiggins, and John Shrout. "Efficient Co-Triangulation of Large Data Sets". *Visualization*, pages 119–126, 1998.

[232] Rüdiger Westermann and Thomas Ertl. "Efficiently Using Graphics Hardware in Volume Rendering Applications". *SIGGRAPH Proceedings*, pages 169–177, July 1998.

[233] Scott Whitman. "Parallel Graphics Rendering Algorithms". *Proc. 3rd Eurographics Workshop on Rendering*, pages 123–134, May 1992.

[234] Scott Whitman. "Dynamic Load Balancing for Parallel Polygon Rendering". *IEEE Computer Graphics and Applications*, pages 41–48, July 1994.

[235] Stephanie Winner, Mike Kelley, Brent Pease, Bill Rivard, and Alex Yen. "Hardware Accelerated Rendering of Antialiasing Using a Modified A-Buffer Algorithm". *SIGGRAPH Proceedings*, pages 307–316, August 1997.

[236] George Wolberg. *Digital Image Warping*. IEEE Computer Society Press Monograph, 1990.

[237] Daniel J. Woods, Alan Norton, and Gavin Bell. "Wired for Speed: Efficient Routes in VRML 2.0". *Virtual Reality Modeling Language Symposium*, pages 133–138, 1997.

[238] Xilinx, Inc. *Virtex™ 2.5V Field Programmable Gate Arrays*, May 2000. `http://www.xilinx.com/partinfo/ds003.htm`.

[239] Yong Yao. "AGP Speeds 3D Graphics". *Microprocessor Report*, 10(8), June 1996.

[240] Hansong Zhang and Kenneth E. Hoff. "Fast Backface Culling Using Normal Masks". *Symposium on Interactive 3D Graphics*, 1997.

[241] Reto Zimmermann. "VHDL Library of Arithmetic Units". *Forum on Design Languages (FDL '98)*, September 1998. Lausanne.

Appendix A

Published papers

This appendix lists the papers published during the Ph.D. study. They are listed in chronological order.

[89] Hans Holten-Lund, Jan Madsen and Steen Pedersen, "A Case Study of a Hybrid Parallel 3D Surface Rendering Graphics Architecture", SASIMI '97 Workshop on Synthesis and System Integration of Mixed Technologies, pages 149–154, December 1997.

[88] Hans Holten-Lund, Martin Lütken, Jan Madsen and Steen Pedersen, "Virtual Prototyping, a Case Study in Dataflow Oriented Codesign", NORCHIP '98 Proceedings, pages 222–229, November 1998.

[86] Hans Holten-Lund, Mogens Hvidtfeldt, Jan Madsen and Steen Pedersen, "VRML Visualization in a Surgery Planning and Diagnostics Application", Web3D-VRML 2000 Fifth Symposium on the Virtual Reality Modeling Language, pages 111–118, February 2000.

[72] Thomas Gleerup, Hans Holten-Lund, Jan Madsen and Steen Pedersen, "Memory Architecture for Efficient Utilization of SDRAM: A Case Study of the Computation/Memory Access Trade-Off", CODES 2000 Workshop on Hardware/Software Codesign, pages 51–55, May 2000.