Hardware Algorithms for Tile-Based Real-Time Rendering

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen

op dinsdag 1 mei 2012 om 15:00 uur

door

Dan CRIŞU

inginer
Facultatea de Electronică şi Telecomunicaţii
Universitatea “Politehnica” din Bucureşti
geboren te Boekarest, Roemenië

Dit proefschrift is goedgekeurd door de promotor:
Prof.dr.ir. H.J. Sips

Copromotor:
Dr.ir. S.D. Cotofana

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
Prof.dr.ir. H.J. Sips	Technische Universiteit Delft, promotor
Dr.ir. S.D. Cotofana	Technische Universiteit Delft, copromotor
Prof.dr. K.L.M. Bertels	Technische Universiteit Delft
Prof.dr.ir. F.W. Jansen	Technische Universiteit Delft
Prof.dr. K.G.W. Goossens	Technische Universiteit Eindhoven
Prof.dr.ir. A. Rubio	Universitat Politecnica de Catalunya
Prof.dr.ir. M. Bodea	University ”Politehnica” of Bucharest

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Crişu, Dan
Hardware Algorithms for Tile-Based Real-Time Rendering
Dan Crişu. – [S.l. : s.n.]. – Ill.
Thesis Technische Universiteit Delft. – With ref. –
Met samenvatting in het Nederlands
ISBN 978–90–72298–26–3
Subject headings: computer graphics, computer design and engineering

Keywords: 3-D graphics algorithms and architectures, tile-based rasterization, embedded systems, low-power circuits, computer arithmetic

In loving memory of my father

Hardware Algorithms for Tile-Based Real-Time Rendering

Dan Crişu

Abstract

In this dissertation, we present the GRAphics AcceLerator (GRAAL) framework for developing embedded tile-based rasterization hardware for mobile devices, meant to accelerate real-time 3-D graphics (OpenGL compliant) applications. The goal of the framework is a low-cost, low-power, high-performance design that delivers good image quality. We focus on several key problem areas in tile-based rasterization, such as: rasterization and triangle traversal, antialiasing, and geometrical primitive list sorting. We present an original triangle traversal hardware algorithm implementation, composed of a systolic primitive scan-conversion subsystem and a logic-enhanced memory subsystem, able to deliver 4 pixel positions per clock cycle in a very advantageous spatial pattern, exploited to reduce the power consumption and increase the throughput, to the pixel processing pipelines for rasterization. Area-sampling antialiasing is achieved with a pixel-coverage mask generation algorithm that reduces the mask storage costs by exploiting the quadrant symmetry property when deriving on the fly, via computationally inexpensive operations, the required coverage masks. The costs are reduced by an order of magnitude and the image quality, i.e., coverage mask accuracy, almost doubles when compared to prior state-of-the-art implementations. At the front end of the rasterization process, as the host processor needs to be able to process different other system tasks in a system-on-chip embedded architecture, we propose a novel and efficient hardware primitive list sorting algorithm that lowers on the one hand the effort of the host processor required to generate the primitive tiling lists and reduces on the other hand the external memory traffic. For an implementation footprint similar to an 8KB SRAM memory macro, the number of the instructions on the host processor for tiling list generation was lowered by 4–$9\times$ and the memory cost by 3–$6\times$, for our embedded benchmark suite GraalBench, when compared to the software driver implementation alone. Our estimations indicate that the GRAAL design, clocked at a frequency of 200MHz, can sustain a rendering and fill rate of 2.4 million triangles/s and 460 million pixels/s for typical 3-D graphics scenes.

Acknowledgments

The work presented in this dissertation contains the results of my research performed at the Computer Engineering Laboratory of the Electrical Engineering, Mathematics and Computer Science Department, Delft University of Technology (2001–2004). This work would not have been possible without the essential and gracious support of many individuals. I would like to take this opportunity to thank them.

First and foremost I offer my sincerest gratitude to my supervisor, Dr. Sorin Coţofană, to the departed promotor Prof.dr. Stamatis Vassiliadis, and to promotor Prof.dr. Henk Sips, for giving me the opportunity to perform my PhD research within their group. They have supported me throughout my thesis with their patience and knowledge whilst allowing me the room to work in my own way. I attribute the level of my PhD achievement to their encouragement and effort, as without them this thesis would not have been completed or written. I loved working in the relaxed and friendly atmosphere they managed to create around us. Thank you.

I would also like to acknowledge Prof.dr. Mircea Bodea for guiding my steps in Microelectronics and believing in me, as a true mentor. Together with Prof.dr. Dan Claudius, Prof.dr. Bodea made me love my work and sustained me even in moments of doubt. Without them, I might have never had the chance to start a dissertation at Delft University of Technology. Thank you, the journey of discovery in Microelectronics has been such a joy with you.

I want also to thank Dr. Arjan van Genderen for helping me with generous advice on setting up the EDA tools for my experiments whenever I needed. I would also like to acknowledge Bert Meijs for helping me access the IT infrastructure in our group and department. Special thanks to Lidwina Tromp for taking the trouble to help me cut through the red tape that I came across. Without their support, I would not have been able to focus strictly on my research.

Furthermore, I would like to thank my office mates, Iosif (Gabi) Antochi and Pepijn de Langen, who have often had to bear the brunt of my frustration and rage against the world when my experiments were falling over. They have always offered generous support and an inspiring conversation. I am especially indebted to Gabi for essential work done on the GRAAL project, without which the quality of this thesis would might have suffered tremendously. I would also like to acknowledge another friendly and cheerful group of fellow doctoral students: Casper Lageweg, Pyrrhos Stathis, and Dmitry Cheresiz who provided me with hours of witty banter and amusement, while relaxing during a lunch or coffee break.

I am also indebted to the small academic Romanian community at the university for hanging around with me and helping me around. Thank you, due to you I have never been homesick.

I would like to give my heartfelt appreciation to my parents, who brought me up with their unconditional love, and encouraged me to pursue my dreams and be always on the righteous path. Their words of encouragement and push for tenacity still ring in my ears. To my sister who has been by my side since we were kids. To my entire wonderful family, I thank you.

To my beloved wife, Cora, my soul mate, who has accompanied me with her love, unlimited patience, understanding and encouragement, through the trials and tribulations of the life abroad. And to my son, Paul, and my daughter, Maria, for bringing such an exhilaration and sense of purpose and joy into my life. You have been my best cheerleaders.

This project was funded by the grant IS00012 from Nokia Research Center, Tampere, which I gratefully acknowledge. I would like to thank Petri Liuha, Tapani Leppanen, and Dr. Kari Pulli for the many insightful discussions we had during the project time frame.

Dan Crişu London, UK

April 2012

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Only a decade ago, at the turn of the millennium, the simple idea of having interactive mobile 3D graphics sounded far fetched. Mobile devices had to manage battery life thus imposing a limit on system performance. Thermal considerations also applied, due to small device footprints heat evacuation was difficult, and cooling systems available in desktop systems were unfeasible, therefore limiting again the performance that could be achieved. As a result, mobile devices were struggling with slow CPUs, little memory capacity and small monochrome displays.

Fast forward a few years. The unrelenting march of Moore's law [45] made yet again the electronic circuits exponentially faster and smaller, with more number-crunching power in CPUs and larger accommodating memories. Not so much so, the battery life has increased only linearly. However, the most important enabler for mobile graphics has been the fast improvement of display technologies [76]. That development was first fueled by the demand from digital cameras, though now the greatest demand comes from mobile phones. A typical mobile phone around year 2000 had an 84×84 1-bit monochrome display, refreshed a few times per second, but in 2011 24-bit RGB displays are becoming the norm, with typical display resolutions around 320×240 pixels, refreshed 30 times per second.

The driver for graphics technology development on mobile devices seems to be the interactive entertainment, especially gaming, markets. In [18] the results of a study in worldwide revenue in the entertainment sector were presented. This economic study indicated that the video gaming industry was going to be the main driving force in the entertainment sector by 2009, replacing the role of non-interactive entertainment like films and music. The total world-

wide revenue in the video game industry in 2009 was $57 billion, in 2011 was $74 billion, and projected to reach a staggering $115 billion by 2015 (mostly due to mobile gaming), with a growth at an annual rate of almost 13% [72]. More importantly, these studies showed a considerable increase in the total revenue for wireless gaming on mobile terminals growing from $281 million in 2004 [18] to $5 billion in 2011 [72], which evaluated to be a 50% annual increase. By now these figures have been materialized by three major players that are involved in mobile gaming devices: Sony, Nintendo, and Apple, of which the first has sold 60 million devices (PSP) as of March 2010, the second has sold 128 million devices (Nintendo DS) as of March 2010, and the last 37 million devices (iPhone) as of March 2012 [1]. These figures clearly show an increasing market for mobile gaming platforms.

Driven by commercial interest in mobile gaming, the request for increasingly fast, graphics-rich, user-friendly interfaces and entertainment environments has triggered the introduction of a new field in research that provides an intriguing design challenge for system engineers. In the traditional computer gaming industry the system solutions that enable computer graphics are usually realized with maximum performance in mind, resulting in the range of power consuming graphics cards available for personal computers today. This is in contrast to the field of mobile graphics, where power consumption, memory bandwidth, and assembly cost are three important additional design criteria used to evaluate possible system solutions. Therefore, implementing graphics acceleration on mobile devices is a field of particular interest, and two APIs (Application Programming Interfaces) were soon ready to emerge: 1) M3G [76], or Mobile 3D Graphics API, written to accelerate graphical applications written in Java, which is the most common programming language used to write mobile applications, and 2) OpenGL ES [48], the mobile counterpart of OpenGL [80], the widespread industry standard for computer graphics. Very quickly afterwards, embedded solutions for computer graphics meant to accelerate the afore-mentioned APIs were announced by Imagination Technologies (PowerVR), ATI, BitBoys, ARM (Falanx), Mitsubishi, NVidia, Sony, and Toshiba [76].

In this line of reasoning, in this thesis, we present a framework for developing embedded rasterization *hardware* for mobile devices. In particular, within this framework, we propose a novel design for an embedded *tile-based* rasterizer called GRAphics AcceLerator (GRAAL). GRAAL is an OpenGL compliant rasterizer to be used in a tile-based rasterization scenario, designed to be low-cost, potentially low-power, having relatively high-performance, and delivering good quality image results. The merits of the proposed implementation are

assessed within our design framework.

For details on *software* aspects of GRAAL, i.e., the software driver stack to be run on the host processor to facilitate the communication with the hardware rasterizer, the reader is referred to Antochi's PhD thesis [9], which was carried out within the same project framework and in the same time frame. The present thesis and thesis [9] are complementary, reflecting the synergistic nature of modern complex systems.

In this introductory chapter, we highlight the initial requirements and freedom degrees of our research activity, that define the dissertation scope. We especially raise three fundamental research questions that are to be answered throughout the presentation. The chapter is organized as follows. The problem overview and the dissertation scope are presented in Section 1.1. The terminology used throughout this thesis is addressed in Section 1.2. The main contributions of this dissertation are enumerated in Section 1.3. Section 1.4 completes the chapter with an overview of the thesis.

1.1 Problem Overview and Dissertation Scope

Along the years a significant amount of work has been carried out to tackle power/performance/bandwidth design objectives at the same time. With the advent of ULSI (Ultra Large Scale Integration), several system modules could be integrated on a single SoC (System On Chip), with the communication between these modules being done via a small on-chip connection, usually a bus, instead of a longer off-chip connection. An example of an on-chip bus is the AMBA system bus [11] especially created for use in conjunction with ARM processing cores. Consequently performance is increased, and power consumption and assembly costs are reduced.

A critical component for any graphics system is the framebuffer, a region of memory that contains the color information for every pixel on the display for a certain frame, effectively holding the desired image to display. This memory is read pixel by pixel by the scan converter in order to display the resulting image on a raster device. The computer graphics process is responsible for generating the contents of the framebuffer before it can be displayed. Most computer graphics hardware (and software) is optimized for processing triangles, because it simplifies the computations and because every object can ultimately be represented by triangle meshes [4][40].

During the last stage of the graphics process, the rasterization stage, the above

mentioned framebuffer is continuously filled with color values corresponding to triangles that are to be displayed on the screen. The rasterization is performed triangle by triangle, and for each triangle it results in color and depth values to be stored in the framebuffer. The depth value is used to determine which triangle is in front. Some computed colors will ultimately be replaced by the colors of another triangle and some colors will be blended with a new color in order to generate a final color value. The new color could be read by sampling a texture stored in a region of memory as well. It is important to observe in this process that the framebuffer is also frequently accessed to retrieve previously computed color (and especially depth) values during rasterization.

Unfortunately, even for small displays such as those widely used in mobile devices, a large amount of data is required to be stored in the framebuffer. For example, a 640×480 size display already requires a framebuffer of over 1MB. If the buffer would be implemented on chip using SRAM (Static RAM), each bit cell would take 6 transistors, with huge area implications. The same buffer could be implemented with fewer transistors using eDRAM (embedded Dynamic RAM), but the stacked or trenched capacitor that would accompany the charging transistor would require an alteration of chip fabrication technology that is expensive and could potentially reduce the performance of logic transistors [44]. For mobile devices, the framebuffer is therefore considered to be too large to be implemented on the same chip as the graphics accelerator, and it is usually implemented on an external memory chip.

Research was first carried out to tackle the external memory traffic towards/from framebuffer or texture memory of the graphics accelerator. External traffic is a major source of power dissipation [42] mainly induced by the high capacitance and resistance of printed circuit board connections when compared to on-chip connections. Most notable, solutions to reduce the external memory traffic were proposed in the field of texture compression [13][38][86], and increasing the texture access locality [25][50][56] at the cost of extra on-chip logic. Other work on mobile graphics includes proposals for anti-aliasing, texture filtering, and occlusion culling [5][3][60] that reduce the power consumption of graphics hardware. Even so, implementing a conventional off-chip framebuffer computer graphics solution on mobile devices becomes a liability due to the fact that sustaining high framebuffer access rates during rasterization translates to high battery discharge rates.

Instead of using a traditional rasterization solution where triangles are processed for the entire screen, *tile-based rendering* (or alternatively known as *bucket rendering* [70][21][54] or *chunk rendering* [90][12]) was initially pro-

posed for high-performance, parallel renderers, where the screen was split in non-overlapping tiles, and the polygons associated geometrically to each tile were rendered in parallel on different processing units, with different load balancing strategies. However, on low-power mobile architectures the tiles are rendered sequentially one by one. If interested in tile-based rendering in general, for a comprehensive discussion, applied to pipeline stages other than the rasterization stage (the main focus of this thesis), the reader is referred to [53].

In tile-based rasterization, the screen is divided into small sections, called tiles. All the rasterization instructions that compose a scene, mainly triangle instructions and state changing instructions, are duplicated for all the tiles they belong to, which enables independent tile processing. On mobile devices, however, due to design constraints, the tiles are processed sequentially by a single rasterizer. The rasterizer needs only a local tile-sized framebuffer where the complete image for that tile is first generated before transferring it to the large external framebuffer.

The main advantage of tile-based rasterization for mobile graphics is that the local framebuffer, required during the rasterization of a tile, can be stored on the same chip as the graphics accelerator. This reduces the external communication during rasterization and results in a lower power consumption when compared to fullscreen rasterization.

The main disadvantage of tile-based rasterization is that it requires a sorting stage before rasterization in order to be effective. In this sorting stage triangles are examined to determine the tiles they are present in. Without this sorting stage, the rasterization hardware wastes valuable time and resources by computing the overlap of triangles that might not even be present in the selected tile. A significant amount of hardware workload can be removed by prior sorting because triangles are usually only present in a small number of tiles [8][59]. This sorting results in a *tiling list* of linked lists of blocks containing the geometry and state-changing commands *per tile* stored in the system memory, rather than a single set for the entire scene. Therefore, the tile-based rasterization, although it saves a lot of external traffic, it introduces a second source of transfers from the host processor to the graphics accelerator via the system memory. In addition, in an embedded system, the host processor has a general purpose role, and the tiling list computation has to be uploaded to the graphics accelerator as much as possible in order to free the host processor that orchestrates the entire activity of the peripherals in the rest of the system.

As described above, there are two important data transfers performed in the system for the purpose of embedded rasterization, and both of them have the

system memory as an intermediary: the first transfer is from the host processor to the hardware rasterizer, and the second transfer is from the rasterizer to the framebuffer location in memory. An embedded hardware rasterizer has to reduce to a minimum the memory bandwidth consumed for rasterization, because any embedded system has available only a limited memory bandwidth budget.

The initial requirements and freedom degrees of our research activity [32] can be summarized as follows:

1. Investigate the tile-based rasterization paradigm to assess what gains and shortcomings can be expected from it for mobile graphics acceleration of OpenGL.

2. Develop a graphics benchmark suite with realistic workloads for the next generation mobile graphics accelerators.

3. Propose hardware algorithms, amenable to efficient circuit implementations, to maximize the gains and alleviate the shortcomings identified.

4. Quantify the hardware algorithm implementations, for various tradeoffs cost/power/performance/image quality, by simulating the workloads using the developed graphics benchmark suite.

Based on these requirements and the available development tools and ASIC libraries for integrated circuit design, we restricted our dissertation scope as follows:

- The proposed tile-based hardware rasterization engine achieves full OpenGL compliance only by a combination of software driver-level techniques and hardware algorithms implemented by the rasterization engine acting in synergy. Thus, only the algorithms implemented in hardware are discussed and software driver-level issues that help augmenting the hardware capabilities are mentioned only when they are deemed absolutely necessary.

- Since we assess the merit figures of GRAAL instantiated in the embedded domain where we simulate an entire system-on-chip (containing bus masters such as a host processor, GRAAL rasterizer's memory transactors, the scan converter memory engine, and bus slaves such as the external memory interface, GRAAL rasterizer's register blocks, and the scan converter register interface), we do not consider as the host processor

a superscalar general-purpose processor augmented with multimedia-assist instructions, e.g., MMX-extended Pentium, but we restrict ourselves to an embedded general-purpose processor, e.g., ARM1020T. Therefore, our virtual driver stack does not make use of any computations running in SIMD fashion, to potentially reduce the load on the host processor.

- The underlying integrated circuit technology used for circuit implementation is UMC 0.18μm Logic 1.8V/3.3V 1P6M GENERICII CMOS and all the reported results are valid for this technology node. It was chosen because, at the time when this research was carried on, the accompanying ASIC libraries were fully characterized for power consumption, allowing the power consumption of synthesized circuits to be estimated with Synopsys Design Compiler, and characterized for interconnect parasitics, and thus permitting parasitics extraction to be performed on custom circuit layouts required to simulate custom memory circuits in HSPICE.

- The proposed antialiasing technique presented in this thesis belongs to pre-filtering (area sampling) antialiasing methods, being an antialiasing method that is challenging to achieve in a low-cost embedded rasterizer. Full scene antialiasing method implementations are not presented, because they are either trivial to achieve with the proposed hardware, or not amenable to the tile-based paradigm. In the former category we could mention the supersampling or multi-sampling with intra-pixel sample resolve case where the on-chip tile buffer stores samples instead of pixels and filters them to pixels prior to the transfer to the external framebuffer. In the latter category we could mention the multi-sampling with inter-pixel sample resolve case for tile-based rasterizers, as pixels have to share, for final filtering, samples with neighbouring pixels and this is difficult at tile edges introducing inter-dependencies in the tile processing.

1.2 Terminology

Before we present the main contributions of the dissertation, we discuss our usage of particular words and terminology.

In the discipline of computer engineering, the term *architecture* is typically used as an abbreviation for *computer architecture*, which is defined as the

conceptual structure, attributes, and behaviour of a computer as seen by a machine-language programmer [46]. A computer, in turn, consists of three major components: the processor that includes a central processing unit (CPU) and a number of on- or off-chip coprocessors, memory, and peripherals (used for input and output). Computers can be classified in two categories: general purpose, and *embedded*, i.e., computers that perform specialised tasks like the ones in cars, cellular phones, game consoles, and other consumer electronics). With embedded computers, people accomplish some task, blithely and happily unaware that there is a computer involved [57]. Prevalent implementations of computers for embedded systems are *system-on-chip* (SoC), where all the components except the memory are laid out on the same silicon die. In embedded computers, the CPU is also alternatively named *host processor*. For this thesis, we examine only the architecture and the design of a *real-time 3-D graphics hardware accelerator*, which could be one of the possible peripherals in an embedded computer. Hence, in this dissertation, we use the term *architecture* as an abbreviation for the *graphics hardware accelerator architecture* rather than an entire computer if we do not specify otherwise.

Real-time computer graphics is the subfield of computer graphics focused on producing and analyzing images in real time. The term is most often used in reference to *interactive* 3-D computer graphics, typically employing a graphics hardware accelerator, and having video games the most noticeable application. The goal of computer graphics is to create a computer generated image having certain characteristics, e.g., being almost photo-realistic, or cartoonish in appearance, etc. This image is often called a *frame* and it is stored in a special area of memory called *framebuffer*. One can determine the method's real-timeliness by observing how fast these images or frames are generated in a given second. The goal of real-time graphics is to generate a number of frames above a certain threshold speed of the human-visual system, usually about 30 frames/s, where the perception of animated frames becomes fluid.

In this thesis, we accelerate only the graphic stages corresponding to a process called *rasterization*, and therefore, we are also using the term *graphics rasterizer* to designate the graphics accelerator. Rasterization is the process of determining the set of pixels covered by a *geometric primitive*. The rasterization process employs a datapath called a *pixel pipeline*. A rasterizer could employ in hardware multiple pixel pipelines, to increase the throughput. Each triangle, line, and point (the latter two are usually represented using triangles) is rasterized according to the rules specified for its kind. The results of rasterization are a set of pixel locations, as well as a set of fragments. The term *pixel* is the short version of *picture element*. A pixel represents the contents of

the *framebuffer* at a specific location, such as the color, depth, and any other values associated with that location. A *fragment* is the data, generated in the pixel pipelines, that can potentially update a particular pixel. The term fragment is used because rasterization breaks up each geometric primitive, e.g., a triangle, into pixel-sized fragments for each pixel that the primitive covers. A fragment has an associated pixel location, a depth value, and a set of interpolated parameters, such as a color, a secondary (specular) color, and one or several texture coordinate sets. These interpolated parameters are derived from the transformed vertices that make up the particular geometric primitive used to generate the fragments. If a fragment passes the various rasterization tests, the fragment updates a pixel in the framebuffer.

1.3 Main Contributions

The display of graphics in real-time, which is the focus of this work, places high demands on mobile devices for transmission, storage, and computation. Dedicated hardware acceleration makes therefore more efficient use of premium embedded resources (power consumption and memory bandwidth) than the more flexible software implementations.

As indicated earlier, most computer graphics hardware (and software) for real-time rasterization is optimized for processing triangles, because it simplifies the computations and because every object can ultimately be represented by triangle meshes. To find the pixels to be rasterized within the area covered by the triangle, many algorithms [92, 66], based on edge functions [74], have been proposed so far to efficiently rasterize triangles on traditional full-screen architectures, but none, to the best of our knowledge, has been proposed for efficient rasterization in a tile-based architecture. All of the proposed algorithms are based on the following conceptual algorithm: while not all the positions inside the triangle are exhausted do 1) save the rasterization context, 2) move to a new rasterization position on screen, 3) test the edge functions value for that position to see if the position is inside the triangle, 4) if it is inside, communicate the position to the pixel processing pipelines and update the rasterization context or else restore the rasterization context, 5) based on the edge functions computed earlier, try to predict the next pixel position inside the triangle. Computationwise, the main difficulty in tile-based rasterization (with this generic algorithm) is to find the first pixel position inside the triangle to be rasterized, as the position of the triangle could be arbitrary in relation to the current processed tile. Our experiments indicated that the overhead can

be between 50%-300% of the triangle rasterization time. In addition, there is always overhead associated with *ghost* triangles, triangles that are assigned to the current tile when they have nothing in common with it (this is due to the simplest algorithm in the software driver that assigns triangles to tiles based on a primitive bounding box test; other more complex tests in the software driver were envisaged eliminating the ghost triangle problem completely, but moving the costs to software). In full-screen rasterization, this overhead is nonexistent due to the fact that a starting point inside the primitive can always be found, e.g., the gravity center.

Based on these considerations, the following major open questions can be posed with respect to tile-based rasterization:

1. Could a hardware algorithm be found to mitigate the ghost triangle overhead?

Contribution 1: We investigate this question and propose an efficient tile-based traversal algorithm hardware implementation that generates pixel positions at high rates with almost no overheads. The proposed design has a latency of several clock cycles and then can deliver a throughput of up to 4 pixel positions per clock cycle to the pixel pipelines for each triangle. Related to the first major open question is:

2. Is it possible to communicate the generated pixel positions to the pixel pipelines in a spatial pattern that is beneficial to a mobile low-power rasterizer?

Contribution 2: We answer the question by presenting hardware implementations working in conjunction with the afore-mentioned traversal algorithm circuits. They are able to deliver pixel positions in Morton order (a particular *space-filling curve* in 2-D plane, i.e., a curve whose range contains the entire 2-D unit square) that increases the hit ratio of texture caches and allows the pixel positions, generated simultaneously, to always be mapped to different memory banks in the local tile framebuffers thus breaking the *read-modify-write* dependencies associated with depth test and color blending. As a result, the power consumption is reduced and the performance is increased.

Contribution 3: In addition, we have proposed an efficient, high image quality run-time pixel coverage mask generation algorithm for embedded 3-D graph-

ics antialiasing purposes, that is compatible with the above triangle traversal algorithm. The algorithm was implemented assuming 4×4 subpixel coverage masks and two's complement number representation. However, it has a higher degree of generality: it can be incorporated in any antialiasing scheme with pre-filtering that is based on algebraic representation of primitive's edges, it is independent of the underlying number representation, and it can be adapted to other coverage mask subpixel resolutions with the only prerequisite for the masks to be square. For the presented hardware implementation, the costs are reduced by an order of magnitude and the image quality almost doubles when compared to prior state-of-the-art implementations.

Since, as previously indicated, tile-based rasterizers rely on triangle sorting for the creation of the tiling lists in system memory, another major open question to be raised is:

3. Is it possible to reduce the host processor computational overhead for the creation of the tiling lists and simultaneously reduce the traffic they create from the host processor to the external memory and from the external memory to the graphics rasterizer?

Contribution 4: We have proposed a novel and efficient hardware primitive list sorting algorithm, able to store a number of the primitives on chip and to perform tile binning based on the primitive bounding box test, that lowers on the one hand the effort of the host processor required to generate the primitive tiling lists and reduces on the other hand the external memory traffic. For an implementation footprint similar to an 8KB SRAM memory macro, the number of instructions on the host processor for tiling list generation was lowered by 4–9 times and the memory cost by 3–6 times, for our embedded benchmark suite GraalBench, when compared to the software driver implementation alone.

To answer these questions, our research activity calls for a high-level architecture design and new implementation of graphics algorithms. Consequently, it includes algorithm research, the creation of hardware/software co-design tools for embedded graphics, and hardware design (synthesizable SystemC RTL code and full-custom ASIC design at layout and circuit level). As demonstrated later (based on the obtained results), we developed novel hardware architectures that are suitable for mobile graphics rasterization with significant performance advantages. More specifically, the rest of the contributions are:

Contribution 5: We have presented a complete mathematical formalism that

could be applied to any tile-based raterization engine. We have described how, after an initial computational stage called triangle setup, which is relative to the current tile and current triangle, operations could be performed to each pixel (or pixel block), in parallel to other pixels (or pixel blocks), to generate the triangle stencil or the attributes that are required by the pixel processing pipelines. Also, we have presented how values, for neighbouring pixels ocurring within the same pixel block, could be derived using only two-operand additions, which are cheaper to implement in hardware than multiplications.

Contribution 6: We have proposed a versatile hardware/software co-simulation and co-design tool framework for 3-D graphics accelerators. The tool framework offers a coherent development methodology based on an extensive library of parametrizable graphics pipeline components modelled at RT-level in SystemC. The framework is an open system, allowing integration with other third-party SystemC models to enable an entire embedded platform simulation if desired. The framework incorporates tools to assist in the visual debugging of the graphics algorithms implemented in hardware, and to estimate the performance in terms of throughput, power consumption, and area.

Contribution 7: We have designed novel hardware circuitry to implement, in a very efficient manner, the algorithms presented above. Driven by the ever increasing delays in the interconnect networks with each technology node, we have adopted modern implementation techniques for embedded design, that not so long ago were the attributes of high-performance computing: high-throughput circuitry, computation units and data storage interwoven together, and a re-compute rather than a compute-once distribute-and-reuse-many-times strategy [57]. Therefore, the triangle traversal algorithm uses a systolic primitive scan-conversion subsystem that has a throughput of 16 pixels per clock cycle. In addition, as a part of the same triangle traversal algorithm, and for the primitive list sorting algorithm, a logic(arithmetic)-enhanced memory is employed. Special considerations were given 1) not to compromise the operational noise margins of the circuitry and 2) the enhancing logic(arithmetic) cells to have a layout with a similar pitch to the data storage cells in order to facilitate high cell integration densities. Therefore, in the logic(arithmetic)-enhanced memory, the storage cells were implemented with traditional SRAM circuitry (two cross-coupled inverters generating the storing latch and two NMOS pass transistors for access), but the logic(arithmetic) cells were implemented in a domino dynamic logic style that enabled all the features described above.

1.4 Overview of Dissertation

In the second chapter, a generic 3-D graphics pipeline is overviewed and the main operations performed are described by laying emphasis on the perspective-correct rasterization from a theoretical point of view. The operations derived there have to be implemented mandatorily, in one way or another, by every hardware rasterization engine. They are relevant because one could understand the degrees of freedom she/he has at each step in order to find how to achieve effective hardware parallelism — the equations are exploited over the next two chapters. The chapter also presents a brief description of the anti-aliasing theory and the existing hardware developments to cope with the aliasing problem. Ample references are made to the OpenGL specification (a 3-D graphics library chosen to be hardware accelerated by the present work), thus outlining the OpenGL embodiments of the theoretical aspects presented there-in. For the readers that are not familiar with the computer graphics field, i.e., readers with a hardware design background and/or a computer architecture background, this introduction is much required and helps them finding their bearings in the thesis, even when we zoom in to present and focus on the parts we improve. However, readers that are familiar with the computer graphics field could skip without any loss of continuity Section 2.1 and Subsection 2.3.1, which present background material regarding the graphics pipeline stages, and antialiasing theory respectively.

In Chapter 3, an algorithmic view of a potential OpenGL-compliant tile-based hardware rasterization engine is described. In this context, the term *potential* refers to the proposal that constitutes a platform to build on towards full OpenGL compliance. This can be achieved only by a combination of software driver-level techniques and hardware algorithms implemented by the rasterization engine. Thus, this chapter discusses the algorithms implemented in hardware whereas the software driver-level issues that help augmenting the hardware capabilities are mentioned only when it is absolutely necessary. The proposed rasterization engine is mainly focussed on three-dimensional triangle rasterization, as this represents the main operation to be performed on any rasterization engine. Consequently, the three-dimensional triangle is the centric element of the rasterization engine, since all other primitives ,e.g., points, lines, and general polygons, can be reduced to triangles at the software driver-level. First, a complete mathematical formalism is presented that could be applied to any tile-based raterization engine. More in particular, it is described how, after an initial computational stage called triangle setup relative to the current tile and current triangle, operations could be performed to each pixel (or

pixel block), in parallel to other pixels (or pixel blocks), to generate the triangle stencil or the attributes that are required by the pixel processing pipelines. Also, it is described how values, for neighbouring pixels ocurring within the same pixel block, could be derived using only two-operand additions, which are cheaper to implement in hardware than multiplications. The chapter ends by presenting how the described rasterization engine is capable to perform well with a multiplicity of triangle rasterization methods, e.g., filled flat- or Gouraud-shaded, both aliased or antialiased, while at the same time it could accomodate tradeoffs in cost, power, performance, with good quality image results.

Chapter 4 presents the GRAAL (GRAphics AcceLerator) framework, a versatile hardware/software co-simulation and co-design tool for embedded 3-D graphics accelerators developed by us. The GRAAL design exploration framework is an open system which offers a coherent development methodology based on an extensive library of graphics pipeline components modeled at RT-level in SystemC, a language developed specifically for system level simulation and design. As a consequence, an entire system-on-chip can be simulated by integrating third-party SystemC models of components (microprocessors, memories, and peripherals) along with our own parameterizable SystemC RTL model of the graphics hardware accelerator. GRAAL framework incorporates tools to assist in the visual debugging of the graphics algorithms implemented in hardware and to estimate the performance in terms of throughput, power consumption, and area. We complete the chapter by presenting results that demonstrate the effectiveness of the design exploration framework.

In Chapter 5, an efficient low-cost, low-power hardware implementation of a run-time pixel coverage mask generation algorithm for embedded 3-D graphics antialiasing purposes is presented. The algorithm exploits the quadrant symmetry property allowing the storage of only the coverage mask information for a few representative edges in one of the quadrants of the plane, the rest of the information being derived on the fly via computationally inexpensive operations. The algorithm is presented assuming 4×4 subpixel coverage masks and two's complement number representation. However, it has a higher degree of generality: it can be incorporated in any antialiasing scheme with pre-filtering that is based on algebraic representation of primitive's edges, it is independent of the underlying number representation, and it can be adapted to other coverage mask subpixel resolutions with the only prerequisite for the masks to be square. In addition, the proposed hardware algorithm represents a natural extension of the algorithm presented in Chapter 3. After the general algorithm is described, a qualitative analysis is performed, the computational

accuracy of the algorithm is investigated, and hardware implementation and simulation results are presented.

Chapter 6 describes an efficient tile-based traversal algorithm hardware implementation to accelerate primitive traversal in 3-D graphics tile-based rasterizers. The hardware implementation consists of two components: a systolic primitive scan-conversion subsystem and a logic-enhanced memory. During rasterization time, the logic-enhanced memory is filled up in several clock cycles by the systolic primitive scan-conversion subsystem with the stencil of the primitive. Once the shape of the primitive has been coded inside the memory, the memory internal logic is capable of delivering on request in one clock cycle at least one and up to four pixel positions to the pixel processing pipelines, signaling when all the pixel positions are consumed. The proposed tile-based traversal algorithm hardware implementation presents the following benefits: it handles ghost primitives efficiently, pixel positions are communicated in a spatial pattern (Morton order) that increases the hit ratio of texture caches, and pixel positions can always be mapped to different memory banks in the Z-buffer or color-buffer breaking the read-modify-write dependency associated with depth test and color blending thus allowing efficient pipelining. Hardware implementation results are presented at the end of the chapter.

In Chapter 7, we present a hardware primitive list smart buffer that lowers the effort on the host processor required to generate the tiling lists and reduces the external memory traffic at the same time. The primitive list smart buffer is able to store a number of the primitives on-chip and to perform tile binning based on the primitive bounding box test. The smart buffer can be queried with the current rasterized tile position and the current state tag as inputs, and as a result it presents at the output all the primitives, one primitive per clock cycle in the driver submission order, that intersect the current tile after the current state changing commands, pointed by the current state tag, have been applied. More in particular, this is achieved by a CAM memory with priority encoders on the outputs, using static RAM bit cells for storage, but dynamic domino logic for the arithmetic circuits to save area. The storage includes information related to global scene primitive vertex data and tags to global scene rasterization state, and the arithmetic circuits are able to perform primitive bounding box intersection tests against the current tile boundaries. As the global scene rasterization data contains state changing commands (i.e. color shading, occlusion tests, color blending modes) and primitives in a strict sequential order, parallel queries in CAM are made using rasterization state tags and the current tile coordinates. The result is the sequence of rasterization state changing commands and the primitives local for the current tile that are sequentially trans-

ferred to the rest of the rasterization system for rendering a tile at a time. The performance achieved by the hardware primitive list accelerator is illustrated comparatively to the software tile sorting approach performed entirely on the host processor. Hardware synthesis has indicated that the hardware implementation using an IC technology node of 0.18μm can be clocked at a frequency of 200MHz and the rendering and fill rate achieved are 2.4 million triangles/s and 460 million pixels/s for graphics scenes with typical average triangle area of 160 pixels.

Chapter 8 — the conclusion of the dissertation — summarizes our findings, and discusses our main contributions, while opening up new areas for further research.

Chapter 2

Background and Preliminaries

In this chapter, a generic 3-D graphics pipeline is overviewed. The main operations performed are described in Section 2.1, emphasizing the perspective-correct rasterization from a theoretical point of view in Section 2.2. The operations derived there have to be implemented mandatorily, in one way or another, by every hardware rasterization engine. Then, in Section 2.3, a brief description of the anti-aliasing theory and existing hardware developments to cope with the aliasing problem are presented. Throughout the chapter, we make ample references to the OpenGL specification [80] (a 3-D graphics library chosen to be hardware accelerated by the present work), thus seeking the OpenGL embodiments of the theoretical aspects presented here-in.

2.1 The Graphics Hardware Pipeline

A graphics pipeline is composed of independent functional stages, in a fixed configuration, each performing a unique graphics-related task on its stage inputs and forwarding its results to the next stage for processing, in a sequential fashion [39]. In hardware implementations, each functional stage works in parallel to any other stage on different inputs, for performance gain reasons. Depending on the particular stage, the inputs and outputs are vertices, geometric primitives, or fragments.

The graphics hardware pipeline is depicted in Figure 2.1. The 3-D application sends the graphics processor a sequence of vertices batched into geometric primitives (polygons, lines, and points), with the topologies presented in Figure 2.2. One way of speeding-up the rendering of polygonal scenes by reduc-

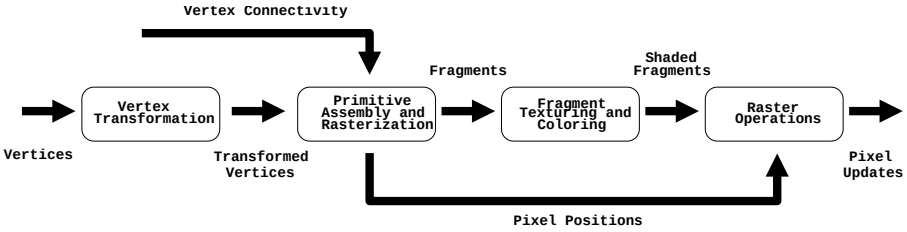

Figure 2.1: Graphics hardware pipeline.

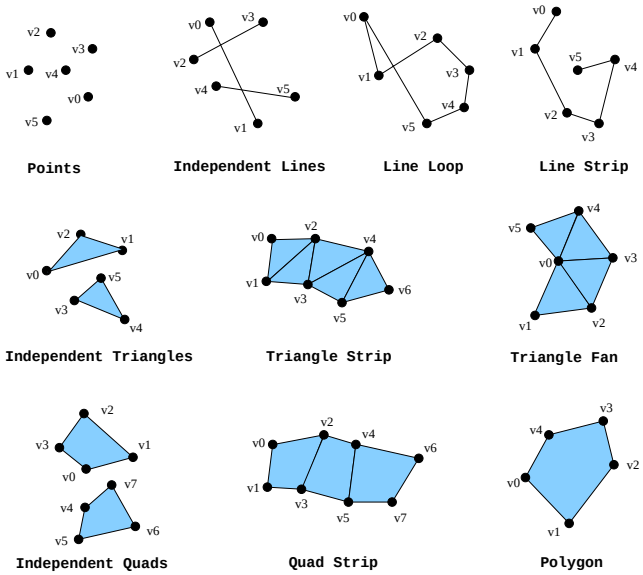

Figure 2.2: Input primitive topologies.

ing the number of vertices that have to be processed by the graphics hardware is the concatenation of adjacent primitives. For triangle meshes, several approaches have been implemented generating triangle strips. A triangle mesh represented by an optimal strip is defined by $n + 2$ vertices instead of $3n$ vertices for n individually rendered triangles.

Each vertex is described by a position and some optional attributes such as primary and secondary (or specular) colors, several texture coordinate sets, and a normal vector for lighting calculations.

2.1.1 Vertex Transformation

The first stage in the graphics hardware pipeline is the vertex transformation stage that applies a sequence of math operations to each vertex. They include transformations from world/eye coordinate systems to the screen positions used by the rasterizer, where texture coordinates for texturing, and lighting computations to generate the vertex color are also generated.

2.1.2 Primitive Assembly and Rasterization

The transformed vertices are passed to the primitive assembly and rasterization stage. The vertices and its accompanying topology information is used to generate graphics primitives (triangles, lines, or points). Following, the geometry is clipped to the viewing frustrum or using application-specified clip planes, and then is culled depending whether the primitives are front facing or back facing.

The primitives that survive the clipping or culling are rasterized. Rasterization is the process of determining the set of pixels covered by a geometric primitive, and for each primitive shape the process is distinct. The results of rasterization are a set of pixel locations, as well as a set of fragments. The term *pixel* is the short version of ”picture element.” A pixel represents the contents of the frame buffer at a specific location, such as the color, depth, and any other values associated with that location. A *fragment* is the state required potentially to update a particular pixel. Multiple fragments are generated when the rasterization process breaks up each geometric primitive into pixel-sized fragments for each pixel that a primitive covers. A fragment has an associated pixel location, and a set of optional interpolated parameters, such as depth value, primary and secondary colors, and one or several texture coordinate sets, all produced in the interpolation stage based on the pixel position. If a fragment passes the various rasterization tests (in the raster operations stage), the fragment updates a pixel in the frame buffer.

2.1.3 Fragment Interpolation

When a primitive is rasterized to multiple fragments, a process of interpolation from the vertex attributes is applied to determine the fragment parameters for each pixel covered by the primitive. In an optimized hardware graphics pipeline, this stage may use the interpolated depth to discard the fragment early if the fragment is determined to be invisible, rather than propagating it

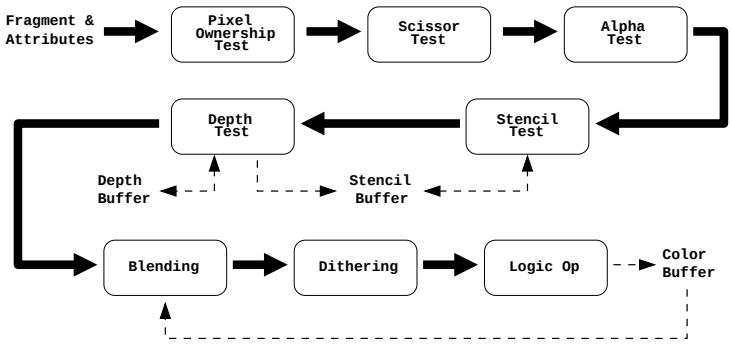

Figure 2.3: Standard OpenGL raster operations.

through the following raster operation stage. Thus this stage emits one or zero fragments for every input fragment it receives.

2.1.4 Raster Operations

The raster operations stage, a standard part of OpenGL and Direct3D graphics libraries, performs a sequence of tests and operations on each fragment, as depicted in Figure 2.3.

These tests are the pixel ownership test (relevant only for multi-window operating systems), scissor test, alpha test, and depth test. If the fragment passes all the tests, the corresponding pixel is updated in the frame buffer with the fragment parameters that have been possibly modified in this stage. These tests involve comparisons of the fragment parameters against the current frame buffer values at the same pixel location, for instance if a fragment passes the depth test (the fragment depth indicates that the fragment is not occluded) then it will replace the pixel depth value in the frame buffer with the fragment depth. Even if the fragment fails the tests and it is discarded, the frame buffer may still be modified as a side effect, e.g. if a fragment fails the depth test it may still modify the stencil value of that pixel in the frame buffer.

When all the tests succeed, a fragment color may be blended with the content of the frame buffer, suffer color dithering (if the frame buffer has a reduced bits-per-pixel capability), or be bitwise logical combined with the frame buffer, before being finally written to the frame buffer at its pixel location.

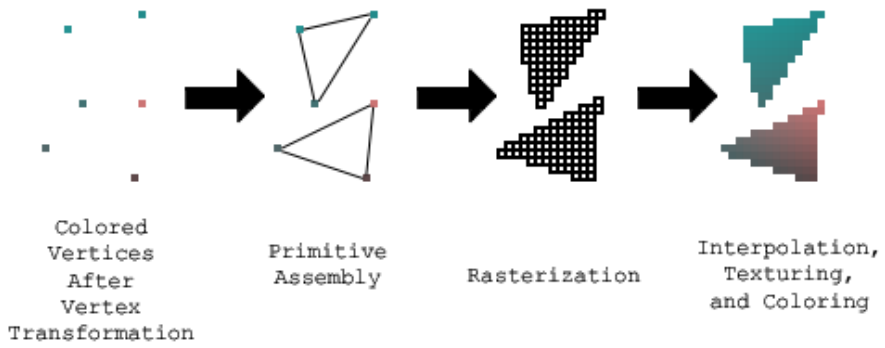

Figure 2.4: Image synthesis using the graphics pipeline.

2.1.5 Graphics Pipeline at a Glance

The process of image synthesis achieved using the stages of the graphics pipeline is depicted in Figure 2.4. The process starts with the transformation and coloring of vertices. Using the topology information coming with the vertices, the primitive assembly stage creates triangles from the vertices. Following, the rasterizer generates all the fragments covered by the triangles. Finally, the fragment parameters are obtained from the vertex attributes via interpolation, and used to update the frame buffer. The figure illustrates that many fragments are generated from just a few vertices, and therefore shows that huge data traffic must be carefully managed by a hardware graphics rasterizer, in order to be efficient.

2.2 Perspective Correct Rasterization

Perspective comes from Latin *perspicere*, “to see clearly”. As a concept in visual arts, it defines an approximate representation on a flat surface (such as paper), of a 3-D image as it is perceived by the eye. The two most characteristic features of perspective are:

- The reduction in size of drawn objects as their distance from the observer increases;
- The distortion of objects when viewed at an angle (spatial foreshortening).

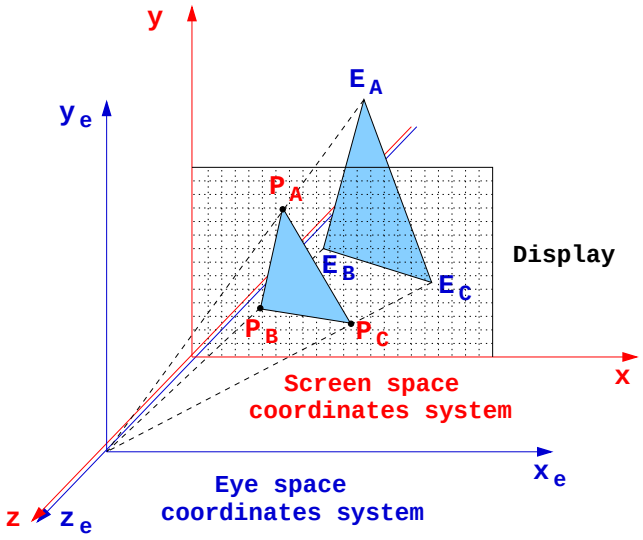

Figure 2.5: Projection of a triangle from the eye space to the screen space (both systems of coordinates are right-handed).

The interpolation process described in Subsection 2.1.3 is performed in the *screen space* (using the projected positional coordinates of the vertices) of the values (colors, texture coordinates) defined in the *eye space* (as explained below). The two coordinates systems are presented in Figure 2.5. The purpose of this subsection is to establish what kind of interpolation is needed to handle correctly in screen space the values defined in the eye space. The set of the equations synthesized here constitutes the basis for *perspective correct rasterization* [14][51].

Some notational conventions are introduced first along with a few explanations about some outstanding operations (for the task at hand) performed by the graphics pipeline.

In general, the graphical pipeline transforms a vertex through a whole chain of coordinate spaces, as the vertex makes its way to the screen. There are only two coordinate spaces that are needed to explain the perspective correct rasterization process: the eye space and the screen space.

First, polygon vertices are given in eye space, the three-dimensional coordinate space with the eye at the origin looking down the z axis. This space is significant because it is the last step in the chain in which the physical distances are meaningful, e.g., here are performed all lighting calculations. Let E be a

position in this space. In homogeneous coordinates, this is:

$E = [x_e, y_e, z_e, 1]$

To project the objects specified in eye space on the screen, a perspective distortion is necessary. This is accomplished in two steps. First, the vertex in the eye space is multiplied by a 4×4 matrix consisting of a perspective transformation and a viewport transformation. Let $\mathbf{M}$ be this matrix. Since $\mathbf{M}$ has a perspective component, the last coordinate of the transformed position will not be 1.

$\widetilde{P} = E \cdot \mathbf{M} = [\widetilde{x}, \widetilde{y}, \widetilde{z}, w]$

The clipping process is performed in this coordinate system. After clipping, the second step is performed: the w component is divided out (the process is called the homogeneous division).

$P = \widetilde{P}/w = [x, y, z, 1]$

Now, with all the preliminaries settled, the linear interpolation in the eye space of a position between two known vertices E_a and E_b can be written as:

$$E = E_a + \beta \cdot (E_b - E_a) = (1 - \beta) \cdot E_a + \beta \cdot E_b\,,\ \beta \in [0, 1] \tag{2.1}$$

where β is the *interpolation coefficient in eye space*.

After applying the combined perspective and viewport transformation in Equation (2.1):

$$\widetilde{P} = E \cdot \mathbf{M} = (1 - \beta) \cdot \widetilde{P}_a + \beta \cdot \widetilde{P}_b \tag{2.2}$$

Knowing that $\widetilde{P} = P \cdot w$, Equation (2.2) can be rewritten as:

$$P \cdot w = (1 - \beta) \cdot P_a \cdot w_a + \beta \cdot P_b \cdot w_b \tag{2.3}$$

Equation (2.2) can be written only for the fourth component w:

$$w = (1 - \beta) \cdot w_a + \beta \cdot w_b \tag{2.4}$$

From Equation (2.3) and (2.4) can be inferred that:

$$P = \frac{(1 - \beta) \cdot w_a}{(1 - \beta) \cdot w_a + \beta \cdot w_b} \cdot P_a + \frac{\beta \cdot w_b}{(1 - \beta) \cdot w_a + \beta \cdot w_b} \cdot P_b \tag{2.5}$$

If we define a new coefficient α:

$$\alpha = \frac{\beta \cdot w_b}{(1 - \beta) \cdot w_a + \beta \cdot w_b} \tag{2.6}$$

then Equation (2.5) becomes:

$$P = (1 - \alpha) \cdot P_a + \alpha \cdot P_b \,,\ \alpha \in [0, 1] \tag{2.7}$$

where α is the *interpolation coefficient in screen space*.

Remark 2.2.1: If the coordinates of a position E in the eye space are linearly interpolated between the coordinates of two vertices E_a and E_b, the coordinates of screen projection P of the position E will be also linearly interpolated in the screen space between the coordinates of the screen projections P_a and P_b of the vertices E_a and E_b, but with a different interpolation coefficient (defined by Equation (2.6)). To generalize, equally spaced positions along a segment in the eye space will be transformed into positions that are indeed colinear in the screen space; they will no longer be equally spaced. This means that a flat polygon in the eye space will be transformed into a flat polygon in the screen space.

A set of values for the interpolation coefficient α in the screen space with a constant step from each other can be found from the projection of the vertices on the screen. Then, for each α value a new position x, y, and z can be computed. This process can be reduced to the modification of the screen space coordinates x, y with fixed increments (usually one pixel) and finding the values for the z coordinate in the screen space. This means that the interpolation coefficient in the eye space β can be written as a function of the known interpolation coefficient in the screen space α as:

$$\beta = \frac{\alpha \cdot w_a}{\alpha \cdot w_a + (1 - \alpha) \cdot w_b} \tag{2.8}$$

Rewriting Equation (2.7) for the z component yields:

$$\boxed{z = (1 - \alpha) \cdot z_a + \alpha \cdot z_b} \tag{2.9}$$

Remark 2.2.2: For perspective correct rasterization, the screen space z coordinate must be linearly interpolated in the screen space.

By substituting the eye space coordinate z_e, Equation (2.1) can be generalized for other attributes of a position that are defined in the eye space, such as colors, alpha value, and texture coordinates:

$$\{C, TX\} = (1 - \beta) \cdot \{C_a, TX_a\} + \beta \cdot \{C_b, TX_b\} \tag{2.10}$$

where $C \in \{R, G, B, A\},\ TX \in \{s, t, r, q\}$.
Using Equation (2.8), Equation (2.10) can be refined further as:

$$\begin{aligned} \{C, TX\} &= \frac{(1-\alpha)\cdot w_b}{\alpha \cdot w_a + (1-\alpha)\cdot w_b} \cdot \{C_a, TX_a\} \\ &\quad + \frac{\alpha \cdot w_a}{\alpha \cdot w_a + (1-\alpha)\cdot w_b} \cdot \{C_b, TX_b\} \\ &= \frac{(1-\alpha)\cdot \frac{\{C_a, TX_a\}}{w_a} + \alpha \cdot \frac{\{C_b, TX_b\}}{w_b}}{(1-\alpha)\cdot \frac{1}{w_a} + \alpha \cdot \frac{1}{w_b}} \end{aligned} \tag{2.11}$$

From Equation (2.11) the formula for the color interpolation is:

$$\boxed{\{R, G, B, A\} = \frac{(1-\alpha)\cdot \frac{\{R_a, G_a, B_a, A_a\}}{w_a} + \alpha \cdot \frac{\{R_b, G_b, B_b, A_b\}}{w_b}}{(1-\alpha)\cdot \frac{1}{w_a} + \alpha \cdot \frac{1}{w_b}}} \tag{2.12}$$

Remark 2.2.3: For perspective correct rasterization the colors and the alpha value must be hyperbolically interpolated in the screen space.

By performing the homogeneous division of texture coordinates, Equation (2.11) for texture coordinates can be rewritten as:

$$\begin{aligned} \{S, T, R\} &= \{s/q, t/q, r/q\} \\ &= \frac{(1-\alpha)\cdot \frac{\{s_a, t_a, r_a\}}{w_a} + \alpha \cdot \frac{\{s_b, t_b, r_b\}}{w_b}}{(1-\alpha)\cdot \frac{1}{w_a} + \alpha \cdot \frac{1}{w_b}} \cdot \frac{(1-\alpha)\cdot \frac{1}{w_a} + \alpha \cdot \frac{1}{w_b}}{(1-\alpha)\cdot \frac{q_a}{w_a} + \alpha \cdot \frac{q_b}{w_b}} \\ &= \frac{(1-\alpha)\cdot \frac{\{s_a, t_a, r_a\}}{w_a} + \alpha \cdot \frac{\{s_b, t_b, r_b\}}{w_b}}{(1-\alpha)\cdot \frac{q_a}{w_a} + \alpha \cdot \frac{q_b}{w_b}} \end{aligned} \tag{2.13}$$

From Equation (2.13) the formula for the non-homogeneous texture coordinate interpolation is:

$$\boxed{\{S, T, R\} = \{s/q, t/q, r/q\} = \frac{(1-\alpha)\cdot \frac{\{s_a, t_a, r_a\}}{w_a} + \alpha \cdot \frac{\{s_b, t_b, r_b\}}{w_b}}{(1-\alpha)\cdot \frac{q_a}{w_a} + \alpha \cdot \frac{q_b}{w_b}}} \tag{2.14}$$

Remark 2.2.4: For perspective correct rasterization the non-homogeneous texture coordinates must be hyperbolically interpolated in the screen space.

The formulas given by Equations (2.9), (2.12), and (2.14) represent the formal proof for statements enclosed in the OpenGL specification [80](Chapter 3).

To rasterize a triangle, the same line of reasoning as before can be pursued. A theoretical model is presented here. To do that, one can use *barycentric coordinates* for a triangle. Barycentric coordinates are a set of three numbers m, n, and p, each in the range $[0, 1]$, with $m + n + p = 1$. These coordinates uniquely specify in eye space any point E within the triangle or on the triangle's boundary as:

$$E = m \cdot E_a + n \cdot E_b + p \cdot E_c \tag{2.15}$$

where E_a, E_b, and E_c are the vertices of the triangle in eye space. The barycentric coordinates m, n, and p can be found as:

$$m = \frac{A(EE_bE_c)}{A(E_aE_bE_c)},\ n = \frac{A(EE_aE_c)}{A(E_aE_bE_c)},\ p = \frac{A(EE_aE_b)}{A(E_aE_bE_c)} \tag{2.16}$$

where $A(E_xE_yE_z)$ denotes the area in eye space of the triangle with vertices E_x, E_y, and E_z.

After applying the combined perspective and viewport transformation on Equation (2.15):

$$\widetilde{P} = E \cdot \mathbf{M} = m \cdot \widetilde{P}_a + n \cdot \widetilde{P}_b + p \cdot \widetilde{P}_c \tag{2.17}$$

and writing it only for the w component:

$$w = m \cdot w_a + n \cdot w_b + p \cdot w_c \tag{2.18}$$

Using $\widetilde{P} = P \cdot w$, Equation (2.17) can be rewritten as:

$$P \cdot w = m \cdot P_a \cdot w_a + n \cdot P_b \cdot w_b + p \cdot P_c \cdot w_c \tag{2.19}$$

From Equation (2.18) and (2.19) follows that:

$$\begin{aligned} P &= \frac{m \cdot w_a}{m \cdot w_a + n \cdot w_b + p \cdot w_c} \cdot P_a + \frac{n \cdot w_b}{m \cdot w_a + n \cdot w_b + p \cdot w_c} \cdot P_b \\ &+ \frac{p \cdot w_c}{m \cdot w_a + n \cdot w_b + p \cdot w_c} \cdot P_c \end{aligned} \tag{2.20}$$

If we define new coefficients a, b, and c:

$$\begin{aligned} a &= \frac{m \cdot w_a}{m \cdot w_a + n \cdot w_b + p \cdot w_c} \\ b &= \frac{n \cdot w_b}{m \cdot w_a + n \cdot w_b + p \cdot w_c} \\ c &= \frac{p \cdot w_c}{m \cdot w_a + n \cdot w_b + p \cdot w_c} \end{aligned} \tag{2.21}$$

they have also the properties of barycentric coordinates being defined this time in screen space. They can be found in screen space by employing formulas similar to those given by Equation (2.16).

Writing eye space barycentric coordinates m, n, and p as a function of screen space barycentric coordinates a, b, and c yields:

$$\begin{aligned} m &= \frac{\frac{a}{w_a}}{\frac{a}{w_a}+\frac{b}{w_b}+\frac{c}{w_c}} \\ n &= \frac{\frac{b}{w_b}}{\frac{a}{w_a}+\frac{b}{w_b}+\frac{c}{w_c}} \\ p &= \frac{\frac{c}{w_c}}{\frac{a}{w_a}+\frac{b}{w_b}+\frac{c}{w_c}} \end{aligned} \tag{2.22}$$

and using the same procedure as before for the interpolation of vertex data, the following interpolation formulas can be inferred:

$$\boxed{z = a \cdot z_a + b \cdot z_b + c \cdot z_c} \tag{2.23}$$

$$\boxed{\{R,G,B,A\} = \frac{a\cdot\frac{\{R_a,G_a,B_a,A_a\}}{w_a}+b\cdot\frac{\{R_b,G_b,B_b,A_b\}}{w_b}+c\cdot\frac{\{R_c,G_c,B_c,A_c\}}{w_c}}{a\cdot\frac{1}{w_a}+b\cdot\frac{1}{w_b}+c\cdot\frac{1}{w_c}}} \tag{2.24}$$

$$\boxed{\{S,T,R\} = \frac{a\cdot\frac{\{s_a,t_a,r_a\}}{w_a}+b\cdot\frac{\{s_b,t_b,r_b\}}{w_b}+c\cdot\frac{\{s_c,t_c,r_c\}}{w_c}}{a\cdot\frac{q_a}{w_a}+b\cdot\frac{q_b}{w_b}+c\cdot\frac{q_c}{w_c}}} \tag{2.25}$$

The formulas and derivations leading to Equations (2.23), (2.24), and (2.25) are in conformity with OpenGL specification [80](Chapter 3).

Notice that formulas given by Equations (2.23), (2.24), and (2.25) did not find their way in practice due to inherent difficulty of assessing the barycentric coordinates in screen space a, b, and c. Instead, formulas given by Equations (2.9), (2.12), and (2.14) have found widespread use in practice because, to rasterize a triangle, fragment data can be interpolated either:

1. along each edge of the triangle and then across each horizontal span from edge to edge (the classical scan conversion algorithm), or
2. by a combination of horizontal and vertical walking along axes x and y (based on Pineda's rasterization algorithm [74]).

In conclusion, by examining Equations (2.9), (2.12), and (2.14), the operations required for **perspective correct rasterization** can be summarized as:

- the z screen space coordinate of fragments have to be linearly interpolated in screen space;
- the colors and the alpha value have to be hyperbolically interpolated in screen space in three steps:
  1. $\{R, G, B, A\}/w$ have to be linearly interpolated;
  2. $1/w$ have to be linearly interpolated;
  3. a per-pixel (or per-subpixel depending if the antialiasing is performed by supersampling or not) division must be performed between interpolated $\{R, G, B, A\}/w$ and interpolated $1/w$ to produce the fragment color or alpha value;
- the non-homogeneous texture coordinates have to be hyperbolically interpolated in screen space in three steps:
  1. $\{s, t, r\}/w$ have to be linearly interpolated;
  2. q/w have to be linearly interpolated;
  3. a per-pixel (or per-subpixel) division must be performed between interpolated $\{s, t, r\}/w$ and interpolated q/w to produce the fragment non-homogeneous texture coordinates.

Moreover, an additional rule has to be enforced for polygon rasterization to be OpenGL specification compliant [80](Chapter 3), this is the fragments whose centers lie on adjacent polygon boundary edges have to be produced by one and only one of the polygons involved. This rule is called polygon *point sampling*.

The OpenGL specification does not fill the implementation details of the polygon point sampling rule. For example, one from the many tie-breaker rules that an implementor may choose to adopt for rasterization of the boundaries of a polygon is presented here. Given that the polygon to be rasterized has oriented edges (clockwise or counter-clockwise), an edge can be classified (based on its associated vector from the source vertex to the sink vertex) as a quadrant one edge, quadrant two edge, quadrant three edge, or quadrant four edge (horizontal and vertical edges are also classified in one of the four previous categories). If the implementor chooses to rasterize fragments whose centers lie on quadrant one edges and quadrant two edges, and not to rasterize fragments whose

centers lie on quadrant three edges and quadrant four edges, then the polygon point sampling rule is satisfied.

The polygon point sampling rule is not enforced in the case of antialiased polygons.

In addition, the OpenGL specification [80](Chapter 5) allows the user to exercise some control over the trade-off between image quality and speed in rasterization by using the function **glHint** along with the GL_PERSPECTIVE_CORRECTION_HINT target parameter and GL_FASTEST (the most efficient option should be chosen), GL_NICEST (the highest quality option), or GL_DONT_CARE (no preference) hint parameter. The particular implementation of OpenGL specification may choose to obey the hint or not. The GL_PERSPECTIVE_CORRECTION_HINT target parameter refers to how color values and texture coordinates are interpolated across a primitive: either linearly in screen space or in the more expensive perspective-correct manner. Often, systems perform linear color interpolation because the results, while not technically correct, are visually acceptable; however, in most cases textures require perspective-correct interpolation to be visually acceptable. Thus, an OpenGL implementation can choose to use this parameter to control the method employed for interpolation.

2.3 A Comparison of Existing Hardware Antialiasing Approaches

Aliasing is the consequence of sampling any signal at a rate insufficient to uniquely reconstruct all of the spectral content input. In computer graphics, aliasing occurs when a geometrical scene from the continuous object space is represented on the discrete grid of pixels of the screen. The aliasing phenomenon creates visual artifacts known as *jaggies* or *staircasing* effect for the object edges or *popping* effect for very small objects that are moving across the screen. This is the result of an all-or-nothing approach to the rasterization process in which each pixel either is replaced with the primitive's color or is left unchanged.

These undesirable visual effects have spurred the development of *antialiasing* hardware solutions to provide more realistic imagery with reduced visual artifacts, and usually with reduced aliasing. The required insights into the aliasing problem were provided by the sampling theory.

2.3.1 Antialiasing Theory

Sampling theory provides an elegant mathematical framework to describe the relationship between a continuous signal and its samples. Images can be considered signals in the *spatial domain*, and they can be represented as a plot of amplitude against spatial position. A signal may also be considered in the *frequency domain*; that is, it can be represented as a sum of sine waves, having different frequencies, phases and amplitudes. Each sine wave represents a component of the signal's *frequency spectrum*. An image can be considered a non-periodic signal, nonzero over a finite domain that tapers off sufficiently fast (faster than $1/x$ for large values of x) and can be represented as a sum of phase-shifted sine waves [40]. However, its frequency spectrum, will not consist of integer multiples of some fundamental frequency (as a periodic signal has), and it will possibly have a very high frequency content, e.g., a polygon edge in the object space can theoretically have an infinitely steep step-function, therefore with infinitely high spectral content. Using two representations for a signal is advantageous, because some useful operations that are difficult to carry out in one domain are relatively easy to do in the other. To understand the aliasing phenomenon, the sampled signal spectrum have to be visualized. In order to do that, one has to work in the frequency domain. The operations will be carried out, to simplify, in the unidimensional space instead of the two-dimensional space. A continuous signal $f(x)$ is transformed from the spatial domain to the frequency domain by employing the *Fourier transform*:

$$F(f) = \int_{-\infty}^{+\infty} f(x) e^{-i2\pi f x} dx \tag{2.26}$$

where f is the frequency and $i = \sqrt{-1}$. For discrete signals, there is also a corresponding *discrete Fourier transform* similar in form with the previous relation, but, for the sake of brevity, it is not shown here. The frequency spectrum of a signal is represented as the magnitude of the Fourier transform plotted against the frequency. To transform a signal from the frequency domain to the spatial domain an *inverse Fourier transform* is performed:

$$f(x) = \int_{-\infty}^{+\infty} F(f) e^{+i2\pi f x} df \tag{2.27}$$

For discrete signals, there is a corresponding *inverse discrete Fourier transform*. From now on, it will be assumed that the proper Fourier transform is

applied with regard to the nature of the signal transformed (either continuous or discrete).

It is well known that multiplying two Fourier transforms in the frequency domain corresponds exactly to performing an operation called *convolution* on their inverse Fourier transforms in the spatial domain. Likewise, multiplying two functions in the spatial domain corresponds exactly to performing a convolution operation on their Fourier transforms in the frequency domain. The convolution is defined as:

$$h(x) = f(x) * g(x) = \int_{-\infty}^{+\infty} f(\tau)g(x-\tau)d\tau \tag{2.28}$$

This corresponds to taking a weighted average of the neighborhood around each point of the signal $f(x)$ — weighted by a flipped copy with respect to its vertical axis of $g(x)$ positioned at the point — and using it for the value of $h(x)$ at the point.

Basically, rasterizing a primitive from the continuous object space to the screen involves a sampling operation. By assuming that the frequency spectrum of the signal that represents the primitive in the object space is known, it is interesting to compute the frequency spectrum of the sampled signal. Sampling a signal with a frequency f_s corresponds to multiplying it in the spatial domain by the *comb* function shown in Figure 2.6a. The comb function has a value of 0 everywhere, except at regular intervals that correspond to the sample points, where its value is 1. The Fourier transform of a comb turns out to be just another comb with teeth at multiples of f_s (Figure 2.6b). The height of the teeth in the comb's Fourier transform is f_s in cycles/pixel [40]. Since multiplication in the spatial domain corresponds to convolution in the frequency domain, the Fourier transform of the sampled signal is obtained by convolving the Fourier transforms of the comb function and the original signal (Figure 2.6c). The result is the replicated spectrum of $f(x)$ at multiples of f_s. An insufficiently high f_s yields aliased spectra (Figure 2.6d). A sufficiently high f_s yields spectra that are replicated far apart from each other. In the limiting case, as f_s approaches infinity, a single spectrum results. The signal can be recovered from its sampled signal by multiplying the Fourier transform of the sampled signal with a (*low-pass*) filter signal that has a box shape in the frequency domain (Figure 2.6e) leaving only a single copy of the original spectrum. The result is presented in Figure 2.6f. As illustrated, due to improper sampling, high-frequency components from the replicated spectra are mixed with low-frequency components from the original spectrum leading to the aliasing phe-

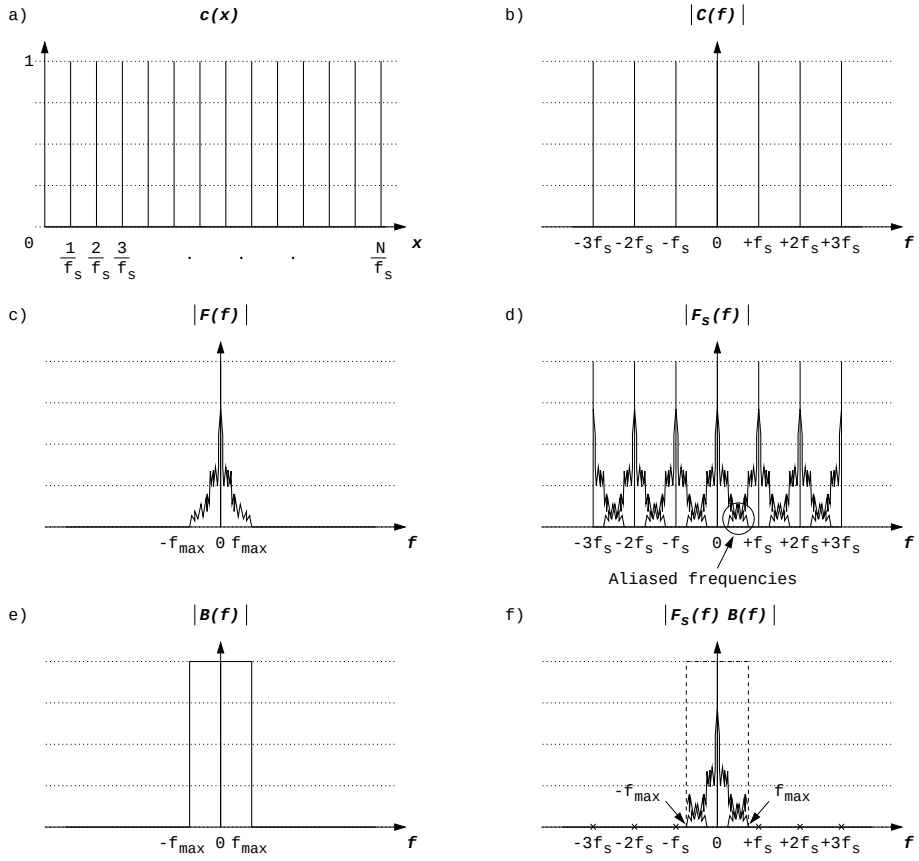

Figure 2.6: Illustration of aliasing phenomenon: a) Comb function, b) Comb's Fourier transform, c) Signal's Fourier transform, d) Sampled signal's Fourier transform, e) Ideal frequency domain box filter (ideal low-pass filter), f) Reconstructed signal's Fourier transform.

nomenon.

Remark 2.3.1: The aliasing consists of the phenomenon of high frequencies masquerading as low frequencies in the reconstructed signal, that is, the high-frequency components appear as though they were actually lower-frequency components.

Sampling theory states that a signal can be properly reconstructed from its samples if the original signal is sampled at a frequency that is greater than

twice f_{max}, the highest-frequency component in its spectrum (we will state the reason why this is not enough in practice later in the subsection). This lower bound on the sampling rate is known as the *Nyquist rate*. The Nyquist criterion can be verified graphically employing the Figure 2.6d: it can be seen that sampling at a rate greater than the Nyquist rate the multiple copies no longer overlap and this assures an alias-free spectrum of the recovered signal. Unfortunately, in computer graphics, the f_s sampling frequency is fixed at the screen resolution (sampling at every pixel or subpixel) and cannot be further increased. To make the situation worse, the object-space image of a primitive can have arbitrarily high-frequency components in its spectrum.

Remark 2.3.2: In computer graphics, it is an impossibility to generate alias-free images and still preserve all the original content of the images.

The goal of antialiasing in computer graphics is not to eliminate aliasing completely and attempt to accurately reconstruct the image on a fixed pixel grid, since that is not theoretically possible. Rather, the goal is to reduce visual artifacts caused by aliasing to an acceptable level, ideally below the threshold at which the human visual system can detect them. Some antialiasing techniques reduce artifacts by attacking the root cause of the aliasing, while others simply try to make the aliased output look better, by exploiting certain behavioral characteristics of the human visual system, e.g., *irregular* or *stochastic* sampling "cover" the aliasing with noise, making it more tolerable [52].

When attacking the root cause of aliasing from the theoretical point of view, there are two options to reduce it:

- by sampling the incoming signal, as is, at a higher rate;
- by *bandlimiting* (low-pass filtering) the incoming signal to a lower frequency range or eliminate the spectral content that is more than one half of the sample frequency, then sampling at the pixel rate of the display.

Usually, the first option cannot be put to work in practice, because the adequate sampling rate depends on the image spectral content, which is impossible to predict. Moreover, without considering the fact that is computationally expensive and costly in terms of memory to sample at higher frequencies than the pixel rate of the display, the obtained images have to be mapped back to the display by a low-pass filtering operation.

Today, all of the existing hardware solutions to mitigate the effects of aliasing rely in one way or another on the second option.

Before analyzing the most common forms of antialiasing implemented in graphics hardware, a discussion about practical low-pass filters is important, taking into account that all the methods rely on the bandlimiting approach.

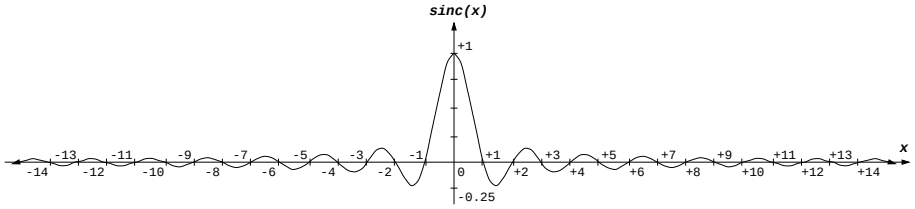

Figure 2.7: The *sinc* filter kernel corresponding to the ideal box (low-pass) filter in the frequency domain.

In Figure 2.6e, it was presented the ideal, box-shaped, low-pass, frequency domain filter. To filter a signal, the signal in the spatial domain has to be convolved (employing Equation 2.28) with the filter image in the spatial domain (this corresponds to the multiplication in the frequency domain performed in Figure 2.6f). The filter function in the spatial domain is often called the *convolution kernel* or *filter kernel*, and the size of the domain over which the filter is nonzero is known as the filter's *support*. Multiplying by a box-shaped function in the frequency domain has the same effect as convolving with the signal that corresponds to the $sinc$ function, which is defined as $sin(\pi x)/\pi x$, in the spatial domain (Figure 2.7). The $sinc$ function has the unfortunate property that is nonzero on points arbitrarily far from the origin (has infinite support and is known to belong to the infinite impulse-response, IIR, filter class). Truncating the $sinc$ function (Figure 2.8b), or *windowing*, by throwing away only those parts of the filter where the value is very small, will lead to a filter image in the frequency domain that suffers from *ringing* (or *Gibbs phenomenon*): some undesired frequency components will leak away though they should be suppressed. One final problem with the $sinc$ function, along with windowed filters derived from it, is that it has parts that dip below zero (negative lobes). When a signal is convolved with a filter that has negative lobes, the resulting signal may itself dip below zero. If the signal represents intensity values, these values correspond to unrealizable negative intensities, and must be ultimately clamped to zero.

Although windowed $sinc$ functions are useful, they are relatively expensive given the fact that the window must be relatively wide; thus, many other functions are of practical interest and are employed instead. These functions are

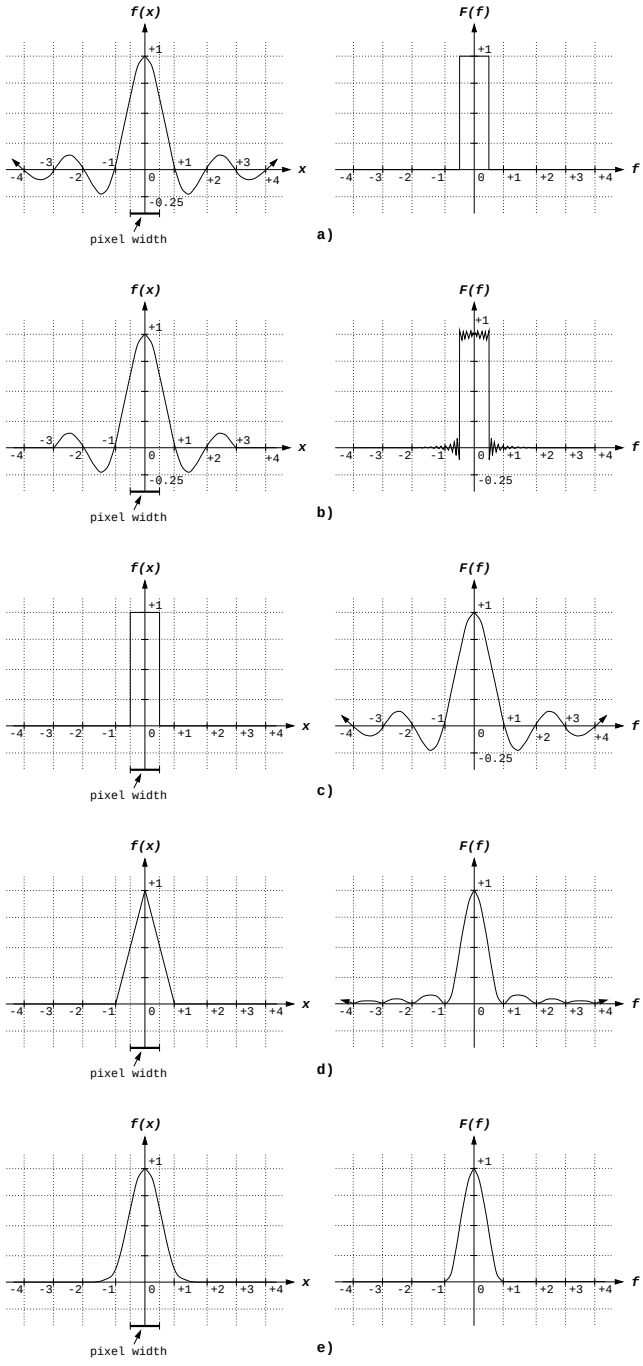

Figure 2.8: Filters in spatial and frequency domains: a) $Sinc$ filter — ideal box (low-pass) filter, b) Truncated $sinc$ filter — ringing box filter, c) Box filter — $sinc$ filter, d) Triangle (Bartlett) filter — $sinc^2$ filter, e) Gaussian filter — Gaussian filter.

only approximations of the ideal $sinc$ filter — an exact reconstruction of a signal from its samples requires the sampling frequency f_s to be pushed farther away than $2f_{max}$. Due to the limitations of the human visual system, this is unnecessary in computer graphics. The $sinc$ filter, the windowed $sinc$ filter, and alternative filters used in practice very often are presented in Figure 2.8, both in the spatial domain and the frequency domain. They are shown in the spatial domain in relation to the pixel width supposing that the sampling is undertaken at pixel rate. Thus, by supposing that the period of the sampling signal T_s is 1 pixel, then the sampling frequency $f_s = 1/T_s = 1$ cycle/pixel. This requires a filter with a cut-off frequency of $f_s/2 = 0.5$ cycles/pixel. Note that the support of most filters is wider than the pixel width and the filters are attenuating frequencies that are within the desired range (less than the cut-off frequency $f_s/2$), thus blurring the image. Also, in the particular case of the box filter with the support width equal with the pixel width (Figure 2.8c), infinitely high frequencies will leak through, making it a worse filter than the others. However, it is the lowest cost filter possible (to be used in unweighted area sampling, details supplied later), and the quality of the image provided is acceptable.

Remark 2.3.3: For a high-quality low-pass filtering operation, the convolution kernel's support has to be wider than the pixel width, with a maximum positioned at the center of the pixel subjected to the convolution integral.

Even if the kernel filter has an almost ideal characteristic, the initial sampling of the image to be low-pass filtered suffers from aliasing phenomenon most of the time (the object-space image of a primitive can have arbitrarily high-frequency components in its spectrum).

Remark 2.3.4: In practical schemes for antialiasing, there are two sources of inaccuracies that offer the aliasing phenomenon chances to leak away on the screen: the inadequate sampling rate of the image to be low-pass filtered, and the imperfect, unideal characteristic of the kernel filter that reconstructs the original image on the screen.

In conclusion, the reconstruction process of an image was reduced to the problem of convolving the presampled image (to be bandlimited) with a suitable filter and then sampling the filtered image to the screen grid. Note that filtering the presampled image anywhere but at the sample points is wasteful. Thus, in practice, the convolution integral is evaluated only at the points where presampling is performed in order to determine the final pixel value. This can be

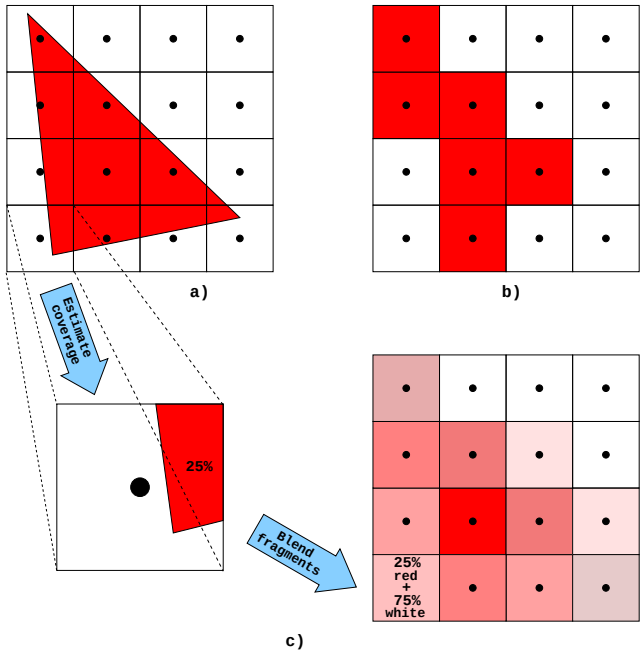

Figure 2.9: Illustration of area sampling: a) Object-space triangle superimposed on the pixel (sampling) grid, b) Image of the triangle obtained solely by point sampling, c) Image of the triangle obtained by unweighted area sampling.

reduced to the computation of a weighted sum: the sum of the intensities of the points resulted from presampling operation weighted by the coefficients contained in a discrete two-dimensional mask that mirrors the shape of the filter kernel.

Now, making use of the insights provided by the sampling theory in the context of computer graphics, the remaining part of this chapter is devoted to a brief presentation of the most common implementations of antialiasing in graphics hardware and of the trade-offs involved.

2.3.2 Antialiasing Algorithms

The two most common forms of antialiasing implemented in graphics hardware today are based on *area sampling*, point *supersampling*, or some derivative or combination of the techniques above. To illustrate the aliasing problem,

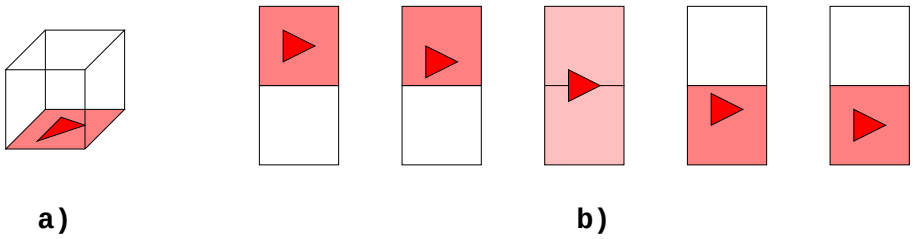

Figure 2.10: Unweighted area sampling: a) The box filter kernel: all points in the pixel are weighted equally, b) Changes in computed intensities as a small object moves between pixels.

let us follow the example in Figure 2.9. The results of a rasterization algorithm for the triangle presented in Figure 2.9a designed to fill pixels or not based strictly on *point-sampling* pixel centers are shown in Figure 2.9b. It can be seen that the resulting reconstruction of the triangle's image is poor due to the lack of compensation for aliasing. Moreover, the sub-objects falling between samples are missed completely — one can see that the right-bottom corner of the triangle is completely absent from the image.

Area Sampling Algorithm

The above problems suggest another approach: integrating the signal over a square centered about each grid point divided by the square's area (thus computing the convolution integral), and using this average intensity as that of the pixel. This technique is called *unweighted area sampling*. The array of non-overlapping squares is typically thought as representing the pixels. Each object's projection, no matter how small, contributes to those pixels that contain it, in strict proportion to the amount of the pixel's area it covers, and without regard to the location of that area in the pixel, as shown by the box filter of Figure 2.10a. As a result, scan conversion can be adapted to smooth polygon edges by approximating the area coverage of the edge over the entire pixel area, rather than just one point in the pixel, as the point-sampling method does. The process of finding the final intensity of a pixel based on the coverage information is presented in Figure 2.9c.

One drawback of the unweighted area sampling is presented in the Figure 2.10b. A small object may move freely inside the pixel, and for each position the value computed for the pixel (shown as the pixel's shade) remains

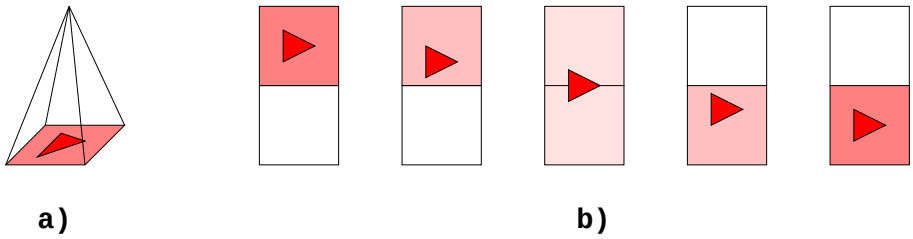

Figure 2.11: Weighted area sampling: a) The pyramidal filter kernel: points in the pixel are weighted differently, b) Changes in computed intensities as a small object moves between pixels.

the same. As soon as the object crosses over into an adjoining pixel, however, the values of both the original pixel and the adjoining pixel are affected. Thus, the object causes the image to change only when it crosses pixel boundaries. This leads to flickering as the object moves across the screen. A solution would be to allow the object's contribution to the pixel's intensity to be *weighted* by its distance from the pixel's center: the farther away it is, the less it should contribute. This suggests the use of *weighted area sampling*. The filter kernel, and how the intensity of the pixel is affected by the distance from the pixel center to the object, are presented in Figures 2.11a, and 2.11b. The weighted area sampling can be implemented starting from the unweighted area sampling algorithm with a small additional cost. In Figure 2.8, it can be seen that the filters used in area sampling are far from ideal (there are two aspects neglected here: the filter shape must have rotational symmetry and the filter support must have to overlap the adjoining pixels), but using more sophisticated filters defeats the very idea of simplicity behind the area sampling.

The primary advantages of area sampling algorithms are the following: simplicity, cost and fill-rate performance. Only one sample is usually taken per pixel, as in the case of point sampling, so the frame buffer bandwidth and storage are essentially (an additional alpha value is needed per pixel, but this is not an issue due to the fact that an OpenGL-compliant rasterization engine has to be able to support the alpha channel) not different than a point-sample approach. The controller hardware is also simple, because most controllers' scan conversion hardware already contains parameters which may be used to approximate coverage. And since frame buffer bandwidth is roughly unchanged, fill-rate performance will not necessarily be reduced when antialiasing mode is enabled [52].

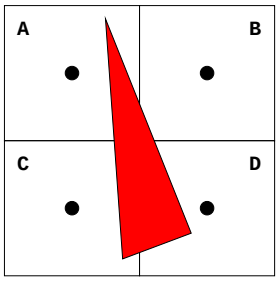

Figure 2.12: The “small triangle” problem case for area sampling.

Another advantage of area sampling is that the intensity of a pixel containing an edge changes gracefully while the edge moves through the pixel. Thus, any jumps in intensity are limited only by the precision of the blending arithmetic. This is one of the few notable advantages in quality that area sampling has over supersampling (which is discussed in more detail later).

Area sampling schemes, employed to smooth polygon edges, also have drawbacks. Sometimes, triangle size, orientation, and vertex location can make simple area sampling calculations fail or make the computation to properly approximate coverage much more complicated. Common pathological cases for simple area sampling include:

- pixels containing one or more vertices or edges of the *same* triangle, possible when rendering thin and/or small triangles;
- triangle intersections;
- pixels containing multiple edges of *different* triangles.

For instance, how will one calculate the coverage for the pixels A, C, and D of the small triangle shown in Figure 2.12? A simple algorithm or heuristic will not do an effective job estimating the coverage for such pixels, as the math is just plain difficult. Because a simple solution to the problem does not exist, the hardware will get more complicated [52].

The case of two intersecting triangles is presented in Figure 2.13. The area sampling antialiasing method works only on the polygon edges but not inside. However, intersecting triangles effectively create a high-frequency seam in the triangle interiors. Because the area sampling algorithm takes care only of one triangle at a time, it is incapable of detecting and smoothing such a seam as it

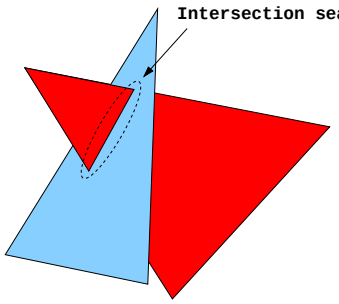

Figure 2.13: The “intersecting triangles” problem case for area sampling.

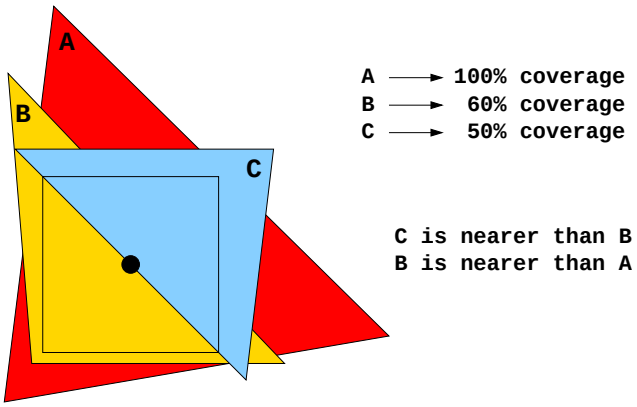

Figure 2.14: Case requiring depth-sort for area sampling.

would do for an edge. Actually, the seem will be generated by the visibility algorithm, typically Z-buffering. The seam will therefore exhibit the staircase effect due to aliasing.

The final shortcoming of the area sampling antialiasing method stems from the very nature of the *blending arithmetic* employed by the graphics rasterization algorithms. In practice, the hardware calculates the proper contribution of a fragment to the final pixel color by using the *alpha value* and the *blending hardware*. In the simple example shown in Figure 2.9, the hardware converts the extent of the estimated coverage to an alpha value, which indicates the percentage of the pixel area covered (the alpha value can also be used for transparency effects). The blending hardware calculates the final pixel color as a linear combination of the incoming fragment color (the source) and of the

already existing color in the frame buffer at the pixel location (the destination). The weighting coefficients used are derived from the alpha values associated with the source and the destination. The blending hardware employs saturated arithmetic, thus clamping the results to a maximum value when they are getting outbound due to repeated accumulations of fragments (or blending). In the scenario presented in Figure 2.14, if the triangles A, B, and C are sent in this order to the blending hardware, the final pixel will probably take the color of triangle A, though this is an incorrect result. When the fragment belonging to the triangle A is received, the pixel will be "saturated" at 100%, blocking any other future triangles from contributing to the pixel. As such, **the algorithm requires that geometry be processed front-to-back (in depth, or z value)**, in order to ensure that subsequent rejected fragments are in fact occluded. As a consequence, sending the triangles sorted in the order C, B, and A will yield the correct result: the final pixel color will be a combination of C and B triangle colors.

The antialiasing method suggested by the OpenGL specification is actually very close to the approach taken by the area sampling method. The procedure is quoted in the followings [80] (Chapter 3):

> "*Polygon antialiasing rasterizes a polygon by producing a fragment wherever the interior of the polygon intersects that fragment's square. A coverage value is computed at each such fragment, and this value is saved to be applied as described in section 3.11. (Section 3.11: Finally, if antialiasing is enabled for the primitive from which a rasterized fragment was produced, then the computed coverage value is applied to the fragment. In RGBA mode, the value is multiplied by the fragment's alpha (A) value to yield a final alpha value.) An associated datum is assigned to a fragment by integrating the datum's value over the region of the intersection of the fragment square with the polygon's interior and dividing this integrated value by the area of the intersection. For a fragment square lying entirely within the polygon, the value of a datum at the fragment's center may be used instead of integrating the value across the fragment.*"

To obtain realistic results for an antialiased scene, one has to use the procedure outlined in the OpenGL Programming Guide [93] (Chapter 6):

> "*In theory, you can antialias polygons in either RGBA or color-index mode. However, object intersections affect polygon*

antialiasing more than they affect point or line antialiasing, so rendering order and blending accuracy become more critical. In fact, they're so critical that if you're antialiasing more than one polygon, ***you need to order the polygons from front to back*** *and then use* ***glBlendFunc()*** *with GL_SRC_ALPHA_SATURATE for the source factor and GL_ONE for the destination factor. Thus, antialiasing polygons in color-index mode normally isn't practical.*

To antialias polygons in RGBA mode, you use the alpha value to represent coverage values of polygon edges. You need to enable polygon antialiasing by passing GL_POLYGON_SMOOTH to ***glEnable()****. This causes pixels on the edges of the polygon to be assigned fractional alpha values based on their coverage, as though they were lines being antialiased. Also, if you desire, you can supply a value for GL_POLYGON_SMOOTH_HINT. Now you need to blend overlapping edges appropriately. First, turn off the depth buffer so that you have control over how overlapping pixels are drawn. Then set the blending factors to GL_SRC_ALPHA_SATURATE (source) and GL_ONE (destination). With this specialized blending function, the final color is the sum of the destination color and the scaled source color; the scale factor is the smaller of either the incoming source alpha value or 1 minus the destination alpha value. This means that for a pixel with a large alpha value, successive incoming pixels have little effect on the final color because 1 minus the destination alpha is almost zero. With this method, a pixel on the edge of a polygon might be blended eventually with the colors from another polygon that's drawn later. Finally, you need to sort all the polygons in your scene so that they're ordered from front to back before drawing them.*"

Also, instead of using this method, a scene can be antialiased in OpenGL by using the accumulation buffer (described in Subsection 2.3.2), but this technique is much more computation-intensive and therefore slower.

Therefore, enforcing the sort order is required. This can impact performance if the application is not sorting as a matter of course. All in all, the area sampling antialiasing method is very appealing because of its simplicity, if one can manage to eliminate or mitigate its drawbacks.

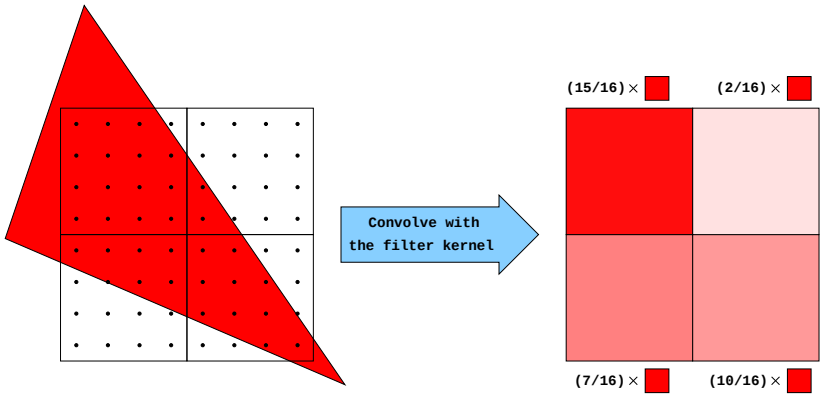

Figure 2.15: Supersampling with a 4×4 grid (assuming a box filter kernel).

Point Supersampling Algorithm

Another antialiasing method, *supersampling*, can deliver improvements in visual quality while eliminating some or all of the drawbacks associated with the area sampling algorithm, including the elimination of the requirement to presort geometry for antialiasing (however, transparency effects can be achieved only with pre-sorting). But supersampling comes at a price, including one or more of the following: higher cost, lower performance, or increased design complexity [52].

Supersampling does what its name suggests, taking multiple point samples per pixel area while each geometry primitive is rasterized. Geometry is rendered as in a conventional scheme, although at a higher spatial resolution, which translates to a higher sample frequency. When the rendering of the scene is complete, multisamples are integrated through a filter kernel (a weighted average is computed) to produce one value per pixel. Then the pixels are displayed on the screen as normal. Essentially, what supersampling does is enabling the creation of a higher-quality reconstructed image, which can be bandlimited and later re-sampled on the pixel grid.

The simplest form of supersampling is *regular* or *uniform* supersampling, which allocates a regular fixed grid of multisamples within the area of the pixel. In practice, to limit costs and to allow an acceptable level of performance, the grid contains anywhere from 4 to 16 multisamples per pixel.

Figure 2.15 illustrates an example by presenting how supersampling antialiasing method works. The filter kernel employed here, to make estimations easier,

for the convolution operation, is a box filter. The area of the pixel contains a 4×4 grid of evenly spaced sampling points. Regular sample grids are not limited to squares, though most often they are some power of 2 in height and width, for simplicity in scan conversion and in filtering.

The supersampling antialiasing method is also called *full-scene antialiasing* (FSAA), because it achieves what its name suggests. Ultimately, it provides better visual quality in comparison with the area sampling antialiasing method because it raises the "front-end" sample frequency (by sampling more points inside the pixel) and avoids many limitations of the former. Most notably, pixels with triangle intersections or triangle multiple edges are handled properly and the geometry does not need to be pre-sorted.

As it has already been mentioned, the supersampling antialiasing method comes with higher associated costs:

- increase in frame buffer memory storage requirements,
- increase in frame buffer bandwidth requirements,
- potential on-chip bottleneck in rasterization hardware.

The frame buffer storage requirements can skyrocket in a single-pass supersampling implementation. When rendering without supersampling, the hardware typically stores alpha, RGB color, and z/stencil for each pixel at a total cost of 8 bytes per pixel. In a simple supersampling scheme with 8 multisamples per pixel, the storage requirement can reach over 83 MB per buffer for an 1280×1024 true-color screen and over 124 MB for double buffer support. Taking into consideration that a graphical subsystem needs to accommodate also textures and other off-screen visuals, the memory costs involved can be prohibitive [52].

Along with the memory storage, the memory bandwidth demands to support single-pass supersampling can grow dramatically. Instead of supporting traffic for one set of parameters per pixel, the memory subsystem must support N times as much, where N is the number of the multisamples taken. Adding this to the baseline requirement for the bandwidth to sustain the screen refresh, this can quickly exceed the bandwidth available, even with today's wide buses driving high-speed DDR SDRAM memories [52].

Even if frame buffer memory consumption and bandwidth demands can be met, the bottleneck can be localized in the rasterization stage. The rasterization hardware needs to calculate parameters for N multisamples for every pixel,

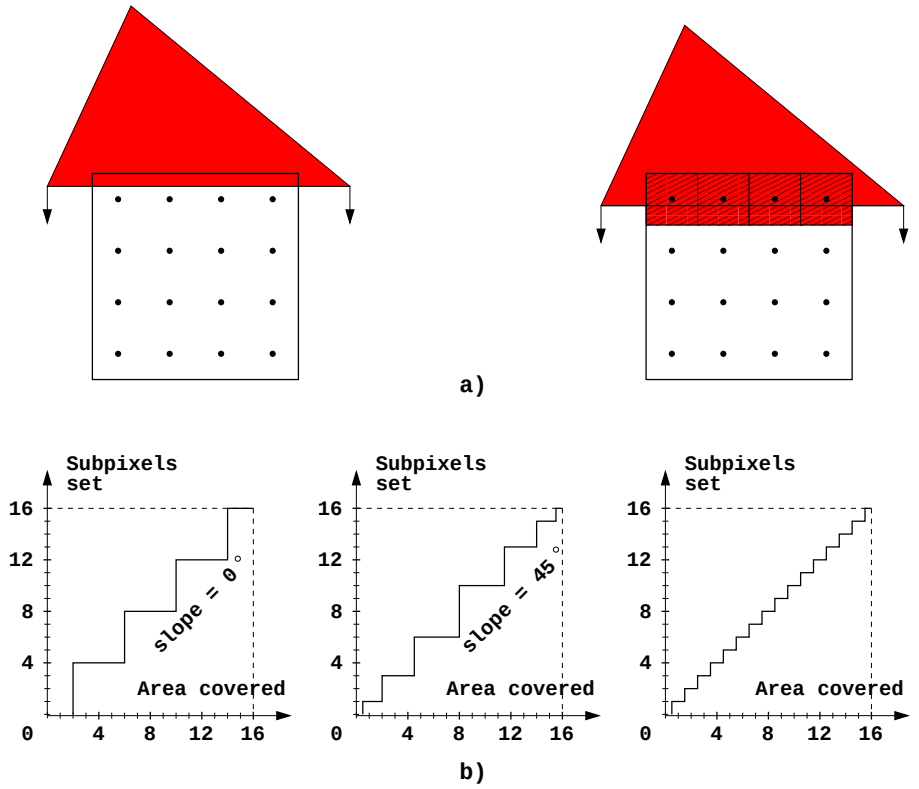

Figure 2.16: Supersampling artifacts (drawing reproduced from [78], each pixel consists of 4×4 subpixels): a) An entire row of subpixels is switched on simultaneously when a horizontal edge is moving downward over a pixel, b) The number of set subpixels versus area covered (from left to right) for the supersampling: with a horizontal (vertical) line, a diagonal line, and the ideal function.

rather than just one. In a simple implementation, rasterizing 16 multisamples may require to step through all 16 multisample points, sequentially processing all fragment parameters (screen coordinates x and y, colors R, G, B, alpha value, depth z, texture coordinates etc.). With no changes, a simple rasterizer could take 16 times as many cycles when supersampling than the normal point-sampling. To minimize hardware complexity and performance penalties, most implementations attempt one or both of the following:

- rasterize x, y, and z for multisamples in parallel with more complex

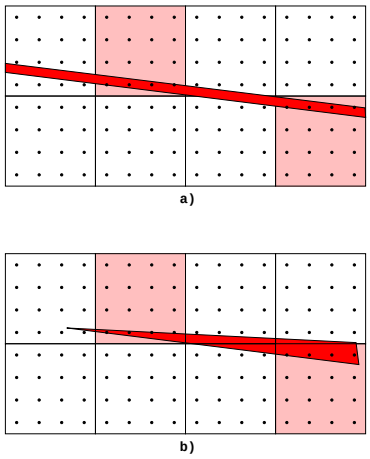

Figure 2.17: Other supersampling artifacts (drawing reproduced from [78], each pixel consists of 4×4 subpixels): a) Thin line, b) Sharp triangle. The filling color indicates the final pixel brightness.

hardware;

- perform calculations for some parameters only once per pixel rather than once per multisample.

Every supersampling implementation needs to scan convert x and y per multisample to at least determine coverage. Thus a design could limit calculations to one alpha value, RGB color, one z value, and one set of texture coordinates per pixel, for example. For most parameters besides x and y, image quality will not suffer much in calculating only one value per pixel. However, if calculating multisample z's is skipped, this can have a noticeable impact on image quality. The tradeoff in generating only one z value per pixel is that certain pathological cases, such as triangle intersections (presented in Figure 2.13) or multiple overlapping fragment edges (presented in Figure 2.14) may not be handled properly. The hardware generally relies upon z comparisons to resolve fragment. Therefore, eliminating per-multisample z makes it impossible to accurately account for contributions in such problematic cases. This is why today a host of supersampling implementations calculate unique multisample values only for x, y, and z.

The point supersampling antialiasing algorithm manifests artifacts stemming from the fact that it samples only subpixel centers to switch on or off subpixels.

Thus, this method will run into problems in certain cases. Consider for example the following case presented in Figure 2.16a. The pixels consist of 4×4 subpixels. An object with a horizontal lower edge is moving slowly downward across the pixel. Nothing happens, until the edge reaches the topmost line of the subpixel centers. As soon as the subpixel centers are reached, all the four upper subpixels are switched on at the same time. As a result, the brightness of the pixel is increased in 4 big steps instead of 16 steps (which we would like to obtain with 16 subpixels). This is shown in Figure 2.16b, where the number of subpixels that are set is plotted as a function of the exact area, covered by the triangle. The same applies for vertical lines and for diagonal lines with a slope of $45°$.

If the objects are very small (smaller than a subpixel), the effect of the errors of the supersampling approach is especially bothersome. The whole object appears and disappears again as it moves across the screen. A thin line or the end of a skinny triangle appears as a dashed line (Figure 2.17).

A way to minimize the aliases generated by regular supersampling is the use of stochastic sampling. Although the most annoying artifact (alias effect) is replaced by an artifact (noise) that is more tolerable, other objectionable effects (such as the blinking of small moving objects or holes in thin lines), are not dealt with correctly [78].

Point Supersampling Derived Algorithms

To overcome the associated costs of supersampling antialiasing method and to keep its benefits, a number of derived antialiasing approaches have been emerged:

- Accumulation Buffer
- Adaptive Supersampling
- A-Buffer
- Tiling Architectures

Accumulation Buffer Algorithm The accumulation buffer algorithm implements supersampling with moderate hardware costs [49] by comparing it with the point supersampling implementation. Beyond full-scene antialiasing, it is useful to deliver other expensive effects in hardware, such as motion blur, depth of field, and soft shadows.

The Accumulation Buffer represents an additional high-precision image buffer. Using an Accumulation Buffer, the geometry is rendered once at screen resolution for every multisample in the grid. For each pass, the sample point is moved to the next set of multisample points at another offset location within the pixel, and the image then re-rendered. After completing the rendering for each pass, the image is added ("accumulated") with a weight, pixel for pixel, to the contents of the Accumulation Buffer. When rendering is complete, the Accumulation Buffer is copied, with its values scaled back to the displayable format, to the frame buffer.

One strength of the accumulation buffer algorithm consists in the possibility of rendering a scene in a user-selectable number of passes, thus controlling gracefully the tradeoff between the quality and the performance achieved. Otherwise, the accumulation buffer is quite slow, because it has to render the same scene multiple times, once for each multisample and finally once to copy the information from the Accumulation Buffer to the frame buffer. This means that the geometry for the entire scene has to be traversed, transformed, lit and rasterized multiple times, a thing which increases the demands across the entire rendering pipeline.

The accumulation buffer algorithm is fully integrated in the OpenGL specification [80] (Chapter 4).

Adaptive Supersampling Algorithm The adaptive supersampling algorithm implements supersampling only in scene areas more prone to aliasing and point-sampling in the other regions. For instance, high-frequency polygon edges are the most logical places to supersample and polygon interiors, with more low-frequency content, can be only point-sampled.

Adaptive supersampling techniques are effective in reducing memory storage and bandwidth demands, but they are "adaptive", they depend on the image content. A pixel with many overlapped edges will consume far more memory resources than a pixel inside a single polygon. As the trend towards higher and higher geometric complexities continues, the great majority of pixels will become "difficult" pixels, and the advantages of the adaptive supersampling will wane and the technique will become less distinguishable from the brute-force full-scene supersampling [52].

A-Buffer Instead of storing the alpha, color, and z values for each multisample (as the full-scene supersampling technique does), the Carpenter's A-Buffer algorithm [19] computes only the color, the z extent (z_{min} and z_{max}), and the

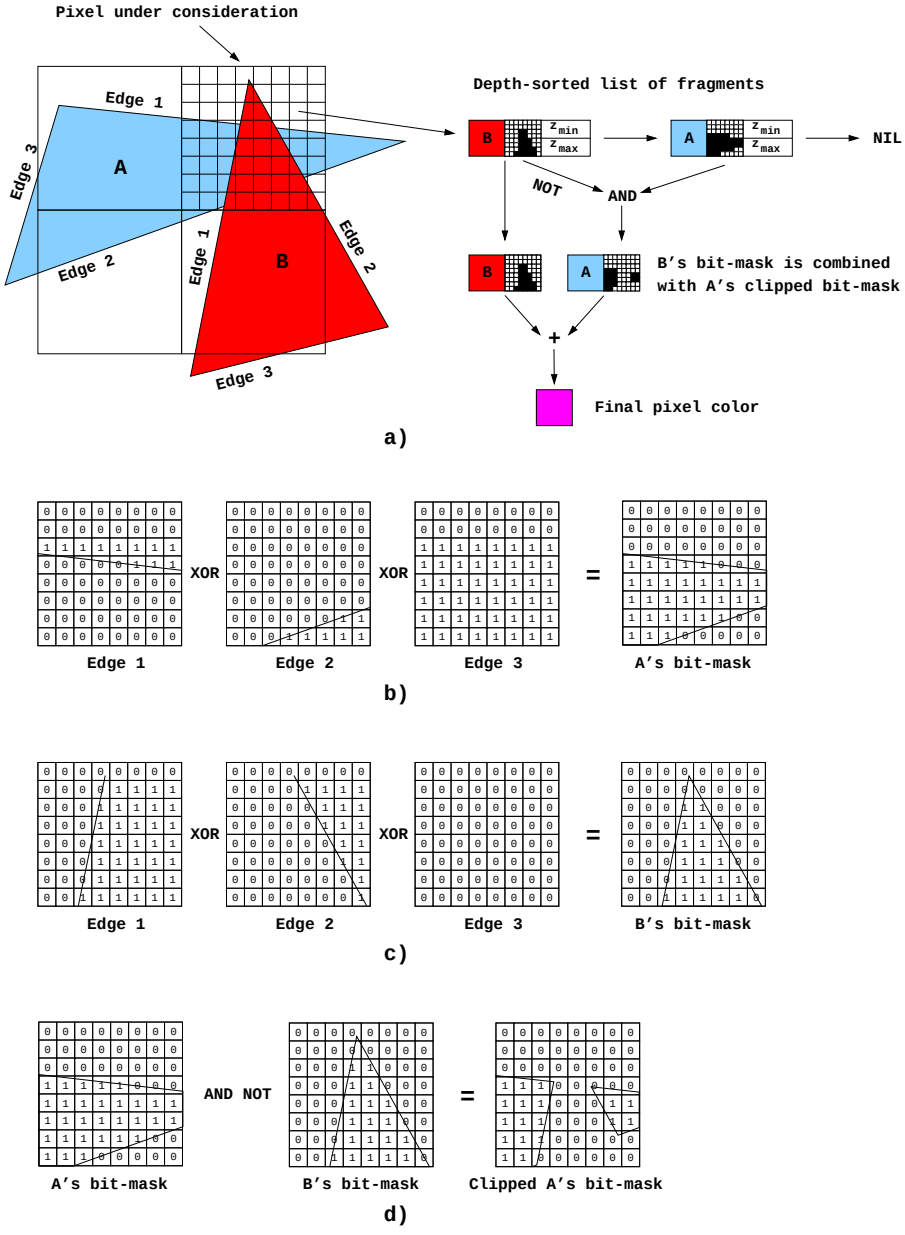

Figure 2.18: The A-Buffer algorithm: a) Overview of the algorithm for two overlapping fragments A and B, b) Computation of the bit-mask for A, c) Computation of the bit-mask for B, d) Computation of the visible bit-mask for A after depth-sorting the fragments.

multisample coverage information (as a bit-mask) of the fragment. Behind each pixel, while the polygons intersecting that pixel are processed, a list containing the above information for each polygon is stored. When all polygons affecting the pixel have been processed, the area-weighted average of the colors of the pixel's visible surfaces is obtained by selecting the fragments from the list in depth-sorted order (from front to back) and using their bit-masks to clip those of farther fragments. The algorithm is depicted in Figure 2.18a.

The bit-mask for a fragment is computed by **xor**ing together masks representing each of the fragment's edges. An edge mask has 1s on the edge and to the right of the edge in those rows through which the edge passes, as presented in Figure 2.18b and c. After the depth information is available, the bit-masks of farther fragments is clipped by those of nearer fragments (Figure 2.18d).

The A-buffer algorithm manages reasonably well with cases troublesome to simpler area sampling algorithms, such as triangle intersections (Figure 2.13). However, because it calculates only the fragment's z extent, and has no information about which part of the fragment is associated with these z values, the algorithm must make assumptions about the subpixel geometry, in cases in which fragment bit-masks overlap in z. This causes inaccuracies, especially where multiple surfaces intersect in a pixel (Figure 2.14).

Tiling Architectures *Tiling*, or *chunking* architectures were adopted by several graphics hardware vendors as a way to counteract the huge increase in storage and bandwidth requirements of full-scene antialiasing. In a tiling architecture, the screen is divided into a number of non-overlapping regions, or tiles, which are processed serially. Geometry is sorted first by screen location and dumped into one or more *bins*, one bin per tile. Geometry that overlaps a tile boundary is referenced in each tile it is visible in. When all the geometry has been specified, it is rendered from bin N to the tile N, before moving to the tile $N + 1$. As the scene is animated, the content of the bins is modified to adjust for geometry that moves from one tile to another.

The advantage of full-scene antialiasing in a tiling scheme is that the intermediate fragment values are only needed to be maintained for the tile, not for the whole screen. This limits the storage and bandwidth requirements to the number of multisamples present in the tile. And if the total number of multisamples is low enough, or if the on-chip memory is large enough, full-scene antialiasing can be achieved internally without expensive accesses to external memory buffers.

As exploited by Microsoft's Talisman rendering architecture, tiling can be used

in synergy with the A-Buffer technique to effectively "compress" tile storage requirements. Tiling requires a geometry sorting pass anyway, so sorting in z to enable support for an A-Buffer can be added more easily and not necessarily incur a performance penalty. Sorting in z allows other potential savings in bandwidth, such as limiting access to memory for z, texture, alpha and colors only for visible fragments. With the A-Buffer's compression of coverage and z information, designers can choose to reduce the on-chip storage requirements or optionally increase the supported tile size, which can yield benefits by optimizing rasterization, binning processing, and bandwidth [52].

2.4 Conclusion

In this chapter a generic 3-D graphics pipeline has been overviewed and the main operations performed were described by laying emphasis on the perspective-correct rasterization from a theoretical point of view. The operations derived here have to be implemented mandatorily, in one way or another, by every hardware rasterization engine. They are relevant because one could understand the degrees of freedom she/he has at each step in order to find how to achieve effective hardware parallelism — the equations are exploited over the next two chapters. The chapter also presented a brief description of the anti-aliasing theory and the existing hardware developments to cope with the aliasing problem. Ample references were made to the OpenGL specification (a 3-D graphics library chosen to be hardware accelerated by the present work), thus outlining the OpenGL embodiments of the theoretical aspects presented here-in.

Chapter 3

Rasterization Algorithm

In this chapter, an algorithmic view of a *potential* OpenGL-compliant tile-based hardware rasterization engine is described. By potential in this context it is meant that the proposal constitutes a platform to build on towards full OpenGL compliance, as this can be achieved only by a combination of software driver-level techniques and hardware algorithms implemented by the rasterization engine. Thus, only the algorithms implemented in hardware are discussed in this chapter and software driver-level issues that help augmenting the hardware capabilities are mentioned only when they are deemed absolutely necessary. The proposed rasterization engine is focussed on three-dimensional triangle rasterization only, as this constitutes the main operation to be performed on any rasterization engine. Consequently, the three-dimensional triangle is the centric element of the rasterization engine, since all other primitives, e.g., points, lines, and general polygons, can be reduced to triangles at the software driver-level. The described rasterization engine is capable to perform well with a multiplicity of triangle rasterization methods, e.g., filled flat- or Gouraud-shaded, both aliased or antialiased. Even though the hardware rasterization engine capabilities are somehow limited as neither textures (however, texture coordinates are interpolated), nor two-dimensional image processing operations are handled, it is constructed as an open kernel to be augmented with these and any other features, e.g., stippled and/or outlined triangles, implemented either at the software driver-level or in hardware, as resulting from time-power-area tradeoff analyses. The algorithms employed by the rasterization engine have been chosen keeping in mind several constraints: low-cost, potentially low-power, relatively high-performance, and good quality image results.

This chapter is organized as follows: in Section 3.1, the internal data formats for positional coordinates (Subsection 3.1.1), colors (Subsection 3.1.2), texture coordinates (Subsection 3.1.3), stencil and w (Subsection 3.1.4) are derived. Then, in Section 3.2, a parallel algorithm for triangle rasterization is presented focusing on: the edge functions determining the triangle stencil in the frame buffer (Subsection 3.2.1), the triangle setup stage (Subsection 3.2.2), and possible strategies for efficient triangle traversing during rasterization (Subsection 3.2.3). The algorithm described in Section 3.2 is extended to perform OpenGL compliant antialiasing in Chapter 5.

3.1 Internal Data Formats

In general, by using a floating-point format for data in computer graphics, a high-enough precision of computation is guaranteed. However, due to the fact that a lot of arithmetic operators are used in graphical algorithms, a low-cost, low-power rasterization engine cannot afford to employ in its datapath floating-point operators. Therefore, in our situation other data formats are to be sought that can manage the rasterization task with good accuracy under the low-cost, low-power constraints.

3.1.1 Positional Coordinates Data Format

Typically in 3-D graphics rendering, polygon vertices are in floating-point format after 3-D transformation and projection. Some implementations round the screen space x and y (notational convention introduced in Chapter 2, Section 2.2) floating-point coordinates to integer values, so that simple integer-based algorithms can be used to compute the triangle edges. This rounding can leave gaps on the order of half a pixel wide between adjacent triangles that do not share common vertices leading to visible artifacts like *dropouts* or *overlaps*. Such a situation arises in the case presented in Figure 3.1 [64] [74].

A common solution has been to grow triangles by at least one half of a pixel, before rounding the vertices to the nearest pixel centers. This guarantees that the rasterization algorithm will not miss any pixels at the boundary between two adjacent triangles. It also implies, however, that many of the pixels near triangle boundaries are written more than once. This generally precludes drawing triangle meshes with read/modify/write pixel operations, which are useful for blending effects such as transparency. It also does not respect the point sampling rule imposed by the OpenGL specification (the reader is again referred to Section 2.2).

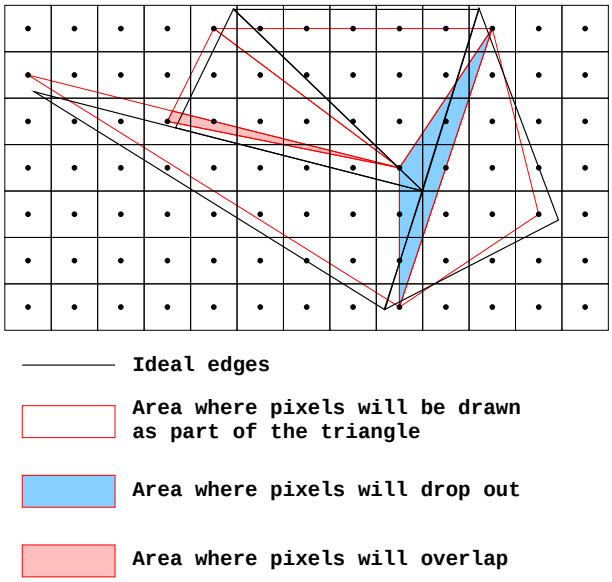

Figure 3.1: Illustration of artifacts generated by rounding floating point coordinates x and y to integer values.

Therefore, we propose the *fixed-point format* for the positional coordinates x and y of the triangle vertices. Gaps will also occur as a result of the finite precision of the fixed point format, but these gaps are much narrower and cannot be noticed. Another issue is that for any interpolator, the fractional precision used in the iterations to rasterize a triangle must be chosen to give an acceptable error across the interpolation. Another aspect concerns the number of bits required to represent the whole part of these fixed-point numbers. The OpenGL specification [80] (Chapter 2) guarantees that the positional coordinates of the triangle vertices received by the rasterization engine will be entirely contained in the screen area, so there is no need to be concerned about negative positional coordinates or to represent their whole part with more bits than it takes to represent the pixel coordinates on the screen.

The assumed coordinates system was presented in Figure 2.5. The basic concepts of tiling architectures were briefly presented in Subsection 2.3.2. If a tiling architecture, as is the case for our architecture, is employed with the tile size of $2^m \times 2^n$ pixels, and the screen partitioned in $2^u \times 2^v$ tiles, then x and y coordinates for a triangle vertex in the screen space will be represented in unsigned two's complement fixed-point format in the following form:

Screen resolution	
$W \times H$	$W \times H$ (prime factorization)
320×240	$2^6 \cdot 5 \times 2^4 \cdot 3 \cdot 5$
640×480	$2^7 \cdot 5 \times 2^5 \cdot 3 \cdot 5$
800×600	$2^5 \cdot 5^2 \times 2^3 \cdot 3 \cdot 5^2$
1024×768	$2^{10} \times 2^8 \cdot 3$
1152×864	$2^7 \cdot 3^2 \times 2^5 \cdot 3^3$
1280×960	$2^8 \cdot 5 \times 2^6 \cdot 3 \cdot 5$
1280×1024	$2^8 \cdot 5 \times 2^{10}$

Table 3.1: Typical screen resolutions and their prime factorization.

$$x = (X_{u-1}X_{u-2}\ldots X_2X_1X_0x_{m-1}x_{m-2}\ldots x_2x_1x_0x_{-1}x_{-2}\ldots x_{-k})_2$$

and

$$y = (Y_{v-1}Y_{v-2}\ldots Y_2Y_1Y_0y_{n-1}y_{n-2}\ldots y_2y_1y_0y_{-1}y_{-2}\ldots y_{-k})_2$$

with k digits for the fractional part. Having a tile with dimensions represented as powers of two pixels simplifies considerably the datapath design.

If the *tile index and offset in the x direction* can be defined as:

$$TIDX = (X_{u-1}X_{u-2}\ldots X_2X_1X_0)_2$$

$$OFFX = (x_{m-1}x_{m-2}\ldots x_2x_1x_0x_{-1}x_{-2}\ldots x_{-k})_2$$

and, in a similar manner, the *tile index and offset in the y direction* as:

$$TIDY = (Y_{v-1}Y_{v-2}\ldots Y_2Y_1Y_0)_2$$

$$OFFY = (y_{n-1}y_{n-2}\ldots y_2y_1y_0y_{-1}y_{-2}\ldots y_{-k})_2$$

then x and y vertex coordinates in the screen space can be written as:

$$x = TIDX \times 2^m + OFFX \tag{3.1}$$

$$y = TIDY \times 2^n + OFFY \tag{3.2}$$

In other words, if the tile where the vertex is positioned in is known, and also if the offset this vertex takes inside the tile is known, then the position of the vertex on the screen can be found by simply concatenating its tile index and offset.

To search for the ”right” tile dimensions, Table 3.1 presents several common screen resolutions. By using their prime factorization, one can see that for a given screen resolution there are a limited number of choices for the tile size.

Actually, one can employ an arbitrary (power of two) tile size. In this case, either the resulting screen resolution will be atypical, or a small part of the frame buffer will not be mapped on the screen and will be wasted. The ultimate tile size will be established after considering a trade-off of several factors: the bandwidth overhead sending triangles multiple times, the associated storage cost with the tile processing (both on the host processor and in the rendering engine), and the tile rasterization engine datapath width.

The z positional coordinate of a vertex in screen space is specified by OpenGL [80] (Chapter 2) as a floating-point value in the interval $[0, 1]$, which can be converted for internal use in an *integer format*. Thus the integer representation of z with l bits represents each value $p/(2^l - 1)$, where $p \in \{0, 1, \ldots, 2^l - 1\}$, as p (e.g., 1.0 is represented in binary as a string of all ones). However, an extra two bits (one for the sign) will be allocated to ensure that the antialiasing algorithm is working properly. The reason will be explained in Chapter 5, Section 5.4. The positional coordinate z of the fragments generated in a tile will be stored (eventually, because the fragments may not "survive" long enough) in the rasterization engine's *depth buffer*. The depth buffer is used usually for primitive occlusion testing, but other uses can also be envisioned. The depth buffer size is in this case $2^m \times 2^n$ words of $l + 2$ bits.

The depth buffer wordlength $l + 2$ is important. Due to the fact that the perspective transformation distorts distances (distances equal in eye space are not equal any more in screen space and vice-versa), the viewer cannot distinguish in the case of distant objects which one is actually closer. The transformation of z coordinate from the screen space (where the interpolation is performed) to the eye space is the following [15]:

$$z_e = \frac{z_{e_{far}}}{\frac{z_{e_{far}}}{z_{e_{near}}} - z \cdot (\frac{z_{e_{far}}}{z_{e_{near}}} - 1)} \tag{3.3}$$

where $z_{e_{near}}$ and $z_{e_{far}}$ ($z_{e_{near}} < z_{e_{far}}$) are two constants in the eye space used to bracket a scene of objects subjected to the perspective transformation. Defining the notation $r = z_{e_{near}}/z_{e_{far}}$, the distortion between the two spaces is presented in Figure 3.2. The values that can be represented in screen space are equally spaced along the z axis (Figure 3.2a). Mapping these points in the eye space leads to the results shown in Figure 3.2b, c, and d — it can be seen that the distortion is increasing as r is getting smaller. In fact, this distortion can be perceived as a loss of resolution in the depth buffer. In this way, roughly $log_2(\frac{1}{r})$ bits of depth buffer precision are lost [83]. It was proved in [15] that if $r > 0.3$ most of the range of the depth buffer would be used. This means that the depth buffer wordlength $l + 2$ has to be chosen differently

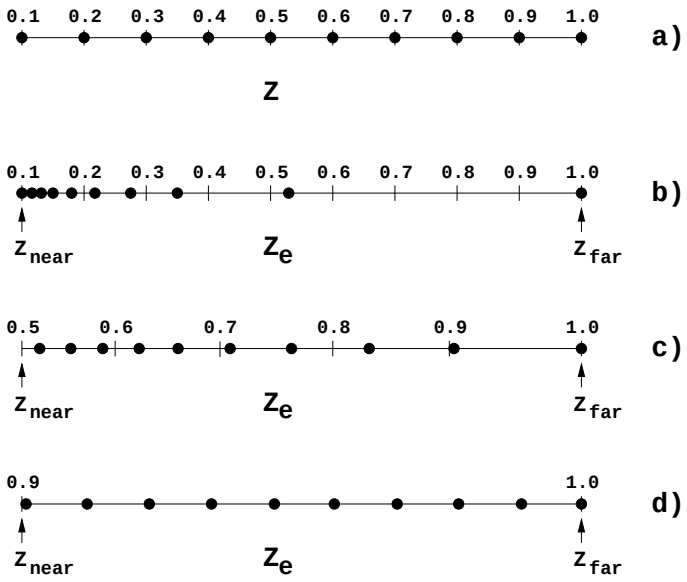

Figure 3.2: Distortion seen when points are transformed from the screen space in the eye space along the z axis: a) Ten equally spaced points in the screen space, and their mapping in the eye space when b) $r = 0.1$, c) $r = 0.5$, d) $r = 0.9$.

based on the targeted application: for open environments with large dynamic range $1/r$, e.g., flight simulators and games, $l+2$ has to be greater, for compact environments with small dynamic range, e.g., CAD programs, a smaller $l + 2$ will suffice. High-end graphics accelerators usually support depth buffers with 24 or 32 bits; low-cost graphics accelerators may store depth values with 16, 20 or 24 bits [63].

3.1.2 Color Data Format

The rasterization engine will be compliant only with the RGBA mode of OpenGL, the palletized colors being a deprecated feature. A general view of the way in which OpenGL handles colors will be presented before discussing data formats.

In OpenGL, the state maintained per vertex contains the *primary* and *secondary colors* for the *front-* and *back-side* of the primitive. These associated colors are either based on the *current color* (a global OpenGL state variable), or produced in the graphical pipeline stages, e.g., the lighting stage, before the

rasterization stage [80] (Chapter2).

The primary and secondary colors are a consequence of the lighting operations. Lighting may be in one of the two states: off or on. When the lighting is off, the current color is assigned to the vertex primary color. The secondary color is $(0, 0, 0, 0)$. When the lighting is on, the vertex primary and secondary colors are computed from the current lighting parameters [80] (Chapter 2). Then, the results are transferred to the rasterization stage. Next, if texturing is enabled, the primary color will be further modified by a texture. Then the two colors, the primary and the secondary, are added together to produce a single post-texturing RGBA color, and the result will be clamped to the range $[0, 1]$ [80] (Chapter 3). After that, fog effects are applied to the fragment. This means the blending of a fog color with a rasterized fragment's post-texturing color using a blending factor f. The factor f computation requires an exponentiation to be performed on the eye-space z_e coordinate of every fragment center [80] (Chapter 3), which is an expensive computation. If this computation is performed at the software driver level for every vertex of the triangle and the results are clamped in the range $[0, 1]$ and transformed in a fixed-point format similar with that employed for the color components, and then sent as the other vertex attributes are to the rasterization engine, the fog blending factor f can be approximated by interpolation across the triangle. Finally, if antialiasing is enabled, it is applied as described in Subsection 2.3.2, and the resulting fragment is sent further to the per-fragment operation stage of the rasterization engine (for pixel ownership test, scissor test, alpha test, stencil test, depth buffer test, blending, dithering, and other logical operations), before being written in the rasterization engine's *frame buffer* or discarded.

The selection between the back color and the front color depends on the primitive of which the vertex being lit is a part. If the primitive is a point or a line segment, the front color is always selected. If it is a polygon, then the selection is based on the user's chosen orientation of the edges of the polygon (clockwise or counter-clockwise), in the screen space to be considered front-facing (the vertices of a polygon are always enumerated in a manner that leads to oriented edges). The method suggested by the OpenGL specification [80] (Chapter 2) to determine the orientation of a polygonal face is to compute its signed area in the screen space and render a decision based on its sign. Signed area computation can be implemented either in software or in hardware. It is true that only one face of a triangle will be visible at a time, thus if the signed area of the triangle is computed at the software driver level, only the visible color will be sent with the vertices for the triangle. Otherwise, the vertices of the triangle will be sent with both the front and back colors and the rasteriza-

tion engine will perform the selection of the color to be applied to the visible face. The two options will be investigated because there are tradeoffs involved.

For an RGBA color, each color component specified as a floating-point value in the interval $[0, 1]$ in the upper graphical pipeline stages of OpenGL will be converted for the internal use of the rasterization engine into an *integer format* with e bits. Thus, the integer representation of a color component with e bits represent each value $f/(2^e - 1)$, where $f \in \{0, 1, \ldots, 2^e - 1\}$, as f (e.g. 1.0 is represented in binary as a string of all ones)[80] (Chapter 2). However, an extra two bits (one for the sign) will be allocated to ensure that the antialiasing algorithm is working properly. The reason will be explained in Chapter 5, Section 5.4.

The RGBA color for the pixels of a tile will be stored in the rasterization engine's frame buffer. In the simplest case, the rasterizer's internal data format for a color component is chosen to represent also that color component in the frame buffer (clamped to the positive range representable with e bits). In this case, the frame buffer size is $2^m \times 2^n$ words of e bits. Otherwise, the frame buffer will be smaller (because the internal format employed is wider to give an acceptable error across the interpolation).

3.1.3 Texture Coordinates Data Format

Texture mapping is a technique that applies a texture onto an object's surface as if the texture were a decal or cellophane shrink-wrap. In OpenGL, a *texture* is a one-, two-, or pseudo-three (a stack of two-dimensional images) dimensional image and a set of parameters that determine how samples are derived from the image. The texture mapping is applied for each primitive for which texturing is enabled. This mapping is accomplished by using the color of the image at the location indicated by a fragment's non-homogeneous $(s/q, t/q, r/q)$ coordinates to modify the fragment's primary RGBA color. Texturing does not affect the secondary color [80] (Chapter 3). After the texturing operation is accomplished, the texture coordinates of the fragment are discarded from the fragment's associated data.

The homogeneous texture coordinates (s, t, r, q) internal data format will be a *fixed-point format*. The texture coordinates will be translated in a fixed range (taking into account the texture unit's current mode of operation) at the software driver level first, to use at its best the limited precision of the internal format employed.

3.1.4 Miscellaneous Data Formats

It is customary when the wordlength of the depth buffer is specified, a supplementary number of bits to be taken into account for stenciling operations. These additional bits will be grouped as an *unsigned word* element in a $2^m \times 2^n$ matrix (with the same size as the tile) on the rasterization engine called the *stencil buffer*. The stencil buffer is employed in synergy with the stencil test and depth test. The stencil test conditionally discards a fragment based on the outcome of a comparison between the value in the stencil buffer at the location (x, y) and a reference value, and then updates if required that location in the stencil buffer based on the depth test result. The stenciling technique is used most often to mask arbitrary regions with arbitrary forms to be affected by the drawing operations on the screen. The stencil test is part of the per-fragment operations category of OpenGL [80] (Chapter 4). The wordlength of the stencil buffer has to be chosen based on the desired precision of the graphical algorithms that make use of the stencil buffer.

As it was presented in Chapter 2.2, the colors and the texture coordinates associated with a fragment make extensive use in their interpolation formulas of the reciprocal of the post-perspective transformation homogeneous components w of the vertices of the triangle. Due to the fact that the w component is a scaled value of the eye space z_e component (as it can be inferred by corroborating the formulas from [80] (Chapter 2)), it is assumed that w's values can vary in a wide range. Therefore, it would be better for the limited width fixed-point datapath of the rasterization engine to receive instead a scaled value of $1/w$ in the range $[0, 1]$ with the only restriction that the same scaling factor would be applied for the three vertices of a triangle. Thus, for a triangle, the three scaled $1/w$ values would be represented in the same format as the screen space z positional coordinate with $g + 2$ bits.

3.2 An Algorithm for Triangle Rasterization

Traditionally, a triangle is rasterized by computing its edges with a line interpolation algorithm, each scan line between the edges being filled with interpolated z, color, and texture coordinates values [40]. Even though conceptually simple, this algorithm cannot be parallelized easily to become suitable for hardware implementation (meaning versatility). Moreover, if antialiasing is desired, it can be achieved without complications only by supersampling techniques, which are expensive (for the traditional rasterization algorithm, a difficult case for antialiasing based on area sampling was presented in Fig-

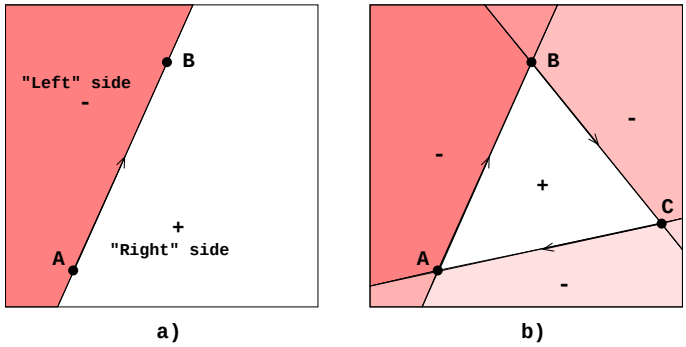

Figure 3.3: Triangle representation using edge functions: a) The oriented segment AB subdivides the plane in two half-planes with opposite edge function sign, b) The interior of the triangle is formed by the union of right sides of AB, BC, and CA.

ure 2.12) [52][74].

The algorithm adopted by us for hardware triangle rasterization was introduced by Pineda [74]. The algorithm is inherently parallel, so that the rendering performance is memory bandwidth limited, rather than computation limited. The algorithm represents each edge of a triangle by a linear edge function that has a value greater than zero on one side of the edge and less than zero on the opposite side. For points lying exactly on the edge, the edge function returns a zero value. A point belongs to the interior of a triangle if all its edge functions computed for that point have the same sign. The representation of a triangle using the edge functions is depicted in Figure 3.3.

To rasterize a triangle, the exact values for the edge functions, z, colors, and texture coordinates are computed for a conveniently chosen pixel (x, y) on the screen, and also interpolation steps along the x and y axes are found for them. This stage is called the *triangle setup stage*. Then, the values for the adjacent pixels can be computed by simple linear interpolators that require only one addition per component per iteration. The values for the edge functions will be used as a ”stencil” that allows a pixel to be modified only if it is interior to the triangle. Thus, the process of rasterizing the triangle can be reduced to an algorithm that traverses any area that includes the interior of the triangle. The particular order of traversal will not be important for correctness, only that each interior pixel will have to be covered once and only once (of course, there are triangle traversal algorithms with various degrees of efficiency).

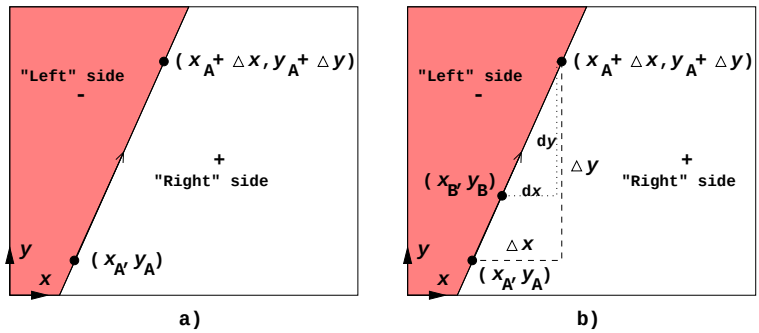

Figure 3.4: Notational conventions for the edge function: a) The edge is defined by a vector starting at the point A with the slope $\Delta y/\Delta x$, b) Any alternative point B on the edge line can be considered instead of point A in the edge equation.

3.2.1 The Edge Function

Considering the vector defined by the points (x_A, y_A) and $(x_A+\Delta x, y_A+\Delta y)$ (shown in Figure 3.4a), the edge function for a certain point at (x, y) position can be defined as:

$$\boxed{E(x,y) = (x - x_A) \cdot \Delta y - (y - y_A) \cdot \Delta x} \tag{3.4}$$

This function has the useful property that its value is related to the position of the point (x, y) relative to its associated line defined by the points (x_A, y_A) and $(x_A + \Delta x, y_A + \Delta y)$:

$E(x, y) > 0$ if (x, y) is to the "right" side of the line;

$E(x, y) = 0$ if (x, y) is exactly on the line;

$E(x, y) < 0$ if (x, y) is to the "left" side of the line.

Two additional properties are worth to be taken into consideration. It can be shown that the *edge function is invariant with the origin of its defining vector along its associated line.* Also, it can be shown that the *edge function is only sign invariant with the length of its defining vector along its associated line*. To prove these statements, using similar triangles in Figure 3.4b, it can be written that:

$$\frac{\mathrm{d}x}{\Delta x} = \frac{\mathrm{d}y}{\Delta y} \tag{3.5}$$

which implies that:

$$(x_A + \Delta x - x_B) \cdot \Delta y = (y_A + \Delta y - y_B) \cdot \Delta x \tag{3.6}$$

Now, using Equation (3.6), Equation (3.4) can be rewritten as:

$$\begin{aligned} E(x,y) &= (x - x_A) \cdot \Delta y - (y - y_A) \cdot \Delta x \\ &= (x - x_A) \cdot \Delta y + (x_A + \Delta x - x_B) \cdot \Delta y \\ &\quad -(y - y_A) \cdot \Delta x - (y_A + \Delta y - y_B) \cdot \Delta x \\ &= (x - x_B) \cdot \Delta y + \Delta x \cdot \Delta y - (y - y_B) \cdot \Delta x - \Delta y \cdot \Delta x \\ &= (x - x_B) \cdot \Delta y - (y - y_B) \cdot \Delta x \end{aligned} \tag{3.7}$$

and this shows that the edge function will have the same value for a vector defined by the points (x_B, y_B) and $(x_B + \Delta x, y_B + \Delta y)$, with x_B being on the associated line of the original vector (actually, a translated version of the original vector along its associated line).

To prove the second property, plugging Equation (3.5) into Equation (3.4) yields:

$$\begin{aligned} E(x,y) &= (x - x_A) \cdot \Delta y - (y - y_A) \cdot \Delta x \\ &= (x - x_A) \cdot \frac{\mathrm{d}y \cdot \Delta x}{\mathrm{d}x} - (y - y_A) \cdot \Delta x \\ &= \frac{\Delta x}{\mathrm{d}x} \cdot [(x - x_A) \cdot \mathrm{d}y - (y - y_A) \cdot \mathrm{d}x] \\ &= \frac{\Delta x}{\mathrm{d}x} \cdot \widetilde{E}(x,y) \end{aligned} \tag{3.8}$$

which shows that the edge functions $E(x,y)$, for the vector $\overrightarrow{(x_A, y_A)(x_A + \Delta x, y_A + \Delta y)}$, and $\widetilde{E}(x,y)$, for the vector $\overrightarrow{(x_A, y_A)(x_A + \mathrm{d}x, y_A + \mathrm{d}y)}$ sharing the same associated line, will behave identically with respect to the sign but will have different magnitudes.

From the computational point of view, the most notable property is that the edge function can be computed incrementally by simple addition for adjacent pixels, as is the case for any rasterization algorithm:

$$\boxed{E(x+1, y) = E(x,y) + \Delta y} \tag{3.9}$$

$$\boxed{E(x, y+1) = E(x,y) - \Delta x} \tag{3.10}$$

The proof of Equations (3.9) and (3.10) is trivial with proper substitutions in Equation (3.4).

Generalizing, the edge function for a pixel with the address $(x + \delta x, y + \delta y)$ can be computed from its value for the pixel (x, y) as:

$$\boxed{E(x + \delta x, y + \delta y) = E(x, y) + \delta x \cdot \Delta y - \delta y \cdot \Delta x} \tag{3.11}$$

by supplementarily using two multiplications and one addition.

Actually, there is a large degree of freedom in implementing the edge function computation for a triangle. Implementations can range from computing the three edge functions in a pipelined fashion sharing the same hardware for the edge function computation, up to computing them in a completely parallel fashion using separate hardware for every edge function. At the same time, parallelizing a single edge function computation is possible. By providing L hardware interpolators for the same edge function, each interpolator can compute the edge function in an interleaved fashion, for pixels a distance L away from a given pixel with the address $(x + i, y),\ i \in \{0, 1, \ldots, L - 1\}$:

$$E(x + i + L, y) = E(x + i, y) + L \cdot \Delta y \tag{3.12}$$

From a similar freedom in implementation may benefit also other functions computed incrementally like the ones for the z, color, and texture coordinates interpolation.

3.2.2 Triangle Setup Stage

To rasterize a triangle, the exact values for the edge functions, z, colors, and texture coordinates are computed for a conveniently chosen pixel (x, y) on the screen and also interpolation steps along the x and y axes are found for them. This stage is called the *triangle setup stage*. Then, the values for the adjacent pixels can be computed by simple linear interpolators that require only one addition per component per iteration.

The Edge Function Setup

As it was said before, a point belongs to the interior of a triangle if all of its edge functions computed for that point have the same sign. Considering the triangle with oriented edges represented in Figure 3.3b, the triangle setup for the three edge functions can be written as:

$$
\begin{aligned}
\Delta x_{AB} &= x_B - x_A \\
\Delta y_{AB} &= y_B - y_A \\
\Delta x_{BC} &= x_C - x_B \\
\Delta y_{BC} &= y_C - y_B \\
\Delta x_{CA} &= x_A - x_C \\
\Delta y_{CA} &= y_A - y_C \\
x_{init_A} &= x_{init} - x_A \\
y_{init_A} &= y_{init} - y_A \\
x_{init_B} &= x_{init} - x_B \\
y_{init_B} &= y_{init} - y_B \\
x_{init_C} &= x_{init} - x_C \\
y_{init_C} &= y_{init} - y_C \\
E_{AB}(x_{init}, y_{init}) &= x_{init_A} \cdot \Delta y_{AB} - y_{init_A} \cdot \Delta x_{AB} \\
E_{BC}(x_{init}, y_{init}) &= x_{init_B} \cdot \Delta y_{BC} - y_{init_B} \cdot \Delta x_{BC} \\
E_{CA}(x_{init}, y_{init}) &= x_{init_C} \cdot \Delta y_{CA} - y_{init_C} \cdot \Delta x_{CA}
\end{aligned}
\tag{3.13}
$$

where the pixel having the *screen* coordinates (x_{init}, y_{init}) represents an initialization point, inside the current processed tile (it may be the pixel with the *tile* offset $(0, 0)$), for the edge functions (more details are provided in Subsection 3.2.3). After the setup stage, the edge functions will be computed incrementally and their signs will determine the pixel relationship with the triangle (interior, exterior, or on the border). Note that all the operands presented in Equation (3.13) are signed.

The above boxed equations will remain valid if they are written for any cyclic permutation σ of vertex indices $\langle A, B, C \rangle$. These alternative reformulations of the equations may prove useful when algorithmic optimizations for triangle meshes (triangle strips, triangle fans) are to be sought in the near future.

As a positive side effect of a triangle being represented with three edge functions, the hardware required to evaluate an edge function can be also used to compute the signed area of a triangle. The signed area of a triangle is used by OpenGL to perform triangle face determination. The signed area of a triangle is defined as the value of the projection on the z axis (in the screen space) of the cross-product of the vectors associated with the triangle's edges (Figure 3.3b):

$$
A = \frac{1}{2} \cdot (\overrightarrow{AB} \times \overrightarrow{BC}) \cdot \overrightarrow{\mathbf{e_z}} = \frac{1}{2} \cdot (\overrightarrow{BC} \times \overrightarrow{CA}) \cdot \overrightarrow{\mathbf{e_z}} = \frac{1}{2} \cdot (\overrightarrow{CA} \times \overrightarrow{AB}) \cdot \overrightarrow{\mathbf{e_z}} \tag{3.14}
$$

where unit vectors $\overrightarrow{\mathbf{e_x}}$, $\overrightarrow{\mathbf{e_y}}$, $\overrightarrow{\mathbf{e_z}}$ are the basis of the vector screen space presented in Figure 2.5.

Edge	Quadrant				Boundary fragment (x, y)
	I	II	III	IV	generated if $E(x, y) \equiv 0$?
$\Delta x > 0, \Delta y \geq 0$	$\surd$	-	-	-	Yes
$\Delta x \leq 0, \Delta y > 0$	-	$\surd$	-	-	Yes
$\Delta x < 0, \Delta y \leq 0$	-	-	$\surd$	-	No
$\Delta x \geq 0, \Delta y < 0$	-	-	-	$\surd$	No

Table 3.2: Formal assignment of oriented edges to quadrants based on the edge factors Δx and Δy, and the point sampling rule for fragment centers that lie on an edge (on the triangle's boundary) based on the quadrant that owns the edge.

For example,

$$\begin{aligned}
\overrightarrow{AB} \times \overrightarrow{BC} &= \begin{vmatrix} \overrightarrow{\mathbf{e_x}} & \overrightarrow{\mathbf{e_y}} & \overrightarrow{\mathbf{e_z}} \\ x_B - x_A & y_B - y_A & 0 \\ x_C - x_B & y_C - y_B & 0 \end{vmatrix} \\
&= [(x_B - x_A)(y_C - y_B) - (x_C - x_B)(y_B - y_A)] \cdot \overrightarrow{\mathbf{e_z}} \\
&= [\Delta x_{AB} \cdot (y_C - y_A - \Delta y_{AB}) \\
&\quad -(x_C - x_A - \Delta x_{AB}) \cdot \Delta y_{AB}] \cdot \overrightarrow{\mathbf{e_z}} \\
&= [\Delta x_{AB} \cdot (y_C - y_A) - (x_C - x_A) \cdot \Delta y_{AB}] \cdot \overrightarrow{\mathbf{e_z}} \\
&= -[(x_C - x_A) \cdot \Delta y_{AB} - (y_C - y_A) \cdot \Delta x_{AB}] \cdot \overrightarrow{\mathbf{e_z}} \\
&= -E_{AB}(x_C, y_C) \cdot \overrightarrow{\mathbf{e_z}} \qquad (3.15)
\end{aligned}$$

The other cross-products yield similar results:

$$\overrightarrow{BC} \times \overrightarrow{CA} = -E_{BC}(x_A, y_A) \cdot \overrightarrow{\mathbf{e_z}} \qquad (3.16)$$

and

$$\overrightarrow{CA} \times \overrightarrow{AB} = -E_{CA}(x_B, y_B) \cdot \overrightarrow{\mathbf{e_z}} \qquad (3.17)$$

Substituting the result of Equations (3.15), (3.16), and (3.17) in Equation (3.14), the signed area of the triangle can be expressed as:

$$A = -\frac{1}{2} \cdot E_{AB}(x_C, y_C) = -\frac{1}{2} \cdot E_{BC}(x_A, y_A) = -\frac{1}{2} \cdot E_{CA}(x_B, y_B) \qquad (3.18)$$

Several things that need further clarifications are presented.

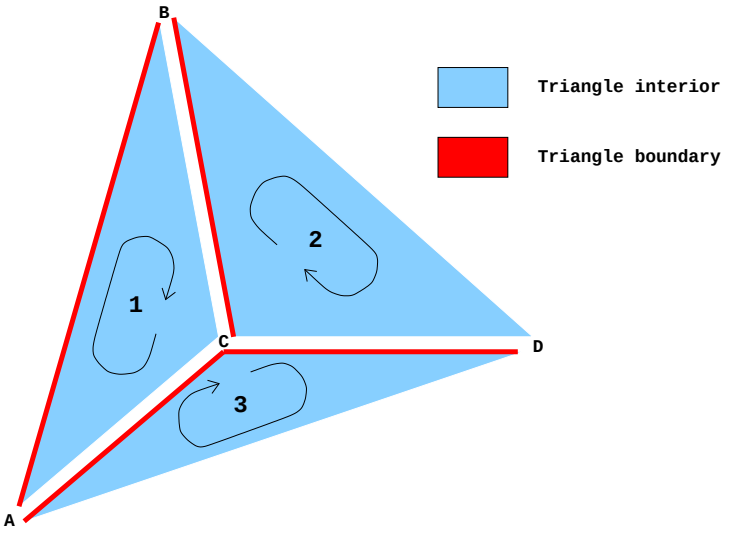

Figure 3.5: Rasterization of a triangle mesh following the point sampling rule convention of Table 3.2.

First, the polygon point sampling rule of OpenGL (presented in Chapter 2.2) will be implemented as in the sequel. For the discussion, it is assumed that a triangle's oriented edge can be represented as a vector from the source vertex to the sink vertex. Next, each such represented edge can be placed in a local coordinates system as presented in Figure 3.4 with its source vertex positioned at the origin of this coordinates system. Each edge then will belong to a specific quadrant of space. Thus, an edge can be classified based on the sign of its Δx and Δy as a quadrant one edge, quadrant two edge, quadrant three edge, or quadrant four edge (horizontal and vertical edges are also classified in one of the four previous categories). Then, the implementation will rasterize fragments whose centers lie on quadrant one edges and quadrant two edges, and will not rasterize fragments whose centers lie on quadrant three edges and quadrant four edges. This will guarantee that the polygon point sampling rule will be satisfied. The assignment of the oriented edges in quadrants, and the point sampling rule for fragment centers that lie on a given edge (on the triangle's boundary) based on the quadrant that owns the edge, are presented in Table 3.2. An example of rasterization with the point sampling rule implemented using the above convention is presented in Figure 3.5. We would like to mention once again that the point sampling rule is enforced only for aliased (point-sampled) triangles.

glFrontFace argument	$E_{AB}(x_C, y_C)$	Selected Colors	Remarks
GL_CW	< 0	Back	The back colors are selected
	$\equiv 0$	don't care	The triangle is degenerate
	> 0	Front	The front colors are selected
GL_CCW	< 0	Front	The front colors are selected
	$\equiv 0$	don't care	The triangle is degenerate
	> 0	Back	The back colors are selected

Table 3.3: The selection of the color for the visible face of a triangle described with edge functions.

Second, as it has been already mentioned, the selection between the back color and the front color depends on the user's chosen orientation of the edges of the triangle (clockwise or counter-clockwise) in the screen space to be considered front-facing. In OpenGL this is specified as an argument (GL_CW or GL_CCW) to the **glFrontFace** function [80] (Chapter 2). Similarly, the triangles may be culled (never rasterized), if culling is enabled (by a **glEnable**(GL_CULL_FACE) OpenGL call) and their edge orientation information is used in conjunction with the OpenGL function **glCullFace** that is taking the arguments GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK [80] (Chapter 3). For example, in a scene composed entirely of opaque closed surfaces, back-facing polygons are never visible and can be culled. A software approach at the driver level to determine the orientation of a triangle is to compute its signed area in the screen space and to check its sign to render a decision. Using the algorithm for triangle rasterization based on edge functions, face determination can be performed as well in the hardware incurring only a little cost for computing the signed area by employing Equation (3.18). The method is illustrated in Figure 3.6. The hardware solution of the problem of determining the colors for the visible face — or alternatively, the culling of the face — is summarized in Tables 3.3 and 3.4.

Third, in OpenGL [80] (Appendix B, Corollary 14) there is another requirement concerning degenerate triangles (triangles that have zero area). The corollary is quoted in the followings:

> *"Because rasterization of non-antialiased polygons is point sampled, polygons that have no area generate no fragments when they are rasterized in GL_FILL mode, and the fragments gen-*

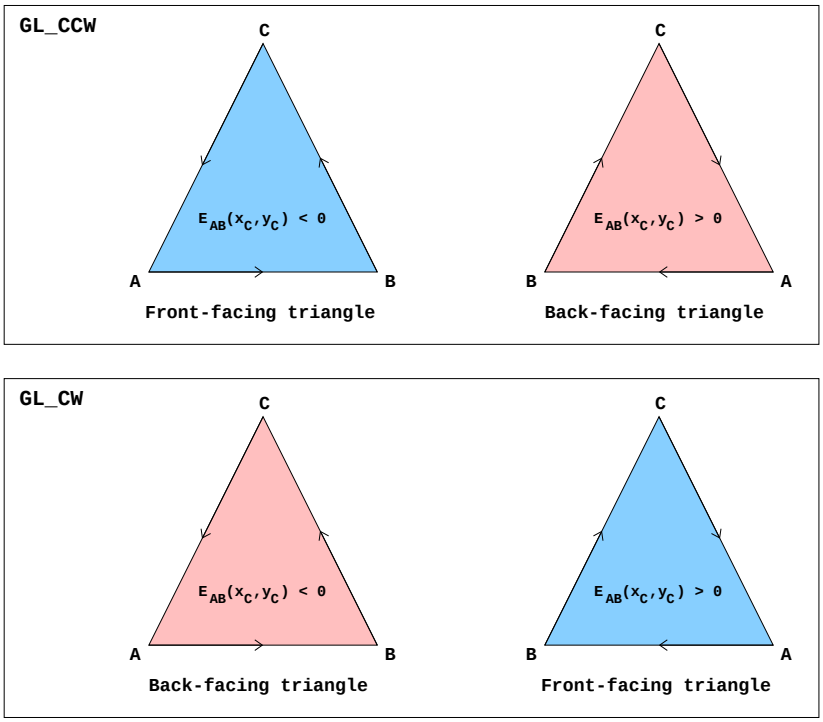

Figure 3.6: The OpenGL facing convention and its correlation with the sign of the $E_{AB}(x_C, y_C)$.

> *erated by the rasterization of "narrow" polygons may not form a continuous array."*

This requirement for aliased triangles in the GL_FILL mode can be implemented either at the software driver level or in the hardware rasterization engine. Nota bene, when a triangle is degenerate, there is no algorithm to tell the color of its visible face (because actually there is not such a face, the triangle is seen from the edge and a triangle can be considered infinitesimally thin), so the color does not matter. On the software side, the signed area of the aliased triangle has to be computed and if the result of the area computation is zero (actually, below a specified threshold, given the fact that the precision of the arithmetic in a digital computer is finite) the primitive can be culled. On the hardware side, when degenerate aliased triangles are detected (testing $E_{AB}(x_C, y_C)$ as in Table 3.3), the generation of fragments will be inhibited.

glFrontFace argument	$E_{AB}(x_C, y_C)$	**glCullFace** argument	Triangle culled	Remarks
GL_CW	< 0	GL_FRONT	no	Back-facing
	$\equiv 0$		yes	Degenerate
	> 0		yes	Front-facing
	< 0	GL_BACK	yes	Back-facing
	$\equiv 0$		yes	Degenerate
	> 0		no	Front-facing
GL_CCW	< 0	GL_FRONT	yes	Front-facing
	$\equiv 0$		yes	Degenerate
	> 0		no	Back-facing
	< 0	GL_BACK	no	Front-facing
	$\equiv 0$		yes	Degenerate
	> 0		yes	Back-facing

Table 3.4: Triangle culling for a triangle described with edge functions when culling is enabled (non-degenerate triangles are never culled if culling is disabled).

There are implementation tradeoffs to consider between the proposed hardware-based solution and the classical software approach to solve the above problems. The hardware solution relieves the software from the computation of signed area for every triangle and comes with no implementation costs (anyhow $E_{AB}(x_C, y_C)$ have to be computed — the reader is referred to the next sections). However, the hardware solution increases the data traffic. For instance, in a typical scene, a large number of triangles will be culled — if one decides to implement culling in the rasterization engine, these triangles will have to be sent over the bus in order to be finally discarded. In the same manner, for every triangle, two pairs of colors per vertex (the front and the back colors), instead of only one pair of colors (the primary and the secondary color of the visible face) per vertex, will have to be sent to the rasterization engine that will select only the one for the visible face of the triangle.

To resume, the rasterization engine will receive the following data per vertex: the screen space x, y, z coordinates, the post-perspective transformation reciprocal $1/w$ (denoted in the following sections by $Reciprocal\_w$), the primary and secondary colors (in format RGBA), the fog blending factor f, and the s, t, r, q texture coordinates for every texture unit. In the following, for every type of data enumerated above, a set of setup values for their interpolators will

be computed.

The z Setup

The interpolation steps and the setup for the z value of the fragments produced by the rasterization engine will be derived first. Using the fact that a flat polygon in eye space transforms into a flat polygon into screen space (Chapter 2, Section 2.2, Remark 2.2.1), the plane equation for the positional coordinates of a triangle in screen space can be written as:

$$a \cdot x + b \cdot y + z(x, y) + c = 0 \tag{3.19}$$

To determine the interpolation steps for the linear interpolation of z (described in Chapter 2.2) along the x and y axes, the partial derivatives $\delta z/\delta x$ and $\delta z/\delta y$ will have to be computed. These partial derivatives tell that for a step of one pixel along the x or y axis, the z value will change with the amount $\delta z/\delta x$ or $\delta z/\delta y$. By derivating Equation (3.19) with respect to x, and separately to y, it yields:

$$a + \frac{\delta z}{\delta x} = 0 \tag{3.20}$$

$$b + \frac{\delta z}{\delta y} = 0 \tag{3.21}$$

or alternatively:

$$\frac{\delta z}{\delta x} = -a \tag{3.22}$$

$$\frac{\delta z}{\delta y} = -b \tag{3.23}$$

The initial value for the z interpolator is:

$$z(x_{hit}, y_{hit}) = -a \cdot x_{hit} - b \cdot y_{hit} - c \tag{3.24}$$

where the pixel having the *screen* coordinates (x_{hit}, y_{hit}) represents the point where the tile scanning process with edge functions intersects for the first time the triangle boundary (more details are provided in Subsection 3.2.3).

To compute the plane coefficients a, b, and c, the fact that the positional coordinates of the three vertices of the triangle are solutions of the plane equation

can be used. This leads to the following system of equations:

$$\begin{cases} a \cdot x_A + b \cdot y_A + c = -z_A \\ a \cdot x_B + b \cdot y_B + c = -z_B \\ a \cdot x_C + b \cdot y_C + c = -z_C \end{cases} \tag{3.25}$$

Actually, the computation of the plane coefficient c is unnecessary because, by employing Equations (3.24) and (3.25), the initial value for the z interpolator can be written as (the following notations $\Delta x_{hit} = x_{hit} - x_A$ and $\Delta y_{hit} = y_{hit} - y_A$ are introduced here):

$$\begin{aligned} z(x_{hit}, y_{hit}) &= -a \cdot x_{hit} - b \cdot y_{hit} + a \cdot x_A + b \cdot y_A + z_A \\ &= -a \cdot (x_{hit} - x_A) - b \cdot (y_{hit} - y_A) + z_A \\ &= \frac{\delta z}{\delta x} \cdot (x_{hit} - x_A) + \frac{\delta z}{\delta y} \cdot (y_{hit} - y_A) + z_A \\ &= \frac{\delta z}{\delta x} \cdot \Delta x_{hit} + \frac{\delta z}{\delta y} \cdot \Delta y_{hit} + z_A \end{aligned} \tag{3.26}$$

Equation (3.25) can be solved for a and b using the Cramer's Rule:

$$a = \frac{\Delta_a}{\Delta} \tag{3.27}$$

$$b = \frac{\Delta_b}{\Delta} \tag{3.28}$$

where Δ, Δ_a, and Δ_b are the determinants computed in the following (the new notations $\Delta z_{AB} = z_B - z_A$ and $\Delta z_{CA} = z_A - z_C$ are introduced here):

$$\begin{aligned} \Delta &= \begin{vmatrix} x_A & y_A & 1 \\ x_B & y_B & 1 \\ x_C & y_C & 1 \end{vmatrix} \\ &= x_A \cdot y_B + x_C \cdot y_A + x_B \cdot y_C + x_A \cdot y_A \\ &\quad - x_C \cdot y_B - x_B \cdot y_A - x_A \cdot y_C - x_A \cdot y_A \\ &= -[(x_C - x_A) \cdot (y_B - y_A) - (y_C - y_A) \cdot (x_B - x_A)] \\ &= -[(x_C - x_A) \cdot \Delta y_{AB} - (y_C - y_A) \cdot \Delta x_{AB}] \\ &= -E_{AB}(x_C, y_C) \end{aligned} \tag{3.29}$$

$$\begin{aligned}
\Delta_a &= \begin{vmatrix} -z_A & y_A & 1 \\ -z_B & y_B & 1 \\ -z_C & y_C & 1 \end{vmatrix} \\
&= (z_C - z_A) \cdot (y_B - y_A) - (y_C - y_A) \cdot (z_B - z_A) \\
&= -(z_A - z_C) \cdot \Delta y_{AB} + \Delta y_{CA} \cdot (z_B - z_A) \\
&= \Delta y_{CA} \cdot \Delta z_{AB} - \Delta z_{CA} \cdot \Delta y_{AB} \qquad (3.30)
\end{aligned}$$

$$\begin{aligned}
\Delta_b &= \begin{vmatrix} x_A & -z_A & 1 \\ x_B & -z_B & 1 \\ x_C & -z_C & 1 \end{vmatrix} \\
&= (x_C - x_A) \cdot (z_B - z_A) - (z_C - z_A) \cdot (x_B - x_A) \\
&= -\Delta x_{CA} \cdot (z_B - z_A) + (z_A - z_C) \cdot \Delta x_{AB} \\
&= \Delta z_{CA} \cdot \Delta x_{AB} - \Delta x_{CA} \cdot \Delta z_{AB} \qquad (3.31)
\end{aligned}$$

As a remark, if the triangle is degenerate (has zero area following that $E_{AB}(x_C, y_C) \equiv 0$), the system of equations (3.25) will not have an unique solution.

From the Equations (3.27), (3.28), (3.29), (3.30), and (3.31), it can be inferred that:

$$a = -\frac{\Delta y_{CA} \cdot \Delta z_{AB} - \Delta z_{CA} \cdot \Delta y_{AB}}{E_{AB}(x_C, y_C)} \qquad (3.32)$$

$$b = -\frac{\Delta z_{CA} \cdot \Delta x_{AB} - \Delta x_{CA} \cdot \Delta z_{AB}}{E_{AB}(x_C, y_C)} \qquad (3.33)$$

By substituting the Equations (3.32) and (3.33) into the Equations (3.22), (3.23), and (3.26), the new values that have to be computed for the setup stage of the z interpolator per triangle (from the previous values already computed

for the edge function setup) are:

$$\begin{array}{rcl} E_{AB}(x_C, y_C) & = & -\Delta x_{CA} \cdot \Delta y_{AB} + \Delta y_{CA} \cdot \Delta x_{AB} \\ \\ R\_E & = & \frac{1}{E_{AB}(x_C, y_C)} \\ \\ \Delta z_{AB} & = & z_B - z_A \\ \Delta z_{CA} & = & z_A - z_C \\ \Delta x_{hit} & = & x_{hit} - x_A \\ \Delta y_{hit} & = & y_{hit} - y_A \\ \\ \frac{\delta z}{\delta x} & = & (\Delta y_{CA} \cdot \Delta z_{AB} - \Delta z_{CA} \cdot \Delta y_{AB}) \cdot R\_E \\ \\ \frac{\delta z}{\delta y} & = & (\Delta z_{CA} \cdot \Delta x_{AB} - \Delta x_{CA} \cdot \Delta z_{AB}) \cdot R\_E \\ \\ z(x_{hit}, y_{hit}) & = & \frac{\delta z}{\delta x} \cdot \Delta x_{hit} + \frac{\delta z}{\delta y} \cdot \Delta y_{hit} + z_A \end{array} \tag{3.34}$$

After the setup is completed, the z interpolator can compute the z values incrementally by simple addition for adjacent pixels:

$$\boxed{z(x+1, y) = z(x, y) + \frac{\delta z}{\delta x}} \tag{3.35}$$

$$\boxed{z(x, y+1) = z(x, y) + \frac{\delta z}{\delta y}} \tag{3.36}$$

Generalizing, the z value for a pixel with the address $(x + \delta x, y + \delta y)$ can be computed from the z value for the pixel (x, y) as:

$$\boxed{z(x + \delta x, y + \delta y) = z(x, y) + \frac{\delta z}{\delta x} \cdot \delta x + \frac{\delta z}{\delta y} \cdot \delta y} \tag{3.37}$$

by using two multiplications and one addition.

The above boxed equations will remain valid if they are written for any cyclic permutation σ of vertex indices $\langle A, B, C \rangle$.

In OpenGL, the depth values of all fragments generated by the rasterization of a triangle may be offset by a single value that is computed for that triangle. The function that determines this value is **glPolygonOffset** [80] (Chapter 3) and it can be implemented using the values for $\delta z/\delta x$ and $\delta z/\delta y$ determined for the z value interpolation. This function is useful for rendering hidden-line images, for applying decals to surfaces, and for rendering solids with highlighted edges [83].

The RGBA Color Setup

As it has been already mentioned in Chapter 2, Section 2.2, the GL_PERSPECTIVE_CORRECTION_HINT target parameter of the **gl_Hint** OpenGL function indicates how color values are interpolated across a primitive: either linearly in screen space, or in the more expensive perspective-correct manner. Often, systems perform linear color interpolation because the results, while not technically correct, are visually acceptable. First, the color setup for perspective correct color interpolation will be treated, followed by the description of the color setup for linear color interpolation.

Perspective Correct Color Interpolation Setup Using the results of Chapter 2.2, the colors will be interpolated hyperbolically in screen space. Thus, the problem of finding the color component c_i ($c \in \{R, G, B, A\}$ and $i \in \{primary, secondary\}$) for a pixel with a given *screen* address (x, y) can be reduced to a linear interpolation in the plane $(x, y, \gamma_{ci}(x, y))$ (where $\gamma_{ci}(x_A, y_A) = c_{Ai}/w_A$, $\gamma_{ci}(x_B, y_B) = c_{Bi}/w_B$, and $\gamma_{ci}(x_C, y_C) = c_{Ci}/w_C$ at the vertices), a linear interpolation in the plane $(x, y, \zeta(x, y))$ (where $\zeta(x_A, y_A) = 1/w_A$, $\zeta(x_B, y_B) = 1/w_B$, and $\zeta(x_C, y_C) = 1/w_C$), and a division of the two interpolated values for that pixel.

Following a similar path with the one presented in Section 3.2.2, it can be written that:

$$\begin{aligned}
\frac{\delta\gamma_{ci}}{\delta x} &= \frac{\Delta y_{CA} \cdot [\gamma_{ci}(x_B, y_B) - \gamma_{ci}(x_A, y_A)]}{E_{AB}(x_C, y_C)} \\
&\quad - \frac{[\gamma_{ci}(x_A, y_A) - \gamma_{ci}(x_C, y_C)] \cdot \Delta y_{AB}}{E_{AB}(x_C, y_C)} \\
\frac{\delta\gamma_{ci}}{\delta y} &= \frac{[\gamma_{ci}(x_A, y_A) - \gamma_{ci}(x_C, y_C)] \cdot \Delta x_{AB}}{E_{AB}(x_C, y_C)} \\
&\quad - \frac{\Delta x_{CA} \cdot [\gamma_{ci}(x_B, y_B) - \gamma_{ci}(x_A, y_A)]}{E_{AB}(x_C, y_C)} \\
\gamma_{ci}(x_{hit}, y_{hit}) &= \frac{\delta\gamma_{ci}}{\delta x} \cdot \Delta x_{hit} + \frac{\delta\gamma_{ci}}{\delta y} \cdot \Delta y_{hit} + \gamma_{ci}(x_A, y_A)
\end{aligned} \tag{3.38}$$

$$\begin{aligned}
\frac{\delta\zeta}{\delta x} &= \frac{\Delta y_{CA}\cdot[\zeta(x_B,y_B)-\zeta(x_A,y_A)]}{E_{AB}(x_C,y_C)} \\
&\quad -\frac{[\zeta(x_A,y_A)-\zeta(x_C,y_C)]\cdot\Delta y_{AB}}{E_{AB}(x_C,y_C)} \\
\frac{\delta\zeta}{\delta y} &= \frac{[\zeta(x_A,y_A)-\zeta(x_C,y_C)]\cdot\Delta x_{AB}}{E_{AB}(x_C,y_C)} \\
&\quad -\frac{\Delta x_{CA}\cdot[\zeta(x_B,y_B)-\zeta(x_A,y_A)]}{E_{AB}(x_C,y_C)} \\
\zeta(x_{hit},y_{hit}) &= \frac{\delta\zeta}{\delta x}\cdot\Delta x_{hit}+\frac{\delta\zeta}{\delta y}\cdot\Delta y_{hit}+\zeta(x_A,y_A)
\end{aligned} \tag{3.39}$$

or, more simply:

$$\begin{aligned}
\frac{\delta\gamma_{ci}}{\delta x} &= \frac{\Delta y_{CA}\cdot(\frac{c_{Bi}}{w_B}-\frac{c_{Ai}}{w_A})-(\frac{c_{Ai}}{w_A}-\frac{c_{Ci}}{w_C})\cdot\Delta y_{AB}}{E_{AB}(x_C,y_C)} \\
\frac{\delta\gamma_{ci}}{\delta y} &= \frac{(\frac{c_{Ai}}{w_A}-\frac{c_{Ci}}{w_C})\cdot\Delta x_{AB}-\Delta x_{CA}\cdot(\frac{c_{Bi}}{w_B}-\frac{c_{Ai}}{w_A})}{E_{AB}(x_C,y_C)} \\
\gamma_{ci}(x_{hit},y_{hit}) &= \frac{\delta\gamma_{ci}}{\delta x}\cdot\Delta x_{hit}+\frac{\delta\gamma_{ci}}{\delta y}\cdot\Delta y_{hit}+\frac{c_{Ai}}{w_A}
\end{aligned} \tag{3.40}$$

$$\begin{aligned}
\frac{\delta\zeta}{\delta x} &= \frac{\Delta y_{CA}\cdot(\frac{1}{w_B}-\frac{1}{w_A})-(\frac{1}{w_A}-\frac{1}{w_C})\cdot\Delta y_{AB}}{E_{AB}(x_C,y_C)} \\
\frac{\delta\zeta}{\delta y} &= \frac{(\frac{1}{w_A}-\frac{1}{w_C})\cdot\Delta x_{AB}-\Delta x_{CA}\cdot(\frac{1}{w_B}-\frac{1}{w_A})}{E_{AB}(x_C,y_C)} \\
\zeta(x_{hit},y_{hit}) &= \frac{\delta\zeta}{\delta x}\cdot\Delta x_{hit}+\frac{\delta\zeta}{\delta y}\cdot\Delta y_{hit}+\frac{1}{w_A}
\end{aligned} \tag{3.41}$$

Thus, the new values that have to be computed for the setup stage of the nine interpolators (there are four interpolators for the primary color and another four for the secondary color associated with γ_{ci}, plus one interpolator independent

of any color component ζ) of the colors per triangle (from the previous values already computed for the edge function setup and z value setup) are:

$$
\begin{aligned}
c\_over\_w_{Ai} &= c_{Ai} \cdot Reciprocal\_w_A \\
c\_over\_w_{Bi} &= c_{Bi} \cdot Reciprocal\_w_B \\
c\_over\_w_{Ci} &= c_{Ci} \cdot Reciprocal\_w_C \\
\Delta Reciprocal\_w_{AB} &= Reciprocal\_w_B - Reciprocal\_w_A \\
\Delta Reciprocal\_w_{CA} &= Reciprocal\_w_A - Reciprocal\_w_C \\
\Delta c\_over\_w_{ABi} &= c\_over\_w_{Bi} - c\_over\_w_{Ai} \\
\Delta c\_over\_w_{CAi} &= c\_over\_w_{Ai} - c\_over\_w_{Ci} \\
\\
\frac{\delta \gamma_{ci}}{\delta x} &= (\Delta y_{CA} \cdot \Delta c\_over\_w_{ABi} \\
&\quad -\Delta c\_over\_w_{CAi} \cdot \Delta y_{AB}) \cdot R\_E \\
\frac{\delta \gamma_{ci}}{\delta y} &= (\Delta c\_over\_w_{CAi} \cdot \Delta x_{AB} \\
&\quad -\Delta x_{CA} \cdot \Delta c\_over\_w_{ABi}) \cdot R\_E \\
\\
\gamma_{ci}(x_{hit}, y_{hit}) &= \frac{\delta \gamma_{ci}}{\delta x} \cdot \Delta x_{hit} + \frac{\delta \gamma_{ci}}{\delta y} \cdot \Delta y_{hit} \\
&\quad +c\_over\_w_{Ai} \\
\\
\frac{\delta \zeta}{\delta x} &= (\Delta y_{CA} \cdot \Delta Reciprocal\_w_{AB} \\
&\quad -\Delta Reciprocal\_w_{CA} \cdot \Delta y_{AB}) \cdot R\_E \\
\frac{\delta \zeta}{\delta y} &= (\Delta Reciprocal\_w_{CA} \cdot \Delta x_{AB} \\
&\quad -\Delta x_{CA} \cdot \Delta Reciprocal\_w_{AB}) \cdot R\_E \\
\\
\zeta(x_{hit}, y_{hit}) &= \frac{\delta \zeta}{\delta x} \cdot \Delta x_{hit} + \frac{\delta \zeta}{\delta y} \cdot \Delta y_{hit} \\
&\quad +Reciprocal\_w_A
\end{aligned}
\tag{3.42}
$$

After the setup is completed, the nine interpolators can compute their values incrementally by simple addition for adjacent pixels:

$$
\begin{aligned}
\gamma_{ci}(x+1, y) &= \gamma_{ci}(x, y) + \frac{\delta \gamma_{ci}}{\delta x} \\
\zeta(x+1, y) &= \zeta(x, y) + \frac{\delta \zeta}{\delta x}
\end{aligned}
\tag{3.43}
$$

$$
\begin{aligned}
\gamma_{ci}(x, y+1) &= \gamma_{ci}(x, y) + \frac{\delta \gamma_{ci}}{\delta y} \\
\zeta(x, y+1) &= \zeta(x, y) + \frac{\delta \zeta}{\delta y}
\end{aligned}
\tag{3.44}
$$

Generalizing, the interpolated values for a pixel with the address $(x + \delta x, y + \delta y)$ can be computed from the interpolated values for the pixel (x, y) as:

$$\boxed{\begin{array}{rcl} \gamma_{ci}(x+\delta x, y+\delta y) & = & \gamma_{ci}(x,y) + \frac{\delta\gamma_{ci}}{\delta x}\cdot\delta x + \frac{\delta\gamma_{ci}}{\delta y}\cdot\delta y \\ \\ \zeta(x+\delta x, y+\delta y) & = & \zeta(x,y) + \frac{\delta\zeta}{\delta x}\cdot\delta x + \frac{\delta\zeta}{\delta y}\cdot\delta y \end{array}} \tag{3.45}$$

using two multiplications and one addition per interpolator.

Finally, the values of the color components for the pixel with the screen space address (x, y) can be computed as:

$$\boxed{\begin{array}{rcl} Reciprocal\_\zeta(x,y) & = & \frac{1}{\zeta(x,y)} \\ \\ c_i(x,y) & = & \gamma_{ci}(x,y)\cdot Reciprocal\_\zeta(x,y) \end{array}} \tag{3.46}$$

with one reciprocal and eight multiplications.

The above boxed equations will remain valid if they are written for any cyclic permutation σ of vertex indices $\langle A, B, C\rangle$.

Linear Color Interpolation Setup If the colors are approximated employing linear interpolation in the *screen* space, similar results with those of z value setup can be found. Thus, the problem of finding the color component c_i ($c \in \{R, G, B, A\}$ and $i \in \{primary, secondary\}$) for a pixel with a given *screen* address (x, y) can be reduced to a linear interpolation in the plane $(x, y, c_i(x, y))$ (where $c_i(x_A, y_A) = c_{Ai}$, $c_i(x_B, y_B) = c_{Bi}$, and $c_i(x_C, y_C) = c_{Ci}$ at the vertices of the triangle).

The new values that have to be computed for the setup stage of the eight color interpolators (there are four interpolators for the primary color and another four for the secondary color) per triangle (from the previous values already computed for the edge function setup and for the z value setup) are:

$$\boxed{\begin{array}{rcl} \Delta c_{ABi} & = & c_{Bi} - c_{Ai} \\ \Delta c_{CAi} & = & c_{Ai} - c_{Ci} \\ \\ \frac{\delta c_i}{\delta x} & = & (\Delta y_{CA}\cdot\Delta c_{ABi} - \Delta c_{CAi}\cdot\Delta y_{AB})\cdot R\_E \\ \\ \frac{\delta c_i}{\delta y} & = & (\Delta c_{CAi}\cdot\Delta x_{AB} - \Delta x_{CA}\cdot\Delta c_{ABi})\cdot R\_E \\ \\ c_i(x_{hit}, y_{hit}) & = & \frac{\delta c_i}{\delta x}\cdot\Delta x_{hit} + \frac{\delta c_i}{\delta y}\cdot\Delta y_{hit} + c_{Ai} \end{array}} \tag{3.47}$$

After the setup is completed, the color interpolators can compute the values of the color components incrementally by simple addition for adjacent pixels:

$$\boxed{c_i(x+1,y) = c_i(x,y) + \frac{\delta c_i}{\delta x}} \tag{3.48}$$

$$\boxed{c_i(x,y+1) = c_i(x,y) + \frac{\delta c_i}{\delta y}} \tag{3.49}$$

Generalizing, the interpolated values of the color components for a pixel with the screen space address $(x+\delta x, y+\delta y)$ can be computed from the interpolated values for the pixel (x,y) as:

$$\boxed{c_i(x+\delta x, y+\delta y) = c_i(x,y) + \frac{\delta c_i}{\delta x}\cdot\delta x + \frac{\delta c_i}{\delta y}\cdot\delta y} \tag{3.50}$$

using two multiplications and one addition per interpolator.

The above boxed equations will remain valid if they are written for any cyclic permutation σ of vertex indices $\langle A, B, C\rangle$.

The fog blending factor can be set up and interpolated in an identical manner with that employed for the RGBA color components.

It can be seen very clearly that the problem of the linear interpolation of the colors is more simple technically than the problem of their perspective correct interpolation.

The Texture Coordinates Setup

Using the results of Chapter 2.2, the texture coordinates (s,t,r,q) will be interpolated hyperbolically in screen space. Thus, the problem of finding the non-homogeneous texture coordinate tex/q ($tex \in \{s,t,r\}$) for a pixel with a given *screen* address (x,y) can be reduced to a linear interpolation in the plane $(x,y,\eta_{tex}(x,y))$ (where $\eta_{tex}(x_A,y_A) = tex_A/w_A$, $\eta_{tex}(x_B,y_B) = tex_B/w_B$, and $\eta_{tex}(x_C,y_C) = tex_C/w_C$ at the vertices of the triangle), a linear interpolation in the plane $(x,y,\xi(x,y))$ (where $\xi(x_A,y_A) = q_A/w_A$, $\xi(x_B,y_B) = q_B/w_B$, and $\xi(x_C,y_C) = q_C/w_C$), and a division of the two interpolated values for that pixel.

Following a similar path with the one presented for perspective correct interpolated colors, the new values that have to be computed for the setup stage of the four interpolators (there are three interpolators associated with η_{tex} plus

one interpolator associated with ξ) of the texture coordinates (from the previous values already computed for the edge function setup, z value setup, and RGBA color setup) are:

$$\begin{aligned}
tex_over_w_A &= tex_A \cdot Reciprocal_w_A \\
tex_over_w_B &= tex_B \cdot Reciprocal_w_B \\
tex_over_w_C &= tex_C \cdot Reciprocal_w_C \\
q_over_w_A &= q_A \cdot Reciprocal_w_A \\
q_over_w_B &= q_B \cdot Reciprocal_w_B \\
q_over_w_C &= q_C \cdot Reciprocal_w_C \\
\Delta tex_over_w_{AB} &= tex_over_w_B - tex_over_w_A \\
\Delta tex_over_w_{CA} &= tex_over_w_A - tex_over_w_C \\
\Delta q_over_w_{AB} &= q_over_w_B - q_over_w_A \\
\Delta q_over_w_{CA} &= q_over_w_A - q_over_w_C \\
\\
\frac{\delta \eta_{tex}}{\delta x} &= (\Delta y_{CA} \cdot \Delta tex_over_w_{AB} \\
&\quad -\Delta tex_over_w_{CA} \cdot \Delta y_{AB}) \cdot R_E \\
\frac{\delta \eta_{tex}}{\delta y} &= (\Delta tex_over_w_{CA} \cdot \Delta x_{AB} \\
&\quad -\Delta x_{CA} \cdot \Delta tex_over_w_{AB}) \cdot R_E \\
\\
\eta_{tex}(x_{hit}, y_{hit}) &= \frac{\delta \eta_{tex}}{\delta x} \cdot \Delta x_{hit} + \frac{\delta \eta_{tex}}{\delta y} \cdot \Delta y_{hit} \\
&\quad +tex_over_w_A \\
\\
\frac{\delta \xi}{\delta x} &= (\Delta y_{CA} \cdot \Delta q_over_w_{AB} \\
&\quad -\Delta q_over_w_{CA} \cdot \Delta y_{AB}) \cdot R_E \\
\frac{\delta \xi}{\delta y} &= (\Delta q_over_w_{CA} \cdot \Delta x_{AB} \\
&\quad -\Delta x_{CA} \cdot \Delta q_over_w_{AB}) \cdot R_E \\
\\
\xi(x_{hit}, y_{hit}) &= \frac{\delta \xi}{\delta x} \cdot \Delta x_{hit} + \frac{\delta \xi}{\delta y} \cdot \Delta y_{hit} + q_over_w_A
\end{aligned} \tag{3.51}$$

After the setup is completed, the four interpolators can compute their values incrementally by simple addition for adjacent pixels:

$$\begin{aligned}
\eta_{tex}(x+1, y) &= \eta_{tex}(x, y) + \frac{\delta \eta_{tex}}{\delta x} \\
\\
\xi(x+1, y) &= \xi(x, y) + \frac{\delta \xi}{\delta x}
\end{aligned} \tag{3.52}$$

$$\boxed{\begin{array}{rcl} \eta_{tex}(x, y+1) & = & \eta_{tex}(x, y) + \frac{\delta \eta_{tex}}{\delta y} \\ \\ \xi(x, y+1) & = & \xi(x, y) + \frac{\delta \xi}{\delta y} \end{array}} \tag{3.53}$$

Generalizing, the interpolated values for a pixel with the address $(x + \delta x, y + \delta y)$ can be computed from the interpolated values for the pixel (x, y) as:

$$\boxed{\begin{array}{rcl} \eta_{tex}(x + \delta x, y + \delta y) & = & \eta_{tex}(x, y) + \frac{\delta \eta_{tex}}{\delta x} \cdot \delta x + \frac{\delta \eta_{tex}}{\delta y} \cdot \delta y \\ \\ \xi(x + \delta x, y + \delta y) & = & \xi(x, y) + \frac{\delta \xi}{\delta x} \cdot \delta x + \frac{\delta \xi}{\delta y} \cdot \delta y \end{array}} \tag{3.54}$$

using two multiplications and one addition per interpolator.

Finally, the values of the non-homogeneous texture coordinates for the pixel with the screen space address (x, y) can be computed as:

$$\boxed{\begin{array}{rcl} Reciprocal\_\xi(x, y) & = & \frac{1}{\xi(x,y)} \\ \\ \frac{tex(x,y)}{q(x,y)} & = & \eta_{tex}(x, y) \cdot Reciprocal\_\xi(x, y) \end{array}} \tag{3.55}$$

with one reciprocal and three multiplications.

The above boxed equations will remain valid if they are written for any cyclic permutation σ of vertex indices $\langle A, B, C \rangle$.

We have to mention that if there is more than one texture unit in the rasterization engine, multiple texture coordinates sets (s, t, r, q) will be supported per vertex (one set for every texture unit), and the above computations have to be performed for every texture coordinates set.

3.2.3 Triangle Traversal Algorithm

As it has already been mentioned, the triangle can be traversed by any algorithm that is guaranteed to cover all of the pixels of the tile that have a relationship with the triangle. Moreover, the rasterization algorithm with the edge functions produces correct results irrespective of the triangle position in relation with the tile (a triangle can be completely outside, completely inside, partially contained in a tile, or can contain the whole tile). In any case, it does not require a prior clipping of the triangle in another triangles fully contained in the tile.

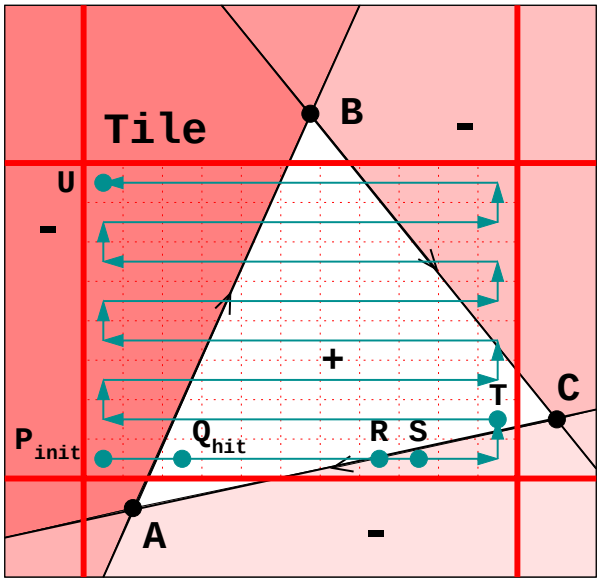

Figure 3.7: Traversing the tile entirely.

An elaboration on the discussion from [74] is presented in the rest of the subsection. Figure 3.7 presents the simplest strategy that consists of covering all of the pixels in a tile. As it can be seen, this algorithm is not efficient because it computes the three edge functions for pixels that may not belong to the triangle. Moreover, if it is considered that no computation of fragment attribute values is wasted for pixels outside the triangle (where, for instance, colors and the z value can become negative and exit by far their numerical range assigned in Section 3.1), and that the fragment attribute values are computed only for pixels that are interior (the precise meaning of interior was described at edge function setup in Subsection 3.2.2), it follows that not always the fragment's attribute digital interpolators can compute the value for the next pixel incrementally by simple addition from the value they have had for the previous pixel. To make things clear, in the following example, only the interpolation of the z fragment attribute values will be considered on the trajectory $\mathrm{P_{init}} \rightarrow \mathrm{Q_{hit}} \rightarrow \mathrm{R} \rightarrow \mathrm{S} \rightarrow \mathrm{T}$ depicted in Figure 3.7. Nevertheless, the situation presented in the following applies equally to all of the fragment's attributes value interpolation. The pixel $\mathrm{P_{init}}$ with the *screen* coordinates (x_{init}, y_{init}) (tile offset $(0, 0)$) represents the initialization point for the edge functions. The initialization values were presented in Equation (3.13). After that, up to the last

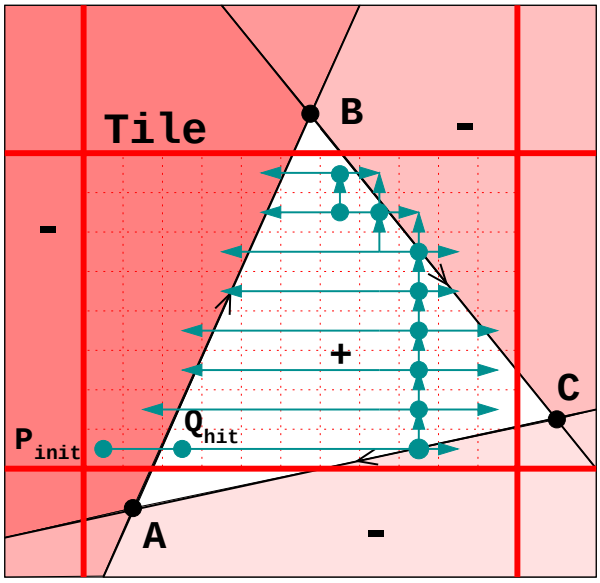

Figure 3.8: A more efficient triangle traversal algorithm.

scanned pixel U of the tile, the edge functions will be computed incrementally using only additions (Equations (3.9) or (3.10)) and their sign will be tested for every new position with the screen coordinates (x, y) to detect if this position is inside or outside the triangle. So, starting from the moment the edge functions are initialized, the process will continue and when the current scanned position matches the pixel $\mathrm{Q_{hit}}$ with the screen coordinates (x_{hit}, y_{hit}) where the boundary of the triangle will be crossed for the first time, the z value interpolator will be initialized accordingly to Equation (3.34). Then, up to the pixel R the z values will be computed incrementally with one addition per pixel employing Equation (3.35). At the next pixel S, a test of the sign of the edge functions will indicate that the triangle boundary was crossed again and the current position S is lying outside the triangle. At this moment, the computation of new z values for the current position and the next scanned pixels will be inhibited. Regarding the z value computation, the status quo will be maintained up to the pixel T, where the test of the edge functions' sign will indicate that the current position T is lying again inside the triangle. As a consequence, the z value interpolator will be activated again, but the z value of the current position T will be computed from the last computed z value that is available, which in this case is the position R's z value. This can be done by employ-

ing the less efficient Equation (3.37) with two additions and one multiplication ($\delta y \equiv 1$), instead of only one addition, indicating clearly another drawback of the algorithm.

A more efficient algorithm for the triangle traversal will eliminate the shortcomings seen above by using only one addition per attribute per pixel to calculate the new values of the fragment attributes from the ones of the previous visited pixels. This can be accomplished by making a simple observation: once the triangle's interior has been reached, one should try to contain the scanning process inside the triangle's boundaries. One such possible algorithm is depicted in Figure 3.8. The tradeoff is that more state information has to be maintained for this algorithm than for the basic algorithm, since the interpolators must be restarted back at the vertical lines, after one side of a scan line has been processed in order to begin, by employing only one addition per attribute per pixel, the scanning process for the other side of the scan line.

Many other traversal algorithms are possible. The best algorithm will depend on the performance/power/cost tradeoffs in the implementation.

Last but not least, an intrinsic drawback of the tiling architectures stems from the fact that a triangle can assume any position in relation with a tile. As a consequence, sometimes the rasterization process can have a "cold" start — when the pixel with the screen address (x_{hit}, y_{hit}) (where a pixel belonging to the interior of the triangle is encountered for the first time, resulting in an initialization of the fragment attribute value interpolators) is far away from the pixel with the screen address (x_{init}, y_{init}) (where the edge function interpolators are initialized), resulting in a lot of "blind" iterations (for a tile with the size $2^m \times 2^n$ pixels, in the worst case approximative $log_2(2^m) \cdot log_2(2^n)$ iterations if a binary search on the tile's pixels is employed using the feedback provided by the value of the edge functions) of the edge function interpolators until they are able to "bite" a bit from the triangle's interior. A possible cure would be the variant where the driver sends additionally, besides the vertex data for the triangle, of an initialization position coordinates inside the current processed tile for the edge function interpolators that is guaranteed to be also inside the triangle (thus $(x_{init}, y_{init}) \equiv (x_{hit}, y_{hit})$). Unfortunately, this solution is unfeasible because it can put a lot of computational burden on the host processor. Another approach, that does not require any additional work on the host processor and which does incure only a minimal penalty for "cold" starts, will be presented in Chapter 6.

3.3 Conclusion

In this chapter we have presented a complete mathematical formalism, based on the perspective-correct interpolation formulas derived in Chapter 2, that could be applied to any tile-based raterization engine. We have described how, after an initial computational stage called triangle setup, which is relative to the current tile and current triangle, operations could be performed to each pixel (or pixel block), in parallel to other pixels (or pixel blocks), to generate the triangle stencil or the attributes that are required by the pixel processing pipelines. Also, we have presented how values, for neighbouring pixels ocurring within the same pixel block, could be derived using only two-operand additions, which are cheaper to implement in hardware than multiplications. We have also described a series of difficulties for the triangle traversal algorithm in the tile-based context — an efficient hardware implementation that solves the issues described here will be presented in Chapter 6.

Chapter 4

Design Space Exploration

With the recent proliferation of embedded systems using the system-on-chip (SOC) design paradigm, such as PDAs (Personal Digital Assistants), cellular phones, and other portable computing appliances, the request for increasingly fast, graphics-reach user-friendly interfaces and entertainment environments opened new market opportunities for 3-D real-time rendering graphics systems meant to accelerate these features. The challenge posed by the severe cost constraints on products for the mobile consumer market requires a new breed of graphics rendering hardware to be developed with a very low power consumption and low implementation costs. This implies that image quality/performance/power/cost trade-offs have to be investigated and decisions have to be made early in the design process regarding the most suitable partition between features that must be provided by software and features that have to be mapped in hardware.

In this context, designing and assessing the performance (throughput, area, and power consumption) of embedded hardware architectures for accelerating graphics has proved to be a difficult endeavor. Among the reasons are the absence of graphics benchmark suites for portable graphics (although graphics system specifications like OpenGL [80], Direct3D [69] and high-tier benchmark suites as [84] exist for many years) and the lack of specific tools to support the graphics architecture development process [50]. As a consequence, heterogeneous design exploration frameworks were created by connecting custom-made tools with tools borrowed from other fields of computer architecture research, thus raising a lot of problems that have to be solved such as the interoperability, flexibility, and specificity for the intended purpose [50][56][26]. The main difficulty stemmed from the fact that all

these tools are based on certain general purpose processor architectural templates [17][91][16][65][91], and can be hardly adapted to model graphics accelerators embedded in SOC designs. On the other hand, integrated software/hardware co-design frameworks for graphics hardware accelerators as the one described in [37], based on mixed-mode C++/VHDL simulation, are plagued by slow simulation speed and by the impossibility to refine the functional description down to the implementation in a single modeling language.

To overcome these difficulties and to produce an efficient and productive development environment, we designed GRAAL [29][30], a versatile hardware/software co-simulation and co-design tool for embedded 3-D graphics accelerators. The GRAphics AcceLerator design exploration framework (GRAAL) is an open system which offers a coherent development methodology based on an extensive library of graphics pipeline components modeled at RT-level in SystemC [73], a language developed specifically for system level simulation and design. As a consequence, an entire system-on-chip can be simulated by integrating third-party SystemC models of components (microprocessors, memories, and peripherals) along with our own parameterizable SystemC RTL model of the graphics hardware accelerator. GRAAL incorporates tools to assist in the visual debugging of the graphics algorithms implemented in hardware, and to estimate the performance in terms of throughput, power consumption, and area.

The rest of the chapter is organized as follows. An overview of a SOC platform for embedded graphics applications is presented in Section 4.1, consisting of a host processor, a memory subsystem, the graphics accelerator to be designed, and other miscellaneous IP cores. GRAAL, the design exploration framework we propose for embedded graphics accelerator development for SOC designs is discussed in Section 4.2: the 3-D graphics component library in Subsection 4.2.1, the available visualization and simulation control in Subsection 4.2.2, the power/energy estimation model at two abstraction levels in Subsection 4.2.3, and the graphics benchmark generation process in Subsection 4.2.4. Results showing the effectiveness of the design exploration framework are presented in Section 4.3.

4.1 Embedded 3-D Graphics

A 3-D graphics rendering system was presented in Chapter 2, Section 2.1. It is organized conceptually as a number of stages chained in a pipelined fashion, as depicted in Figure 2.1. The conceptual stages of the graphics pipeline are

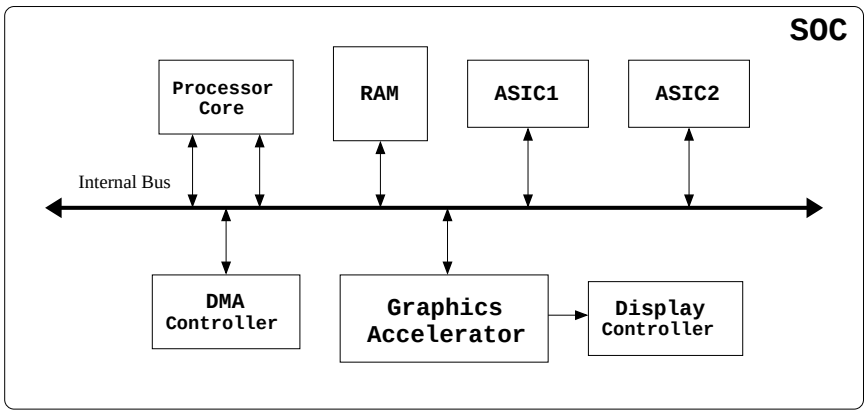

Figure 4.1: SOC organization.

the *application*, the *geometry*, and the *rasterizer stage*.

Typically, a platform for embedded graphics applications looks like in Figure 4.1, and the mapping of the conceptual stages of the 3-D graphics pipeline in the system are described in the following. The graphics software application is running on the host processor of the system. The software application corresponds to the conceptual Application Stage of the graphics pipeline. The software application is relying on a 3-D graphics library (perceived in the sense of a software interface to the graphics hardware [93]), such as OpenGL or Direct3D to have its graphic calls taken care further. This 3-D graphics library usually executes the conceptual Geometry Stage on the host processor. The code that implements the Geometry Stage in the library can further make calls to the graphics hardware accelerator by means of a standardized, virtual interface, to ensure library portability. Between this virtual interface and the graphics hardware accelerator (on which the conceptual Rasterizer Stage is mapped) there is another piece of code executed on the host processor called a device driver. This device driver performs the function of a hardware abstraction layer and translates the calls through the virtual interface in actual memory-mapped or programmable I/O instructions (seen from the host processor point of view) particular to the graphics hardware accelerator's input and output register port mapping in the system address space. Finally, the Rasterizer Stage is executed in hardware on the graphics hardware accelerator (due to the computational explosion at this level) performing the following operations. Given the primitives (usually triangles) received from the host processor with transformed and projected vertices, colors, and texture coordinates computed for this vertices,

the goal of the Rasterizer Stage inside the graphics accelerator is to perform rasterization: to assign correct colors to the pixels that will be stored in a memory called the *frame buffer*, which is read periodically by the display controller to form the image on the screen. The operations required by the Rasterizer Stage were overviewed in Chapter 2, Section 2.1, and described in detail in Chapter 3.

Efficient hardware mapping of the Rasterizer Stage functionality can only be achieved by employing the design exploration tools described in the next section.

4.2 GRAAL Design Exploration Framework

Once the graphics accelerator functionality has been partitioned experimentally between software and hardware, a design exploration framework should assist the designer in assessing early in the design process the merits of a potential implementation. More specifically, the Rasterizer Stage is divided in the functional stages that were presented, but a functional stage describes only the task to be performed in the pipeline, and does not specify the way the task is executed in the underlying hardware pipeline. A functional pipeline stage may be divided in several hardware pipeline stages, or two functional pipeline stages may be implemented in one hardware pipeline stage. A hardware pipeline stage may be also parallelized in order to meet high performance demands. At the same time, for every function performed in the Rasterizer Stage, a considerable number of hardware algorithms exists. Within a hardware algorithm datapath, various fixed-point data formats and precision can be employed that might have an impact on the quality of the generated image. As a consequence, different image quality/performance/power/cost trade-offs would have to be explored by a designer in order to choose the best solution for the to-be-developed graphics hardware accelerator, and the huge design space has to be timely explored under the pressure of tight time-to-market constraints.

GRAAL is meant to assist the designer in addressing these issues. An overview of the design exploration framework that we propose is presented in Figure 4.2. The central elements (discussed in the next subsections) of the framework are:

- The reference implementation of a SystemC simulator available at [73];
- GRAAL (GRAphics AcceLerator) Simulator, our own custom-designed

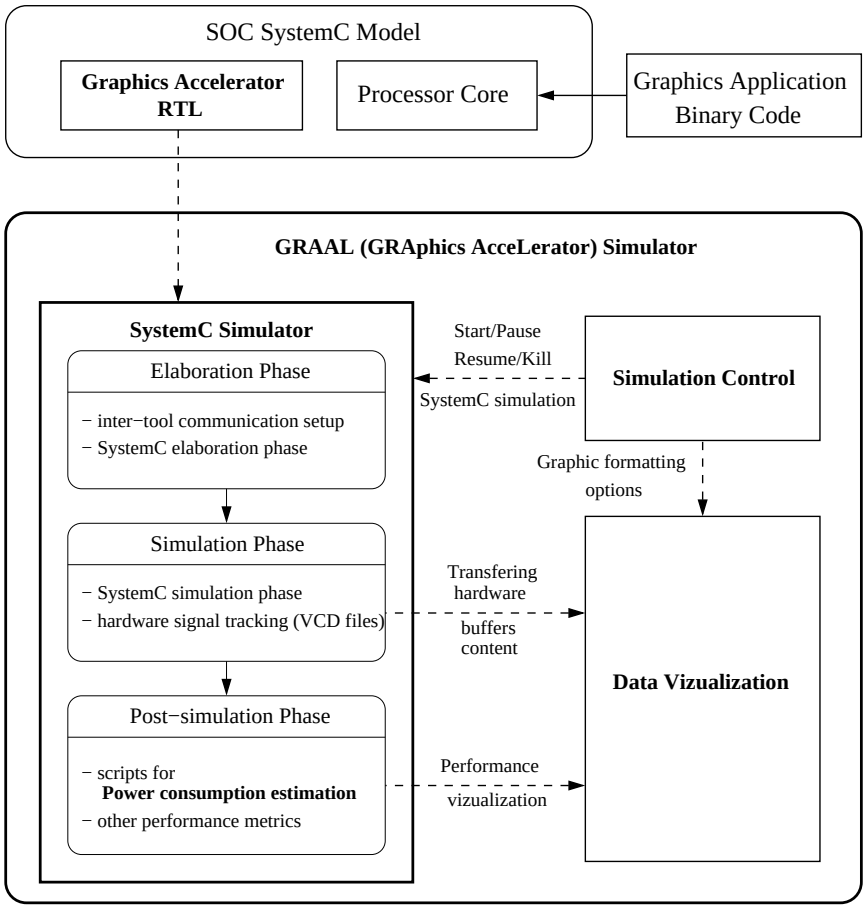

Figure 4.2: GRAAL tool framework.

tool that acts as a graphical front-end for SystemC simulation control, data visualization, and performance estimation;

- The SystemC model of the candidate graphics hardware accelerator designed using our (extendable) library of graphics pipeline components modeled in SystemC at RT-level;
- Third-party SystemC models of SOC components including a processor core, memories, and peripherals to simulate the entire system;
- The graphics software application in a binary form, which runs on the processor core, and makes use of the capabilities of the graphics hardware accelerator.

Within this framework, different graphics software applications can be run on the virtual system and various graphics hardware accelerator organization can be developed, verified, evaluated, and optimized, without building a physical prototype.

4.2.1 3-D Graphics Component Library

To facilitate the design exploration process, we have modeled in SystemC a library of OpenGL-compatible hardware modules that include all the Rasterizer Stage functionality and that can be plugged together to build a full-fledged graphics hardware accelerator. These models can be used as micro-architectural templates to support further refinement. The library amounts to approximate 28000 lines of SystemC code. The library has the following features:

- All modules have a fully parameterizable datapath by using SystemC/C++ templates;
- All modules (excepting the system interface) are described at RT-level with word-level specified operators in the datapath (they can be further refined to bit level in the final stages of implementations);
- All modules can be configured to support a tile-based rasterization approach [90];
- Aside modules that implement the OpenGL functionality, the following are provided:
  - modules for OpenGL state management to respect the OpenGL semantics;
  - modules for interfacing in SOC using OSCI TLM [73] (transaction level models);
  - modules for tile management;
  - finer-grain modules to implement various datapath operators.
  - pseudo-modules to implement performance-related counters or to allow graphical visualization, that can be enabled or disabled;

Using the library, the following information can be gathered during simulation:

- Number of frames generated per second;
- Number of primitives rasterized (shaded, antialiased, and textured);

- Number of fragments entering the per-fragment operations stage and the number of fragments discarded in each individual per-fragment operation sub-stage (pixel flow estimation);
- Total number of clock cycles to produce a frame or a number of frames;
- Number of the clock cycles, while the graphics hardware unit is busy processing and where these cycles were spent (rasterization setup, pixel fill engines, and texture units);
- Number of the clock cycles spent stalling the hardware units;
- Number of transactions and data traffic at the graphics accelerator—system interface;
- Frame buffer and local buffers utilization (number of reads, writes, and stalls);
- Run-time communication of graphics related data to graphical visualization modules (explained in Section 4.2.2);
- Hardware signals transitions in VCD (value change dump format) files for hardware debug and power consumption estimation.

All the statistics gathered can be employed to check the balance of the graphics hardware micro-architecture and to suggest changes in the next iterations in the design exploration process.

4.2.2 Visualization and Simulation Control

The GRAAL Simulator program was developed to provide a graphical front-end for SystemC simulation control, data visualization, and performance estimation in our graphics hardware accelerator design exploration framework. The implemented capabilities (see Figure 4.3) are the following:

- Simulation Control:
  - Start, pause, resume, and abort the SystemC simulation process at any time within the graphical environment;
  - Redirection to the graphical front-end of GRAAL Simulator program of all the messages normally displayed by the SystemC reference simulator.
- Real-time graphical data visualization with regard to SystemC simulation:
  - Visualization of various buffers content (screen frame buffer, tile color buffer, tile depth, and stencil buffers);

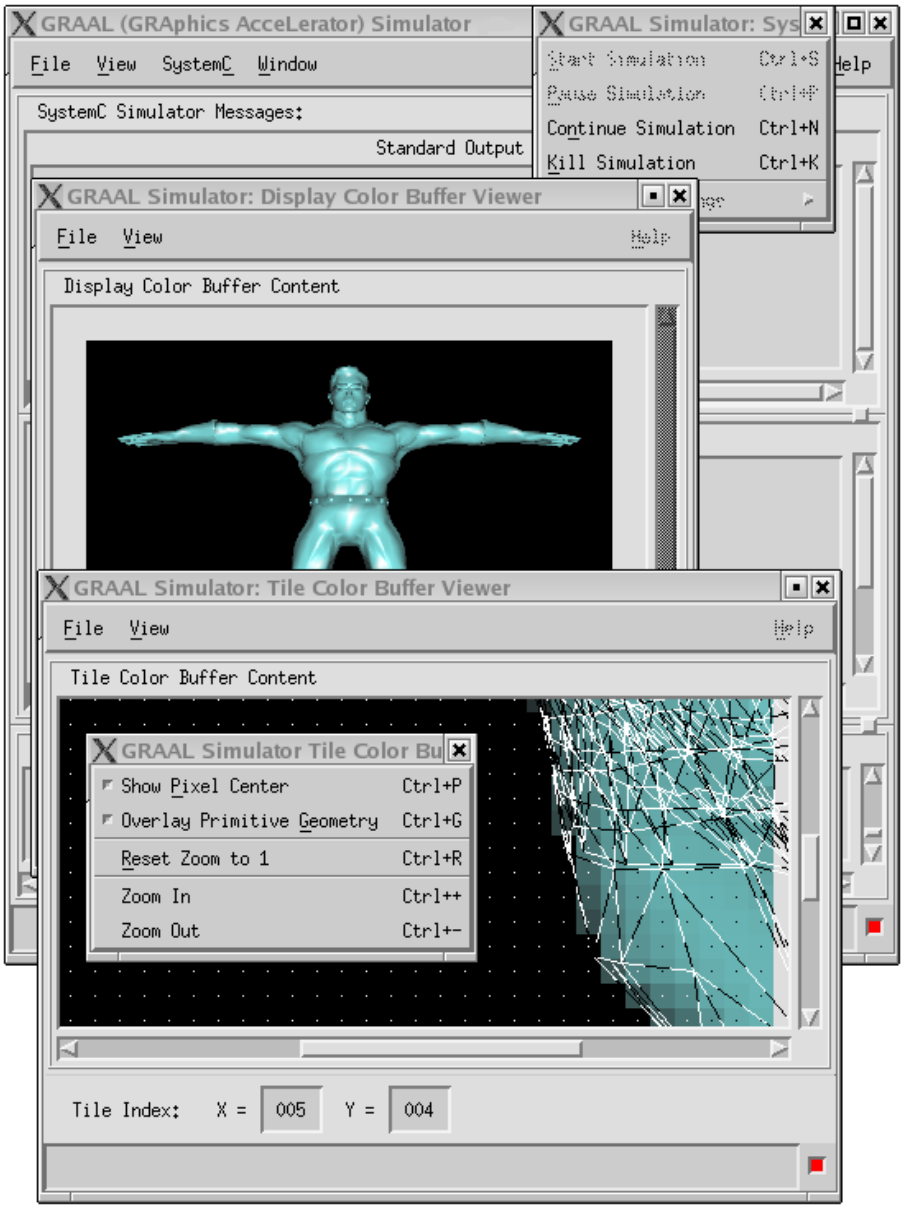

Figure 4.3: SystemC simulation control and graphical visualization in GRAAL Simulator program.

- Various levels of zoom;
- Markers overlapped on images for debugging purposes showing pixel centers, the rendering primitives whose image is rasterized, and/or tile boundaries;
- Image saving capability.

The GRAAL Simulator is a stand-alone program, implemented using the OSF Motif toolkit [41] for UNIX/X Window System workstations. The program amounts to approximately 18000 lines of C code. From an interface point of view, the GRAAL Simulator program launches the SystemC reference simulator (the executable SystemC system model) as an independent software entity. The ability to control the aspects already enumerated, disallowing the slightest change to the SystemC reference simulator source code, was very challenging, but the mechanisms provided by the system libraries of the UNIX operating system (spawning *child processes* with `fork()` and `execv()`, output redirection, and interprocess communication with *pipes* and *FIFOs* special files etc. [85]) and the OSF Motif toolkit API made the task possible. For data visualization, the pseudo-modules mentioned in Subsection 4.2.1 were employed to snoop relevant busses and to communicate the captured data via FIFO special files to the GRAAL Simulator program.

4.2.3 Energy/Power Consumption Estimation

Energy consumption is a critical factor in the system-level design of embedded portable appliances. A hardware-software co-design framework must be employed to proceed with the design from the software applications intended to run on these appliances to the final specifications of the hardware that implements the desired functionality, given the above-mentioned constraints. Studies have demonstrated that circuit- and gate-level techniques have less than a $2\times$ impact on power reduction, while architecture- and algorithm-level strategies offer savings of $10 - 100\times$ or more [62]. Hence, the greatest benefits are derived by trying to assess early in the design process the merits of the potential implementation. Architecture optimization corresponds to searching for the best design that optimizes all objectives. Since the optimization problem involves multiple criteria (power consumption, throughput, and cost), a set of Pareto points [68] in the design space have to be found to reach the global optimum. Ideally, when designing an embedded system, a designer would like to explore a number of architectural alternatives and test functionality, energy consumption, and performance, without the need to build a prototype first.

Usually, typical portable systems are built of commodity components and have a microprocessor-based architecture. Full system evaluation is often done on prototype boards resulting in long design times. Power consumption estimation can be done only late in the design process, after the prototype board has been built, resulting in slow power tuning turnarounds that do not meet the requirement of fast time to market. Moreover, using field programmable gate array (FPGA) hardware emulators for functional debugging, with a fast prototyping time, cannot give accurate estimates of energy consumption or performance.

In the last decade, among the tools preferred for early performance assessment at the algorithmic and architectural level, were the cycle-accurate instruction-set simulators. Unfortunately, for power consumption estimation, this approach was seldom easy to follow. There were only a few academic tools for power estimation (all based on or integrated in the SimpleScalar instruction set simulator toolset framework [17], [91], [16]) and almost no commercial products.

For several target general purpose processors, a number of techniques emerged in the last few years. The processor energy consumption for an instruction trace has generally been estimated by instruction-level power analysis [89], [88]. This technique estimates the energy consumed by a program by summing the energy consumed by the execution of each instruction. Instruction-by-instruction energy costs, together with non-ideal effects, are precharacterized once for each target processor. A few research prototype tools that estimate the energy consumption of processor core, caches, and main memory have been proposed [65], [61]. Memory energy consumption is estimated using cost-per-access models. Processor execution traces are used to drive memory models, thereby neglecting the non-negligible impact of a non ideal memory system on program execution. The main limitation of these approaches is that the interaction between memory system (or I/O peripherals) and processor is not modeled. Cycle-accurate register-transfer level energy estimation was proposed in [91]. The tool integrates RT level processor simulator with DineroIII cache simulator and memory model. It was shown to be within 15% of HSPICE simulations.

The drawback of all the above methods to estimate the power consumption is that they are based on certain architectural templates, i.e., general purpose processors, and can be hardly adapted to model system-on-chip designs.

The GRAAL design exploration framework offers two power estimation strategies, both based on SystemC simulation. Both strategies are estimating the

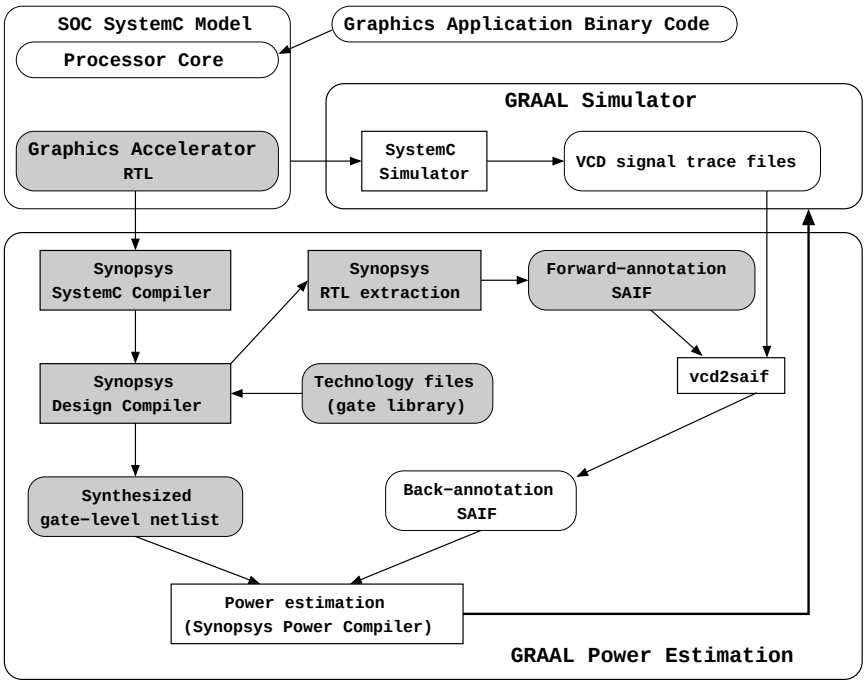

Figure 4.4: GRAAL netlist-level power estimation strategy.

average power consumption over the entire simulation period, providing as a byproduct the energy drawn by the graphics accelerator from the battery. The strategies are presented in the following.

Netlist-level Energy/Power Estimation

The first strategy [29] is depicted in Figure 4.4. It employs several Synopsys tools (CoCentric SystemC Compiler, Design Compiler, and Power Compiler) [87]. All the steps presented in the figure are automatized with custom-developed scripts for driving the tools. The prerequisites are a power precharacterized library of standard cells and an initial hardware synthesis step of the graphics accelerator RTL SystemC model to produce the gate-level netlist that will be used by the above-mentioned tools. The steps presented in a darker color have to be performed once for every candidate implementation of the graphics accelerator being decoupled from the benchmarking process. The RTL power consumption estimation process acquires information about

switching activity from SystemC RTL simulation. The switching activity is obtained by translating the VCD (Value Change Dump format) files, where the hardware signal traces are logged during SystemC RTL simulation, to SAIF (Switching Activity Interchange Format) files, the format recognized by the tools. Switching activity obtained from RTL simulation can be categorized as synthesis variant switching activity (SVSA) and synthesis invariant switching activity (SISA). SVSA comes from the combinational logic of the design that will be heavily optimized during the synthesis process. SISA comes from the synthesis invariant elements, which generally include inferred registers, inferred tri-state devices, hierarchical boundaries, black box IOs, and primary IOs. Since the synthesis process does not modify these elements, SISA information is still valid after the RTL design is mapped to the gate-level netlist. SVSA information is less accurate and is estimated statistically by propagating synthesis invariant element switching activity probabilities to the synthesis-optimized combinational logic trees. For the tool inter-operation details sketched in Figure 4.4, the reader is referred to [87]. The first strategy provides estimates reliable enough to be used in the micro-architecture exploration phase, where the difference in power estimates between two candidate micro-architectures in successive iteration steps is expected to be significantly different (greater than 100%).

Architecture-level Energy/Power Estimation

The second strategy utilizes the approach described in [30], requiring more technology-dependent data setup from the user, and is capable to deliver estimates within 25% of circuit-level simulations. The methodology of modeling power consumption at the architecture level is based on [62]. The premise for the success of such methodology lies in the existence of a library of hardware cells consisting of various operators for the datapath part, gates for control logic, and bit-cells, decoders, sense amplifiers for memory cores. Depending on the estimation accuracy desired, the individual cells can be specified at gate-level, if the gates employed are already characterized for power, or can be specified at the layout-level in order for the internal interconnect parasitics between individual constituent transistors to be accounted for. Once such a library exists, it can be precharacterized via gate-level, respectively circuit-level simulations, resulting in a table of effective capacitive coefficients for every element in the library. Then, using only this tables and the activity statistics derived during the architectural-level simulation, the power consumption can be estimated easily. This precharacterization has to be done only once and

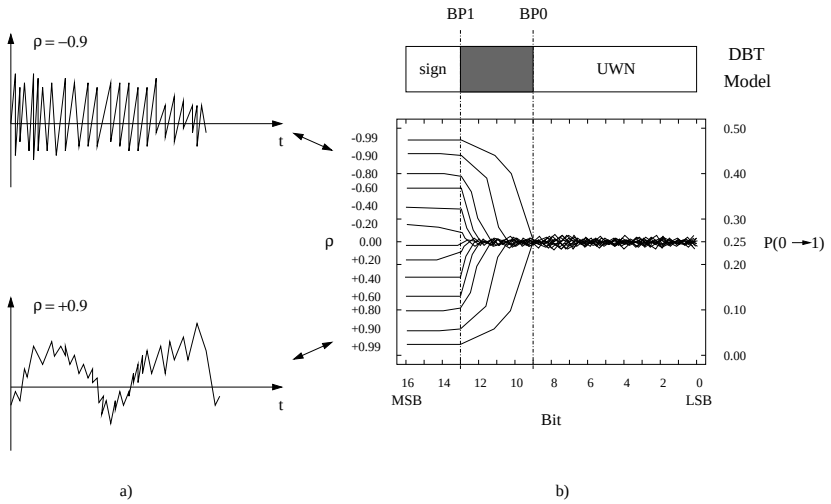

Figure 4.5: Bit transition activity for 2 two's complement data streams modeled as Gaussian processes with different temporal correlation ρ: a) Activity for positively and negatively correlated waveforms. b) Bit transition activity for data streams with varying temporal correlation.

only the effective capacitive coefficients table is needed for power estimation. The precharacterization results are valid only for a specific library of hardware cells and a given IC technology.

The power estimation methodology presented in [62] analyzes separately the four main classes of chip components: datapath, memory, control, and interconnect. For the first two classes, a model called the Dual Bit Type (or DBT) model was developed, demonstrating good accuracy results, with power estimates typically within $10 - 15\%$ of results from switch-level simulations. The DBT model achieves its high accuracy by carefully modeling both physical capacitance and circuit activity. The key concept behind the technique is to model the activity of the most significant (sign) bits and least significant bits separately, due to the fact that they exhibit different statistical behavior (as presented in Figure 4.5). The least significant bits are modeled as uniform white noise (UWN). The DBT model applies only to parts of the chip that manipulate data. A separate model is introduced to handle power estimation for control logic and signals. This model is called the Activity-Based Control (ABC) model. The method relies on the observation that although the implementation style of the controller, e.g., ROM, PLA, random logic, etc., can heavily

impact the power consumption, it is still possible to identify a number of fundamental parameters that influence the power consumption regardless of the implementation method. In a chip, datapath, memory, and control blocks are joined together by an interconnect network. The wires comprising the network have capacitance associated with them and, therefore, driving data and control signals across this network consumes power. The precise amount of power consumed depends on the activity of the signals being transferred, as well as the physical capacitance of the wires. The DBT and ABC models provide the activity information for control and data buses, but the physical capacitance depends on the average length of the wires in each part of the design hierarchy. The average length of the wires is estimated based on the Rent's rule [22].

Having the library of hardware cells specified, for example, at the layout-level the library is precharacterized first with a circuit-level simulator. During the precharacterization stage of the library of hardware cells, a black-box model of the capacitance switched in each module for various types of inputs is produced. If desired, these capacitance estimates can be converted to an equivalent energy, $E = CV^2$, or power, $P = CV^2f$. The black-box capacitance models can be parameterized, i.e., taking into account the size or complexity of the module. The model accurately accounts for activity as well as physical capacitance. As a result, the effect of the input statistics on module power consumption is reflected in the estimates.

For illustrative purposes, we will exemplify only briefly the modeling of the capacitance of a ripple-carry subtractor. Intuitively, the total power consumed by a module should be a function of its complexity, i.e., size. This reflects the fact that larger modules contain more circuitry and, therefore, more physical capacitance. The amount of circuitry and physical capacitance an instance of this subtractor will contain is determined by its word length, N. In particular, an N-bit subtractor can be realized by N one-bit full-subtractor cells. The total module effective capacitance function should receive an argument proportional to the word length, as shown here:

$$C_T = f\left(activity\_statistics, C_{eff}N\right) \tag{4.1}$$

where C_{eff} is the average capacitive coefficients per bit table, f represents the total module effective capacitance function, and C_T represents the total module effective capacitance. The average capacitive coefficients per bit table (presented in Table 4.1) is obtained after a process of data fitting (using least-squares approximation method), employing the effective capacitive coefficients tables generated for several sample width of the datapath, e.g., 4 bits, 8 bits, 16 bits, 32 bits. The average capacitive coefficients per bit table will be

Transition Templates	Capacitive Coefficients (fF/bit)			
UU/UU	35.12			
UU/SS	34.05	51.26	49.73	17.26
SS/UU	0.00	28.32	41.02	29.56
SS/SS/SS	0.00	28.33	41.17	3.15
	41.92	0.00	18.61	0.00
	52.13	25.71	0.00	0.00
	0.51	0.00	0.00	0.00
	0.00	35.42	0.00	52.12
	16.00	38.02	36.21	15.69
	0.00	55.79	0.00	0.00
	49.10	27.42	0.00	0.00
	0.00	0.00	69.21	60.01
	0.00	0.00	43.17	0.00
	7.65	32.73	48.82	10.13
	46.18	0.00	19.29	0.00
	0.00	0.00	0.00	2.41
	0.00	0.00	54.92	72.12
	0.00	0.15	0.00	46.89
	0.75	25.02	22.83	2.81

Table 4.1: Average capacitive coefficients per bit for the ripple-carry subtractor.

generated for the subtractor during the precharacterization stage and stored for the time when the estimate of the power consumption is needed.

With reference to Table 4.1, the number of coefficients that are required will depend on how many inputs the module has. In this case, the two-input module must be characterized on transitions on both inputs. The LSB region sees random (UWN) transitions on both inputs. The transition ID for this region of the module is written **UU/UU**. The effective capacitance of the module in this region will be described by the coefficient $C_{UU/UU}$. In the sign region, the module transition template has three components (**SS/SS/SS**) rather than two that might be expected. This is because the output sign can affect the capacitance switched in the module, and for some modules the sign of the inputs does not completely determine the output sign (in the case of the subtractor, subtracting two positive numbers could produce either a positive or a negative

result). If the transition template for the module only includes inputs, then a particular transition might be classified as ++/++. This makes it appear as though the module undergoes no activity even though it is possible that the output does make a sign transition. Specifying the output sign (e.g. ++/++/+-) avoids this ambiguity. For this reason, the transition template for the sign region of a two-input module should contain an entry for the output, as well as the inputs.

In addition to the subtractor, many modules follow a simple linear model for the argument of the total module effective capacitance function. For example, ripple-carry adders, comparators, buffers, multiplexers, and Boolean logic elements all obey Equation (4.1). The DBT method is not restricted to linear capacitance models and can model non-linear modules like array multipliers and logarithmic shifters. The total module effective capacitance function f is actually the power model of the module under consideration and receives the activity statistics seen on the module terminals, the complexity parameters of that module, e.g., N, and a pointer to the average capacitive coefficients per bit table for that module. The reader is referred to [62] for more details. The total module effective capacitance C_T represents the effective capacitance switched by that module every clock cycle during the execution of an application program on the host processor in the system-on-chip.

Figure 4.6 gives an overview of the architectural power analysis strategy that we propose. The inputs from the user are a description of a candidate architecture for the desired graphics accelerator in structural SystemC and the application program for which a power analysis is desired. Every clock cycle, the activity on internal relevant signals is collected and sent to the power analysis units. Rather than attempting to find a single power model for the entire chip, we take the approach of identifying four basic classes of components: datapath, memory, control, and interconnect. The total power consumption of the coprocessor or peripheral unit per program executed on the host processor is estimated.

The components relevant to the architectural power/energy estimation framework are:

- Precharacterized Power models and Effective Capacitance Coefficient Tables Module, that contain for a library of hardware cells all the technology dependent information required by the power analysis modules to compute the power consumption; the tables are derived only once for a given library of hardware cells;

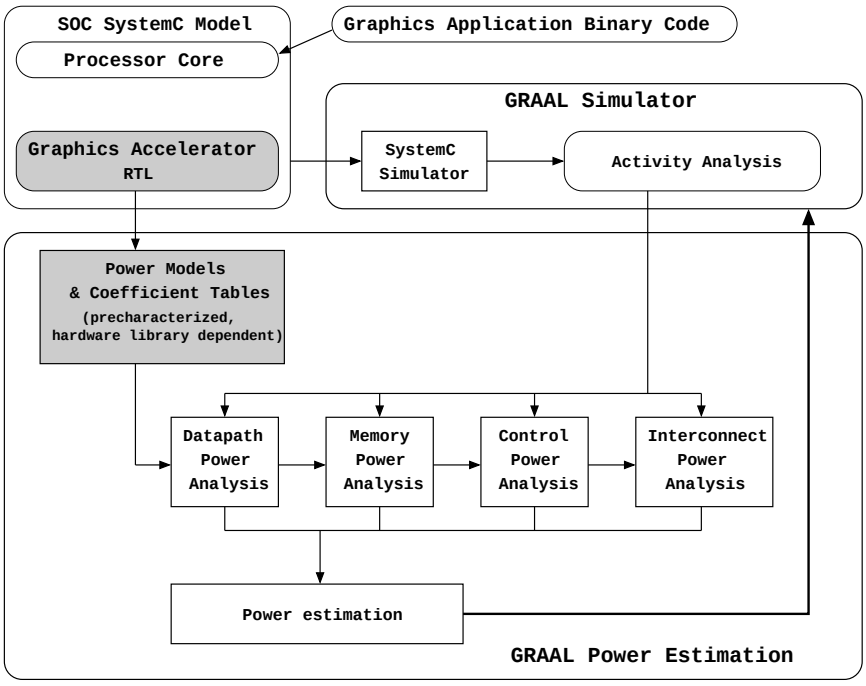

Figure 4.6: GRAAL architecture-level power estimation strategy.

- Activity Analysis Module that feeds the Power Analysis modules (power calculators) with statistics about signal activity inside the simulated hardware description;
- Power Analysis Modules that estimate the power consumption in the datapath, control, memory, and interconnect based on statistics received from the Activity Analysis Module and lookups in the effective capacitance coefficient tables;
- Power Estimator Module that adds the estimates of power consumption of datapath, control, memory, and interconnect and offers the total figure of power consumption in the graphics accelerator per program executed on the ARM processor;

The architecture-level power estimation strategy, although faster and more accurate, has the drawback of requiring a difficult pre-characterization process on existing libraries of hard-cells. Depending on the situation, either one or both of the power estimation strategies can be employed.

4.2.4 Graphics Benchmark Generation Process

Once the embedded graphics accelerator has been assembled from the library components, to gather data used in design exploration, a graphics library, a driver, and a graphics software application are needed. To this end, an OpenGL-compatible library was developed for the SOC platform, borrowing source code from MESA library [67] (an open source clone of the OpenGL library), together with the device driver for the graphics hardware accelerator. If the OpenGL software application is at source-code level, the only subsequent steps to be taken are to port the application to the SOC platform software environment, then to compile and link it with the OpenGL-compatible library, by using the native SOC platform application development tools. More often, interesting graphics applications with advanced features that can stress the graphics accelerator are available only in binary form, usually on PCs. For these situations, we dynamically linked the PC original binary applications against a modified Mesa library, which executes all the graphics pipeline in software, to trace and log all the graphics calls and relevant data in files. Then, a software player developed for the SOC platform is used to recreate all the original graphics calls and data from those files. This is actually substituted for the original graphics software application.

4.3 Case Studies

To assess the effectiveness of the GRAAL development framework (and to mainly illustrate the two power/energy estimation strategies described in Subsection 4.2.3), we designed two hardware circuits. The first one was implemented via synthesis from SystemC RTL and the other was obtained employing semi-custom techniques to generate the hard-cells required for the architecture-level power estimation strategy. The IC technology employed in both cases was a UMC 0.18μm Logic 1.8V/3.3V 1P6M GENERICII CMOS process.

4.3.1 Synthesizable Design

The first design implemented an OpenGL 1.2 compliant 3-D graphics hardware accelerator to be embedded in an ARM based SOC platform, using the SystemC module library described in Subsection 4.2.1. The following 3-D OpenGL rasterization functionality was incorporated in hardware:

- Triangle rasterization: flat- and Gouraud-shaded with/without antialiasing with all the options controlling rasterization;
- Texturing with only RGBA8 internal texture format, texture fetching on demand;
- Per-fragment operations: scissor test, alpha test, stencil and depth buffer test, blending, and logical operation;
- Whole frame buffer operations: fine control of buffer updates, clearing the buffers;
- State management: all the state management for previously mentioned functionality respecting all the invariance rules imposed by the OpenGL 1.2 specification;

The other primitives supported by OpenGL (points, lines, and polygons with more than three vertices) are processed by the software driver and presented to the graphics hardware accelerator as a combination of triangles.

Referring to the internal organization, the graphics accelerator adopts a tile-based rasterization approach. The tile size chosen for this particular implementation was set at 32×16 pixels, which implies that all the internal buffers (color buffer, depth buffer, and stencil buffer) composing the tile frame buffer (TFB) have this size. The display size resolution was set at 320×240 pixels (a quarter VGA), meaning that the display can be conceptually divided into 10×15 tiles. The graphics accelerator has only one pixel processing pipeline. The fixed-point formats utilized at the interface with the internal datapath are all unsigned. The screen coordinates (X, Y) are represented on 9.8 bits, the color components (R,G,B,A) on 0.8 bits, the depth component (Z) on 0.24 bits, and the stencil component on 8.0 bits.

One frame of the AWadvs-04 component of the OpenGL benchmark SPECViewperf $6.1.2$ [84] was generated on our virtual SOC platform, by the GRAAL tool framework. The resultant frame image is presented in Figure 4.3. A few characteristics of the frame workload on the graphics accelerator obtained by SystemC RTL simulation are presented in Table 4.2. The AWadvs-04 benchmark generates images with a higher geometric complexity than the typical graphical applications expected to be encountered on the low-power portable graphics terminals of the near future.

The results of the hardware synthesis on the graphics accelerator and estimated (netlist-level) average power/energy drawn from the battery per frame duration are presented in Table 4.3.

Processed Triangles	Fragments		Frame duration (clock cycles)
	processed	passed to color TFB	
15518	9510877	8759339	12168491

Table 4.2: Frame workload.

IC Technology		Std. Cell Library
UMC *Logic18-1.8V/3.3V-1P6M*		VST *e*Si-Route/11
Target Clk. Freq.	Std. Cell No.	Total Cell Area
200MHz	106k	2.44mm^2
Frame Estimated Average Power		Frame Estimated Energy
206mW		12.53mJ

Table 4.3: Graphics hardware estimation results.

GRAAL tool framework was run entirely on a Linux platform equipped with an AMD Athlon XP microprocessor running at 1.6GHz and 512MB of RAM. To assemble the graphics accelerator hardware from the provided library modules and to configure the software drivers, it took 16 man-hours, the hardware synthesis took 1hr 40min. This is one-time overhead, independent of the benchmarking process. The RTL SystemC simulation for the frame described took 15min, and the power and energy estimation took 15min. Both these figures depend on the image complexity to be generated on the graphics accelerator. However, when multiple graphics frames are processed, the simulation and the estimation can be overlapped in time. These means that data for approximatively 100 frames per day can be gathered, analyzed, and interpreted on one computer. Of course, the performance can be further improved by running the simulations on a cluster of computers.

4.3.2 Semi-custom Design

To verify the power consumption prediction accuracy of the architecture-level strategy, the following experimental setup should be provided: a precharacterized library of hardware cells, the description in SystemC of the hardware unit to be implemented, and the binary code of the program to be simulated on the host processor.

We precharacterized parts of a datapath library of cells (including a ripple-carry subtractor) designed in the same UMC 0.18μm Logic 1.8V/3.3V 1P6M

Instruction Trace	Power Consumption (estimated)	Power Consumption (simulated)	Relative Error (%)
A	0.77mW	0.91mW	-15
B	1.02mW	0.84mW	21
C	0.63mW	0.61mW	3

Table 4.4: Power consumption results for the ripple-carry subtractor.

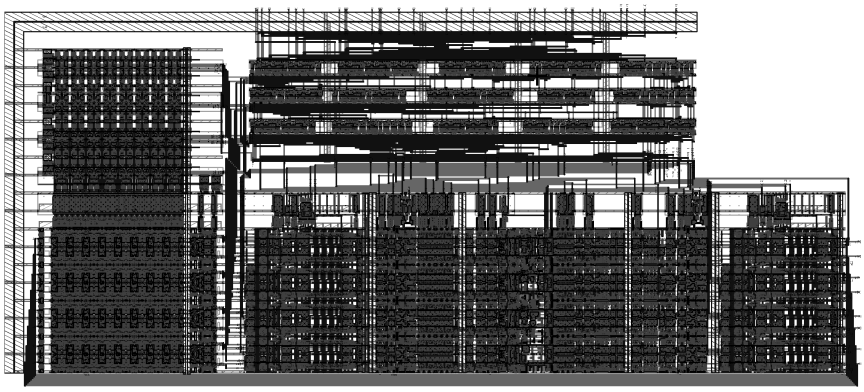

Figure 4.7: Sample coprocessor layout. From left to right and up to down: register file, control, and datapath.

GENERICII CMOS technology using Alliance VLSI CAD System [6]. We extracted from the layout the circuit of the subtractor in three variants of the datapath width: for 4 bits, 8 bits, and 16 bits. After using the architecture-level modelling method outlined in Subsection 4.2.3, we obtained the average capacitive coefficients per bit values presented in Table 4.1.

We modeled in SystemC a sample coprocessor for an ARM1020T CPU core. It was designed starting from the datasheet of AMD's Am2901 four-bit bipolar microprocessor slice. The coprocessor has a datapath width of 8 bits. The coprocessor consists of a 16-word by 8-bit two-port register file, an ALU and the associated shifting, decoding and multiplexing circuitry. The 9-bit microinstruction word is organized in three groups of three bits each and selects the ALU source operands, the ALU function, and the ALU destination register. The ALU provides various status flag outputs. The ALU is capable of performing three binary arithmetic ($R+S$, $S-R$, $R-S$) and five logic functions

(R OR S, R AND S, $\overline{R}$ AND S, R XOR S, R XNOR S).

To generate the application programs, we analyzed real trace data for environmental control realized with well known microcontrollers (Intel 8051 and compatible). We extracted the recurrent patterns of control and data in these instruction flows and generated three instruction flows A, B, and C, along with the data, using constrained pseudo-random generators specified with the aid of SystemC Verification Library API. We used the pseudo-random generators because in the case of our sample coprocessor no software applications exist. We executed these instruction flows on the ARM processor family ISA (storing the traces dumped by Armulator [10]) and, using the design framework and the traces, we obtained power consumption estimates for the subtractor. They are presented in the second column of Table 4.4.

In order to find the relative error of these estimations for the ripple-carry subtractor, we had to compare the results obtained employing our design exploration tool with the power consumption estimated accurately using HSPICE circuit simulator on exactly the same excitation patterns. For this purpose we designed the sample coprocessor down to the layout level, making use of the library of hardware cells generated beforehand. The layout of the sample coprocessor is presented in Figure 4.7. In order to use Alliance, the models were translated from SystemC to VHDL. The control circutry was synthesized, the register file was generated automatically by a macro generator, and the datapth was assembled in structural VHDL, with the glue logic between arithmetic operators synthesized by a datapath generator.

The simulation results on the extracted netlist of the ripple-carry subtractor are presented in the third column of Table 4.4. The clock frequency for the sample coprocessor assumed throughout these experiments is 200MHz. We have to mention here that the circuit-level simulation of the subtractor took several hours for the three instruction traces executed on the ARM processor. This clearly indicates that a circuit-level simulation of the whole coprocessor, to obtain the power consumption directly, for an instruction trace executed on the ARM processor is computationally unfeasible. The relative error between the power estimated and the power consumption obtained by circuit-accurate simulation is presented in the last column of Table 4.4. The power prediction accuracy is good, well within 25% of a direct circuit simulation with HSPICE.

4.4 Conclusion

In this chapter we have presented GRAAL, a versatile hardware/software co-simulation and co-design tool framework for embedded 3-D graphics accelerators. The tool framework offers a coherent development methodology based on an extensive library of parametrizable graphics pipeline components modelled at register transfer level (RTL) in SystemC. The framework is an open system, allowing integration with other third-party SystemC models of other various components (microprocessors, memories, and peripherals) to create an entire virtual simulation platform if desired. The framework incorporates tools to assist in the visual debugging of the graphics algorithms implemented in hardware, and to estimate the performance in terms of throughput, power consumption, and area. We have used the framework extensively and effectively throughout the project to assess the merits of various proposed hardware implementations and software/hardware partitioning algorithms.

Chapter 5

Efficient Antialiasing Coverage Mask Generation Hardware

In this chapter, an efficient low-cost, low-power hardware implementation of a run-time pixel coverage mask generation algorithm for embedded 3-D graphics antialiasing purposes is presented [33, 34]. The algorithm exploits the quadrant symmetry property allowing the storage of only the coverage mask information for a few representative edges in one of the quadrants of the plane, the rest of the information being derived on the fly via computationally inexpensive operations. The algorithm is presented assuming 4×4 subpixel coverage masks and two's complement number representation. However, it has a higher degree of generality: it can be incorporated in any antialiasing scheme with pre-filtering that is based on algebraic representation of primitive's edges, it is independent of the underlying number representation, and it can be adapted to other coverage mask subpixel resolutions with the only prerequisite for the masks to be square. In addition, the proposed hardware algorithm represents a natural extension of the algorithm presented in Chapter 3, Section 3.2.

The chapter is organized as follows. Background and preliminaries regarding antialiasing with prefiltering are presented in Section 5.1. In Section 5.2, the antialiasing coverage mask generation algorithm is introduced and highlights of its hardware implementation are discussed. The additional triangle setup reguired for triangle antialiasing is reviewed in Section 5.3. The required modifications of the triangle traversal algorithm are presented in Section 5.4. In Section 5.5, a qualitative analysis of the proposed algorithm is performed. The computational accuracy of the algorithm is investigated in Section 5.6. Finally,

hardware implementation and simulation results are presented in Section 5.7.

5.1 Background and Preliminaries

Antialiasing schemes can be classified in pre- and post-filtering methods (the reader is referred for an overview to Chapter 2, Section 2.3). The algorithm we propose is OpenGL-compliant. The antialiasing method suggested by the OpenGL specification, as presented in Subsection 2.3.2, is very close to the approach taken by pre-filtering (area sampling) antialiasing method.

Within this last category, one efficient approach for triangle rasterization and triangle antialiasing is based on the algebraic representation of triangle's edges with edge functions [43, 74] and normalized edge functions [78]. In hardware implementations for antialiasing with normalized edge functions, i.e. Exact Area Subpixel Algorithm (EASA) [78], subpixel representations of the pixel coverage coded in coverage masks (depicted in Figure 5.1(a)) are precomputed for various distances to the pixel center and angles of the edge and stored in a coverage mask lookup table (LUT). During rasterization, the normalized edge function computed for the current rasterization position (x_M, y_M) is interpreted as a distance and together with the edge slope is used as a LUT address to fetch the coverage mask of the current rasterization position for that edge (presented in Figure 5.1(b)). The table lookup is performed for all the three edges and the three resultant coverage masks are logically combined to produce a final coverage mask of the triangle over the current rasterization position (x_M, y_M). Then, the coverage mask is either employed as in [19] or used to compute a coverage value — the fraction of the pixel covered by the triangle — from the number of lit subpixels out of the total number of subpixels (see Figure 5.1(c)). Further, the coverage value is used to modulate the color (the transparency or the alpha value), which is also computed by interpolation for the current rasterization position (x_M, y_M).

The algorithm we propose can work in conjunction with various normalized edge functions from which the distance d from the pixel center and the angle α with the horizontal can be inferred from parameters of the normalized edge function (for illustration of the coverage mask generation algorithm we have used the normalized edge function proposed by Schilling [78]). This algorithm is independent of the underlying number representation and can be adapted to other coverage mask subpixel resolutions with the only prerequisite for the masks to be square. For illustrative purposes, the hardware circuits presented in figures employ two's complement arithmetic. For nu-

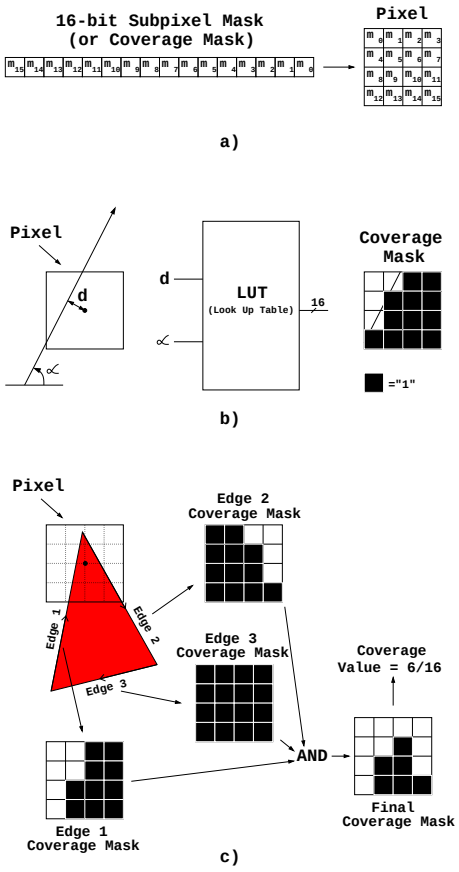

Figure 5.1: The basic principle of EASA: a) The 16-bit subpixel mask numbering scheme, b) Fetching a coverage mask from the lookup table using as indices the distance d of the edge vector from the pixel center and the angle α of the edge vector, c) The method to determine the coverage value of the pixel under consideration using box filtering by combining the coverage masks for the three edge vectors of the triangle.

merical bit-strings we utilize the following notation: $a^{\{\mathrm{sgn},n,...,0,-1,...,-m\}}$ or $a^{(\mathrm{sgn})}a^{(n)}...a^{(0)}a^{(-1)}...a^{(-m)}$ represents the fixed-point number whose value is $-a^{(\mathrm{sgn})}2^{n+1}+a^{(n)}2^{n}+...+a^{(0)}+a^{(-1)}2^{-1}+...a^{(-m)}2^{-m}$. The bit ranges presented in figures reflect the precision required in the antialiasing datapath of an embedded QVGA graphics accelerator. Thus, the numerical ranges for the antialiasing operands used throughout the paper will be $d_{L_1}^{\{\mathrm{sgn},10,...,-24\}}$,

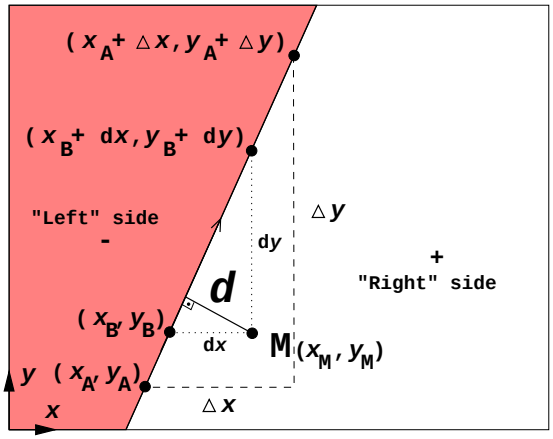

Figure 5.2: Computing the distance d of an arbitrary point M to the vector $\overrightarrow{(x_A, y_A)(x_A + \Delta x, y_A + \Delta y)}$.

$de_x^{\{\text{sgn},0,\ldots,-20\}}$, $de_x^{\{\text{sgn},0,\ldots,-20\}}$, $\Delta x^{\{\text{sgn},8,\ldots,-4\}}$, and $\Delta y^{\{\text{sgn},8,\ldots,-4\}}$.

In the following paragraphs, a connection between the edge function equation presented in Subsection 3.2.1, the distance d to the pixel center, and the angle of its defining vector with horizontal α is presented.

Referring to Figure 5.2, the distance d and the angle α can be expressed with the aid of the edge function $E(x, y)$ associated with the edge vector $\overrightarrow{(x_A, y_A)(x_A + \Delta x, y_A + \Delta y)}$ written for the point M(x_M, y_M). The distance d can be found by using similar triangles in Figure 5.2 as:

$$\begin{aligned} d &= \frac{\mathrm{d}x \cdot \mathrm{d}y}{\sqrt{\mathrm{d}x^2 + \mathrm{d}y^2}} = \frac{(x_B + \mathrm{d}x - x_B) \cdot \mathrm{d}y - (y_B - y_B) \cdot \mathrm{d}x}{\sqrt{\mathrm{d}x^2 + \mathrm{d}y^2}} \\ &= \frac{(x_M - x_B) \cdot \mathrm{d}y - (y_M - y_B) \cdot \mathrm{d}x}{\sqrt{\mathrm{d}x^2 + \mathrm{d}y^2}} = \frac{\widetilde{E}(x_M, y_M)}{\sqrt{\mathrm{d}x^2 + \mathrm{d}y^2}} \end{aligned} \tag{5.1}$$

where $\widetilde{E}(x, y)$ represents the edge function associated with the vector $\overrightarrow{(x_B, y_B)(x_B + \mathrm{d}x, y_B + \mathrm{d}y)}$ written for the point M(x_M, y_M).

Equation 5.1 expresses d as function of $\widetilde{E}(x, y)$ but given that during the triangle rasterization process we actually compute the $E(x, y)$ edge function, we would rather like to express d as function of $E(x, y)$. We can carry on this derivation if we utilize the edge function properties we proved in Subsection 3.2.1. To accomplish this, we have to note that using similar triangles in

Figure 5.2, it can be written that:

$$\frac{\mathrm{d}x}{\Delta x} = \frac{\mathrm{d}y}{\Delta y} = \frac{\sqrt{\mathrm{d}x^2 + \mathrm{d}y^2}}{\sqrt{\Delta x^2 + \Delta y^2}} = k \tag{5.2}$$

where k is a constant of proportionality. Now, using the Equations (5.1), (5.2) and the property presented and proved in Subsection 3.2.1 that the edge function is invariant with the origin of its defining vector along its associated line, the distance d can be written as:

$$\begin{aligned} d &= \frac{\widetilde{E}(x_M, y_M)}{\sqrt{\mathrm{d}x^2 + \mathrm{d}y^2}} = \frac{(x_M - x_B)\cdot \mathrm{d}y - (y_M - y_B)\cdot \mathrm{d}x}{\sqrt{\mathrm{d}x^2 + \mathrm{d}y^2}} \\ &= \frac{(x_M - x_B)\cdot k\cdot \Delta y - (y_M - y_B)\cdot k\cdot \Delta x}{k\cdot\sqrt{\Delta x^2 + \Delta y^2}} \\ &= \frac{(x_M - x_B)\cdot \Delta y - (y_M - y_B)\cdot \Delta x}{\sqrt{\Delta x^2 + \Delta y^2}} \\ &= \frac{(x_M - x_A)\cdot \Delta y - (y_M - y_A)\cdot \Delta x}{\sqrt{\Delta x^2 + \Delta y^2}} = \frac{E(x_M, y_M)}{\sqrt{\Delta x^2 + \Delta y^2}} \end{aligned} \tag{5.3}$$

For the sake of brevity, the above proof was presented assuming that $\Delta x > 0$ and $\Delta y > 0$, but this equation still holds for the general case. Rewriting the equation to let the angle α come into play, for any arbitrary point M it can be written:

$$\begin{aligned} d_{L_2}(\mathrm{M}) &= \frac{E(x_M, y_M)}{\sqrt{\Delta x^2 + \Delta y^2}} \\ &= (x_M - x_A)\cdot \frac{\Delta y}{\sqrt{\Delta x^2 + \Delta y^2}} - (y_M - y_A)\cdot \frac{\Delta x}{\sqrt{\Delta x^2 + \Delta y^2}} \\ &= (x_M - x_A)\cdot \sin\alpha - (y_M - y_A)\cdot \cos\alpha \end{aligned} \tag{5.4}$$

Regarding Equation (5.4), we would like to stress out the following:

First, the distance $d_{L_2}(\mathrm{M})$ can be regarded as a normalized edge function with an L_2 norm or Euclidean norm (the norm is $\sqrt{\Delta x^2 + \Delta y^2}$), and hence the notation for this distance. It can be proved that the *normalized edge function is invariant with the origin of its defining vector along its associated line* and the *normalized edge function is invariant with the length of its defining vector along its associated line* by following similar derivations as in Subsection 3.2.1. The former property is inherited from the edge function, but the last is a generalization of the property of the edge function.

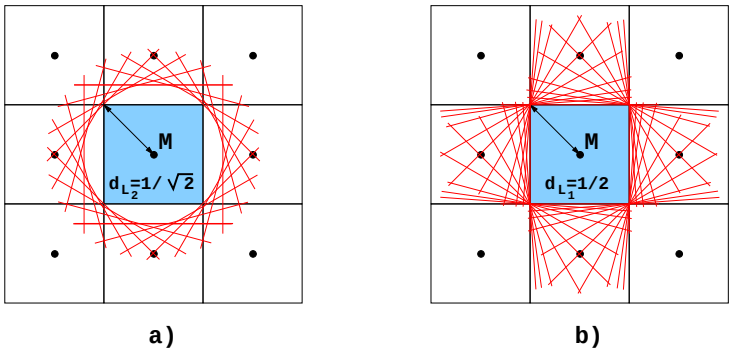

Figure 5.3: Geometrical locus of equidistant edges to a pixel center M (drawing reproduced from [78]): a) A circle for the Euclidean distance $|d_{L_2}(\mathrm{M})| = 1/\sqrt{2}$, b) The pixel square (approximative) for the L_1 norm distance $|d_{L_1}(\mathrm{M})| = 1/2$.

Second, looking at its expression, it follows that the distance $d_{L_2}(\mathrm{M})$ is a signed distance. Referring to the Figure 5.2 again, if the point M would be positioned in the other half-plane, the distance would be negative having as magnitude the Euclidean distance from the point M to the edge vector.

Third, the edge slope used to fetch the coverage mask from the LUT is represented by the dependency on $\sin\alpha$ and $sgn\{\cos\alpha\}$ (the last one being necessary in order to obtain a unique composite index for $\alpha \in [0, 2\pi]$).

Last, as explained in [78], the Euclidean distance has two disadvantages. The first one is a consequence of the fact that for every pixel a square subpixel mask is used. In Figure 5.3a, it is indicated that if all the pixels not more than $1/\sqrt{2}$ away from the edge are taken into consideration (to account for the subpixels positioned in the corners of the square represented by the pixel), it is possible for the edge to step on pixels other than the pixel (the rasterization's current position) for which the coverage value is supposed to be computed. This means additional complexity in the hardware, as the covering masks for neighboring pixels are also involved in the computation. The second disadvantage is that the computation of Euclidean distance is demanding (the square root is a costly operation in hardware).

To eliminate the above disadvantages of $d_{L_2}(\mathrm{M})$, we will consider, as in [78], the distance given by the normalized edge function with an L_1 norm or Man-

hattan norm $|\Delta x| + |\Delta y|$:

$$\begin{aligned} d_{L_1}(M) &= \tfrac{E(x_M,y_M)}{|\Delta x|+|\Delta y|} \\ &= (x_M - x_A)\cdot \tfrac{\Delta y}{|\Delta x|+|\Delta y|} - (y_M - y_A)\cdot \tfrac{\Delta x}{|\Delta x|+|\Delta y|} \\ &= (x_M - x_A)\cdot \sin\alpha \cdot \tfrac{\sqrt{\Delta x^2+\Delta y^2}}{|\Delta x|+|\Delta y|} \\ &\quad -(y_M - y_A)\cdot \cos\alpha \cdot \tfrac{\sqrt{\Delta x^2+\Delta y^2}}{|\Delta x|+|\Delta y|} \\ &= (x_M - x_A)\cdot de_x(\alpha) - (y_M - y_A)\cdot de_y(\alpha) \end{aligned} \tag{5.5}$$

It can be proved that the L_1 *normalized edge function is invariant with the origin of its defining vector along its associated line* and the L_1 *normalized edge function is invariant with the length of its defining vector along its associated line* by following similar derivations as in Subsection 3.2.1.

To demonstrate that indeed, by considering the distance $d_{L_1}(\mathrm{M})$ instead of the distance $d_{L_2}(\mathrm{M})$, all the previously mentioned disadvantages are eliminated, a brief justification is presented in the following.

To keep the material presented here-in at a reasonable length, in the following derivations only the case where $\Delta x > 0$ and $\Delta y > 0$ will be presented but the results can be easily generalized. The interest here is to find the geometrical locus of the edges for which the distance $d_{L_1}(\mathrm{M})$ to a given pixel with the center M is a constant. Because the Euclidean distance is easily comprehensible (is the "common sense" distance), the distance $d_{L_2}(\mathrm{M})$ of the edges belonging to this geometrical locus to the pixel center M will be computed. From Equations (5.4), (5.5) it can be written that:

$$\begin{aligned} d_{L_1}(\mathrm{M}) &= d_{L_2}(\mathrm{M})\cdot \frac{\sqrt{\Delta x^2+\Delta y^2}}{\Delta x+\Delta y} \\ &= d_{L_2}(\mathrm{M})\cdot \frac{\sqrt{1+\tan^2\alpha}}{1+\tan\alpha} \end{aligned} \tag{5.6}$$

Now, the distance $d_{L_1}(\mathrm{M})$ to the pixel with the center M for a 45° edge passing through the upper left corner of this pixel will be:

$$d_{L_1}(\mathrm{M}) = \frac{1}{\sqrt{2}}\cdot \frac{\sqrt{1+\tan^2 45^\circ}}{1+\tan 45^\circ} = \frac{1}{2} \tag{5.7}$$

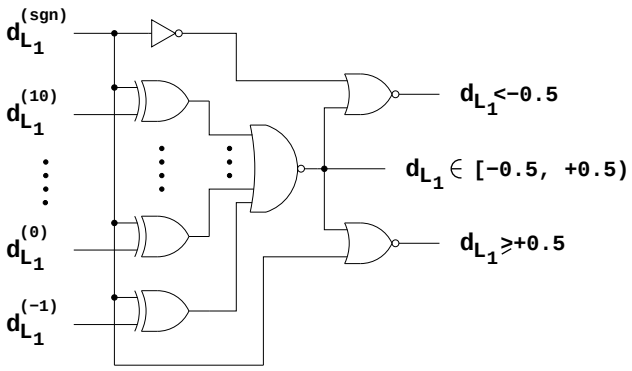

Figure 5.4: Efficient $\mathbf{d_{L_1}}$ range detector.

Now, by employing Equations (5.6) and (5.7), the geometrical locus of the edges for which the distance $d_{L_1}(\mathrm{M})$ to a given pixel center M is $1/2$ are the edges that have the following distance $d_{L_2}(\mathrm{M})$ to the pixel center M:

$$d_{L_2}(\mathrm{M}) = \frac{1}{2} \cdot \frac{1 + \tan\alpha}{\sqrt{1 + \tan^2\alpha}} \tag{5.8}$$

For instance, by plugging in Equation (5.8) for α the values $0°$, $30°$, and $60°$ the results obtained for $d_{L_2}(\mathrm{M})$ are $1/2$, $(\sqrt{3}+1)/4$, and $(\sqrt{3}+1)/4$.

Generalizing, the geometrical locus of the edges for which the magnitude of the distance $d_{L_1}(\mathrm{M})$ (keep in mind that $d_{L_1}(\mathrm{M})$ is a signed distance — it depends on the orientation of the edge vector) to a given pixel center M is $1/2$ is approximatively the pixel square. This is depicted in Figure 5.3b. This is in fact a desired result [78], because the previous Euclidean circular filter never would result in a homogeneous coverage of the screen. Moreover, the expression for the L_1 norm distance is simple enough to be computed during the rasterization process, and this L_1 norm distance will be considered for the computation of the distance and the angle information used in the coverage mask table look up.

To conclude, by referring to Equation (5.5), the required indices for table look up will be: the L_1 norm distance $d_{L_1}(\mathrm{M})$, $de_x(\alpha)$, and $sgn\{de_y(\alpha)\}$.

For practical interest, only the coverage masks for partially covered pixels have to be stored in the coverage mask LUT imposing the range for the L_1-norm distance to be $d_{L_1}(M) \in (-0.5, +0.5)$. Outside this range, the pixel is totally covered or totally uncovered by the triangle edge, and the coverage mask can

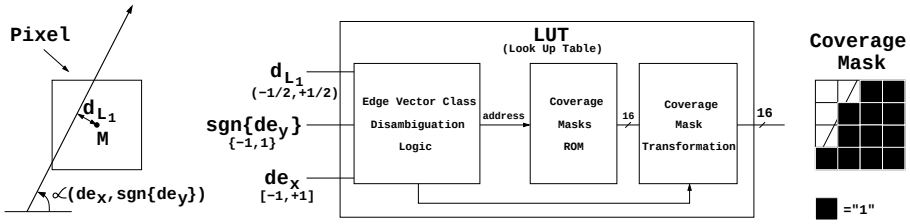

Figure 5.5: The indices and their range for coverage mask table look up.

be assigned implicitly to be with all subpixels set or unset depending on the sign of L_1-norm distance $d_{L_1}(M)$. This scheme can easily be implemented in hardware with trivial multiplexing circuitry using the two's complement circuit we propose in Figure 5.4.

Regarding the other index de_x, its range can be established easily. Employing Equation (5.5), the magnitude of de_x can be written as:

$$\begin{aligned} |de_x(\alpha)| &= |\sin\alpha| \cdot \frac{\sqrt{\Delta x^2 + \Delta y^2}}{|\Delta x| + |\Delta y|} \\ &= \sin\alpha_r \cdot \frac{\sqrt{1+\tan^2\alpha_r}}{1+\tan\alpha_r} = \sin\alpha_r \cdot \frac{\sqrt{\cot^2\alpha_r + 1}}{\cot\alpha_r + 1} \end{aligned} \quad (5.9)$$

in alternative form to avoid singularities in some legitimate cases (when edges are vertical or horizontal). Here $\alpha_r = \arctan(|\Delta y|/|\Delta x|)$ with $\alpha_r \in [0, \pi/2]$. Studying the extremes of the function described by Equation (5.9), it can be shown that the *range for $de_x(\alpha)$ is the interval* $[-1, 1]$.

The discrete range of $sgn\{de_y\}$ is given by the following assignment:

$$sgn\{de_y\} = \begin{cases} -1 & \text{if } de_y < 0 \\ +1 & \text{if } de_y \geq 0 \end{cases} \quad (5.10)$$

so the *range for $sgn\{de_y\}$ is the set* $\{-1, +1\}$.

Depicting the addressing scheme discussed in this subsection, the method of fetching the coverage masks for the partially covered pixels is presented in Figure 5.5. The functions performed by the blocks depicted in the figure will be explained in the next section.

To keep the coverage masks LUT within reasonable size, the edge vectors can be grouped in edge vector classes and only several representative classes can be stored in the coverage masks LUT. *An edge vector class is defined as a set*

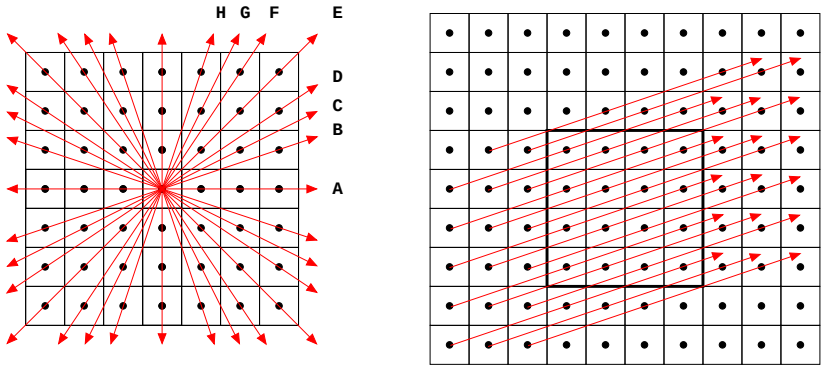

Figure 5.6: The edge vectors stored in the coverage masks LUT (the subpixels are represented as dotted squares).

of all the edge vectors with the same $de_x(\alpha)$ *and sgn* $(de_y(\alpha))$ *values, but with distinct* $d_{L_1}(M)$ *values (the values lie in the above ranges).* Hence, an edge vector class contains all the edge vectors with the same slope that partially cover a pixel. In the particular case of the EASA antialiasing scheme [78], only 32 edge vector classes from all the four quadrants of plane were stored as presented in Figure 5.6(a). The 32 edge vector classes were chosen by drawing all the possible edge vectors that were passing through the subpixel centers of a pixel (the edge vectors belonging to edge vector class B are depicted in Figure 5.6(b)). Then, the coverage mask that was stored corresponding to the index given by a combination of $d_{L_1}(M)$, $de_x(\alpha)$, and sgn $(de_y(\alpha))$ was computed by insuring that the number of subpixels lit in the coverage mask was correct plus or minus $1/2$ a subpixel, based on the exact covered area of the pixel. However, additional redundancy had to be incorporated in the LUT to ensure that for two adjacent triangles, both front-facing or both back-facing, a total coverage of more than 1 pixel was impossible and to counteract in two's complement number system the biasing of a rounding scheme based on truncation towards $-\infty$. This increased the coverage masks LUT size to 8k words of 16 bits (8k coverage masks) [78].

5.2 Proposed Coverage Mask Generation Scheme

By maintaining the same system parameters described in the previous section, we propose an algorithm that makes possible a reduction of coverage

mask LUT size to no more than 256 16-bit coverage masks, without downgrading the antialiasing quality. Considering that a triangle's oriented edge can be represented as a vector from the source vertex to the sink vertex, the 32 edge vector classes can be clustered according to the quadrant they belong (for horizontal/vertical edge vector classes a convention is made) as illustrated in Figure 5.7. Our algorithm proceeds from the consideration that the coverage masks required for the edge vectors of each of the four quadrants correspond to each other in a *rotationally symmetric* manner. That is, if an edge vector, which belongs by its orientation to one quadrant and which requires a specific coverage mask, is rotated in steps of $90°$, the resulting edge vectors in the other quadrants will require the same specific coverage mask rotated in corresponding steps of $90°$. *It is therefore proposed to store only coverage masks for edge vectors belonging by their orientation to one of the selected quadrants, e.g., to the first quadrant.* The coverage masks for edge vectors belonging by their orientation to another quadrant are obtained by a simple transformation of a coverage mask fetched for a corresponding edge vector belonging to the selected quadrant. Transposing the original edge vector into the selected quadrant and transforming the fetched coverage mask into the quadrant of the original edge vector can be achieved in hardware by computationally inexpensive operations, such as simple mask bitwise negations (an inverter per bit of coverage mask), mirrorings, and/or rotations with $90°$ (involving only the proper routing of signals representing the bits in the coverage mask).

The proposed algorithm for coverage mask generation for an edge vector that presents a partial coverage over the current rasterization position, as depicted in Figure 5.1(b), is presented in the following. For a correctness proof of the algorithm, the reader is referred to [32].

Algorithm

1. *Compute de_x, de_y for the edge vector and determine the initial quadrant for the edge vector (performed only once per edge);*
2. *Compute d_{L_1} for the current rasterization position that the edge touches;*
3. *Quadrant Disambiguation — perform the next operations if the initial quadrant for the edge vector is the following:*
   - *Q1:* $de_x^{Q1} = de_x$; $d_{L_1}^{LUT\_index} = d_{L_1}$
   - *Q2:* $de_x^{Q1} = -de_y$; $d_{L_1}^{LUT\_index} = d_{L_1}$
   - *Q3:* $de_x^{Q1} = -de_x$; $d_{L_1}^{LUT\_index} = -d_{L_1}$
   - *Q4:* $de_x^{Q1} = de_y$; $d_{L_1}^{LUT\_index} = -d_{L_1}$
4. *Edge Vector Class Disambiguation — Disambiguate the value for de_x^{Q1} using*

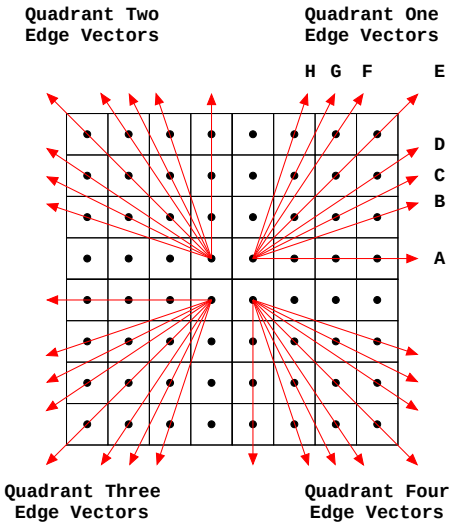

Figure 5.7: The new method of edge vector class clustering in the four quadrants of the plane (for clarity the edge vectors were drawn in four distinct pixels).

bisectors according to Table 5.1 thus producing a 3-bit $de_x^{LUT\_index}$ *value, if this disambiguation has produced a wrap-around set the* wrap *flag, else unset* wrap*;*

5. *Use 3-bit* $de_x^{LUT\_index}$ *value and 5 most significant bits of* $d_{L_1}^{LUT\_index}$ *to compose the address and fetch the coverage mask* Mask *from the coverage masks LUT;*
6. *Adjust if necessary the coverage mask* Mask *by producing an intermediary coverage mask* Adjusted_Mask*:*
   - *if* wrap *was set then perform:*
     $\text{Adjusted\_Mask} = \updownarrow (\text{Mask} \circlearrowleft 90°)$
   - *else perform:*
     $\text{Adjusted\_Mask} = \text{Mask}$
7. *If the initial quadrant for the edge vector was the following then compute another intermediary coverage mask* Coverage_Mask*:*
   - *Q1:* $\text{Coverage\_Mask} = \text{Adjusted\_Mask}$
   - *Q2:* $\text{Coverage\_Mask} = \text{Adjusted\_Mask} \circlearrowleft 90°$
   - *Q3:* $\text{Coverage\_Mask} = not\ (\text{Adjusted\_Mask})$
   - *Q4:* $\text{Coverage\_Mask} = not\ (\text{Adjusted\_Mask} \circlearrowleft 90°)$
8. *Compute the final coverage mask for the edge vector by testing the orientation*

of the triangle's edges:

- *if triangle's edges are oriented clockwise (*$d_{L_1}^{AB}(x_C, y_C) > 0$ *or* $E_{AB}(x_C, y_C) > 0$*) perform:*
  $\mathrm{Final\_Coverage\_Mask} = \mathrm{Coverage\_Mask}$
- *else*
  $\mathrm{Final\_Coverage\_Mask} = not\,(\mathrm{Coverage\_Mask})$

□

In the description of the algorithm, the operator $\circlearrowleft 90^\circ$ denotes a counter-clockwise rotation with 90° of the 4×4 grid of subpixels that is encoded as a 16-bit coverage mask, the operator $\updownarrow$ signifies a vertical mirroring of the 4×4 grid of subpixels, and the operator not() signifies a bitwise negation of the 16-bit coverage mask.

Due to the fact that the coverage mask LUT contains only instances of the quadrant one edge vector classes the indexing scheme became simpler when compared with previous implementations [78]: the index has to be composed taking into account only the transformed $d_{L_1}^{LUT\_index}(M)$ and $de_x^{Q1}(\alpha)$. Now the range for $de_x^{Q1}(\alpha) \in [0, +1)$ (the vertical edge vector class found at the intersection between quadrant one and two belongs according to the convention made to quadrant two) and the quadrant one edge vector classes can be distinguished from each other by the $de_x^{Q1}(\alpha)$ value only.

The algorithmic steps that are particular to the proposed algorithm are explained in the following and their implications for the hardware implementation are also discussed. The **steps 1, 2, 5** are almost identical with the steps that would be necessary in previous algorithms [78] with the exception that now the look up process is performed on a much smaller table with decreased access latency.

The quadrant determination of the initial edge vector specified by **step 1** can be implemented using the circuit presented in Figure 5.8. The 2-bit quadrant code assignment is "00" for quadrant one (Q1), "01" for quadrant two (Q2), "10" for quadrant three (Q3), and "11" for quadrant four (Q4). An additional error signal is provided to flag degenerate edge vectors ($\Delta x = \Delta y = 0$) and in effect to disable the rasterization of such degenerate triangles. As this circuit is already employed by the point-sampling triangle rasterization datapath to impose tie-rules for pixel rasterization on triangle shared edges, it will not be considered in the following as part of the antialiasing datapath.

The quadrant disambiguation (**step 3**) and the coverage mask transformation to the originary quadrant (**step 7**) are meaningful only if they are explained in

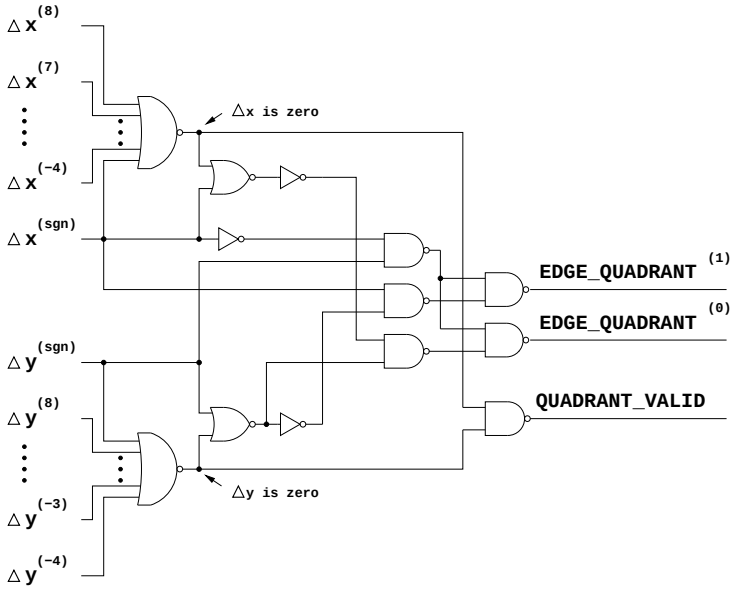

Figure 5.8: Edge vector quadrant computation.

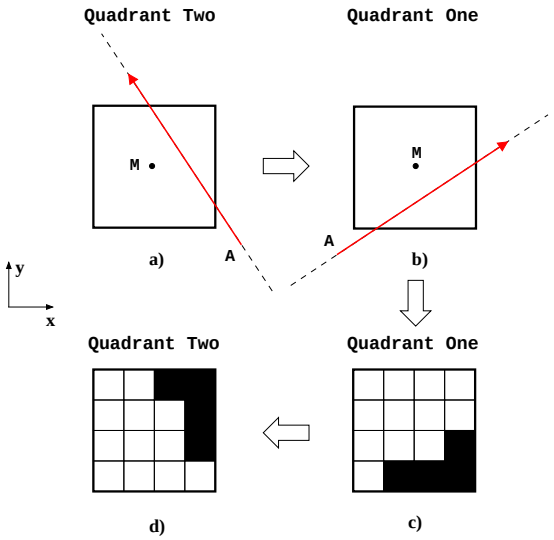

Figure 5.9: Q2 edge vector coverage mask generation.

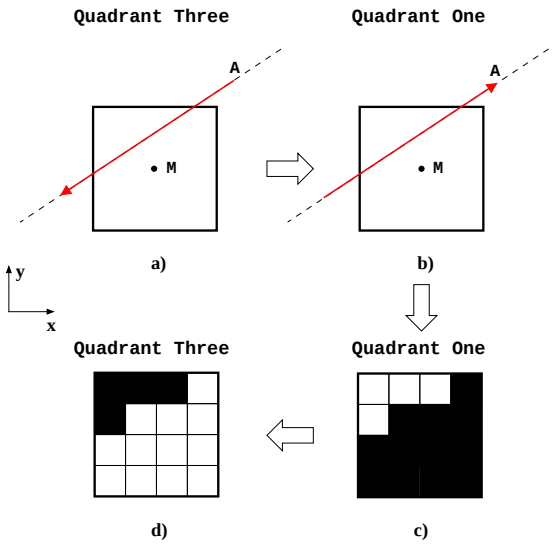

Figure 5.10: Q3 edge vector coverage mask generation.

synergy. The idea behind is to transform the arbitrary quadrant edge vector into an equivalent Q1 edge vector in order to use for coverage mask retrieval only a reduced coverage mask LUT for Q1 edge vectors. After the coverage mask is fetched from the LUT, inverse transformations have to be operated on the coverage mask in order to obtain the correct coverage mask for the initial, arbitrary quadrant edge vector. The equivalent underlying geometrical transformations to the above-mentioned formulas required to generate Q2 and Q3 coverage masks are depicted in Figure 5.9, respectively Figure 5.10. When the edge belongs to Q4, the operations required are fused computations Q4→Q2→Q1→Q2→Q4. The transformations for forward transition Q4→Q2 and backward transition Q2→Q4 are similar to Q3→Q1 and Q1→Q3, respectively. This forward/backward transformations ensure by construction that two adjacent triangles, both front-facing or both back-facing, always complement each other, and a total coverage of more than 4×4 subpixels is impossible, meaning that the algorithm is water-tight. In the following, efficient circuits to implement **step 3** are presented. There are two problems to be tackled with when using two's complement number representation. The first one is the requirement for wide-operand addition to implement the sign complementation unary operator (for our required precision 26-bit addition for d_{L_1} and 22-bit addition for de_x). The second one and the only mean to warrant water-

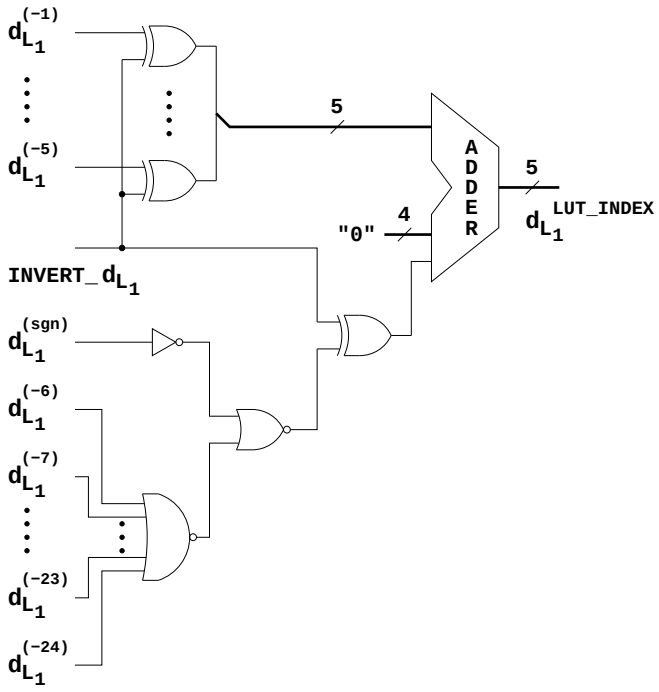

Figure 5.11: $\mathbf{d_{L_1}}$ selective sign complementation and truncate-to-zero circuit diagram.

tightness, given the asymmetrical behavior of positive and negative numbers under truncation (required for **steps 4, 5**), is to employ a truncate-to-zero rounding scheme. This is accomplished by ignoring the least significant bits on the right side and adding the sign bit to the least significant bit of the remaining bits. However this only occurs if at least one of the ignored bits is nonzero. This involves a chain of two additions, one of them being expensive. To simplify things and reduce it to two narrow-operand additions, it can be shown that a sign complementation followed by a truncate-to-zero rounding is equivalent to a truncate-to-zero rounding first followed by the sign complementation of the resultant reduced number of bits. Furthermore, it is possible to fuse these two additions in only one narrow-operand addition by using the circuits presented in Figure 5.11 and Figure 5.12 and employing little additional logic and a signal that indicates if the sign complementation is required. This circuits eliminate the need for additional redundancy to be built in the coverage masks LUT, as in [78], lowering the LUT foot-print further.

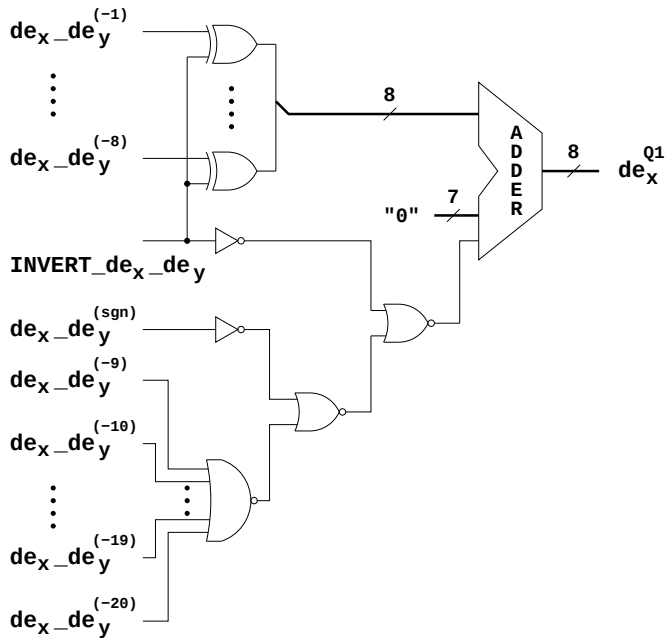

Figure 5.12: $\mathbf{de_x}$ or $\mathbf{de_y}$ selective sign complementation and truncate-to-zero circuit diagram.

Range $\mathbf{de_x^{Q1}}(\alpha)$	**Disambiguated** $\mathbf{de_x^{Q1}}(\alpha)$	$\mathbf{de_x^{LUT\_index}}$ **(binary)**	**wrap (binary)**
$\left[0, de_x^{bsct\_AB}\right)$	de_x^A	000	0
$\left[de_x^{bsct\_AB}, de_x^{bsct\_BC}\right)$	de_x^B	001	0
$\left[de_x^{bsct\_BC}, de_x^{bsct\_CD}\right)$	de_x^C	010	0
$\left[de_x^{bsct\_CD}, de_x^{bsct\_DE}\right)$	de_x^D	011	0
$\left[de_x^{bsct\_DE}, de_x^{bsct\_EF}\right)$	de_x^E	100	0
$\left[de_x^{bsct\_EF}, de_x^{bsct\_FG}\right)$	de_x^F	101	0
$\left[de_x^{bsct\_FG}, de_x^{bsct\_GH}\right)$	de_x^G	110	0
$\left[de_x^{bsct\_GH}, de_x^{bsct\_HV}\right)$	de_x^H	111	0
$\left[de_x^{bsct\_HV}, +1\right)$	de_x^A	000	1

Table 5.1: Edge vector class disambiguation rules.

The role of the edge vector class disambiguation (**step 4**) is to map the parameters of the quadrant one edge vector resulted from the previous step (quadrant

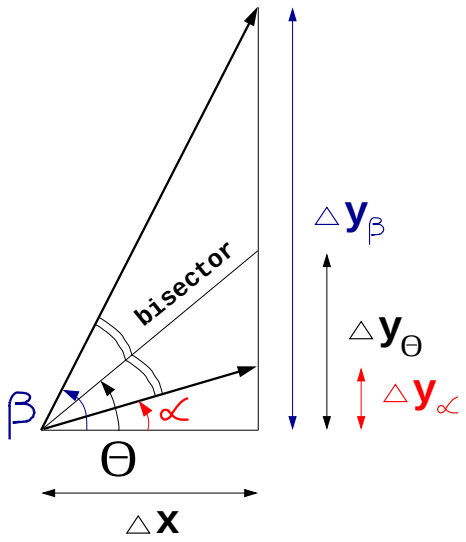

Figure 5.13: The bisector between two neighboring quadrant one edge vectors.

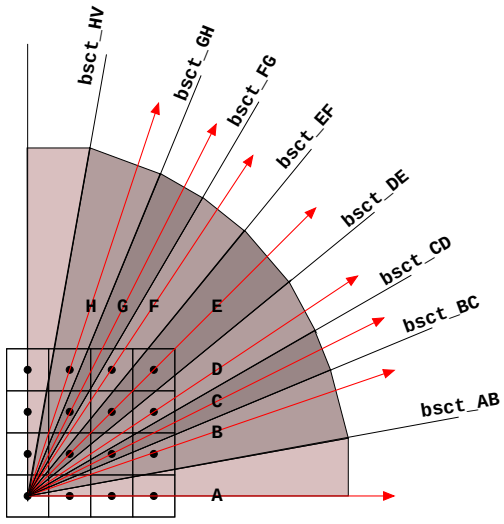

Figure 5.14: Edge vector class disambiguation employing bisectors.

disambiguation — **step 3**) into parameters of the closest matching representative edge vector whose coverage mask is resident in LUT. The quadrant one

edge vector that results after the quadrant disambiguation process (**step 2**) has to be classified in one of the eight quadrant one edge vector classes whose coverage masks are stored in LUT. Conceptually, the disambiguation process of the edge vector class is reduced to the problem of finding the boundaries between neighboring quadrant one edge vector classes with correspondence in the coverage masks LUT. This was solved by finding the $de_x^{Q1}(\alpha)$ value of the bisectors between two adjacent quadrant one edge vector classes with correspondence in LUT.

Two such edge vectors are isolated and depicted in Figure 5.13. For each bisector, a de_x value can be found. This de_x value represents the boundary between two neighboring edge vector classes. It can be computed using the following equations.

Referring to the Figure 5.13, it can be written that:

$$de_x^{\alpha} = \frac{\Delta y_{\alpha}}{\Delta x + \Delta y_{\alpha}} = \begin{cases} \frac{\tan\alpha}{1+\tan\alpha} & \text{if } \alpha \in [0, \pi/2) \\ 1 & \text{if } \alpha = \pi/2 \end{cases} \tag{5.11}$$

Similarly:

$$de_x^{\beta} = \frac{\Delta y_{\beta}}{\Delta x + \Delta y_{\beta}} = \begin{cases} \frac{\tan\beta}{1+\tan\beta} & \text{if } \beta \in [0, \pi/2) \\ 1 & \text{if } \beta = \pi/2 \end{cases} \tag{5.12}$$

The bisector's angle Θ can be written as a function of α and β as $\Theta = (\alpha + \beta)/2$. It follows that:

$$\begin{aligned} de_x^{\Theta} &= \frac{\Delta y}{\Delta x + \Delta y} = \frac{\tan\Theta}{1+\tan\Theta} = \frac{\tan\frac{\alpha+\beta}{2}}{1+\tan\frac{\alpha+\beta}{2}} \\ &= \frac{1}{\frac{1-\tan\frac{\alpha}{2}\cdot\tan\frac{\beta}{2}}{\tan\frac{\alpha}{2}+\tan\frac{\beta}{2}} + 1} \end{aligned} \tag{5.13}$$

Now, a modality has to be found to link the expressions for $\tan\frac{\alpha}{2}$ and $\tan\frac{\beta}{2}$ by their de_x^{α} and de_x^{β}. For instance, in the $\alpha's$ case it can be written that:

$$\tan\alpha = \frac{2\cdot\tan\frac{\alpha}{2}}{1-\tan^2\frac{\alpha}{2}} \tag{5.14}$$

which implies that:

$$\tan\frac{\alpha}{2} = \begin{cases} \frac{\sqrt{1+\tan^2\alpha}-1}{\tan\alpha} & \text{if } \alpha \in (0, \pi/2) \\ 1 & \text{if } \alpha = \pi/2 \\ 0 & \text{if } \alpha = 0 \end{cases} \tag{5.15}$$

From the Equation (5.11) it follows that:

$$\tan\alpha = \frac{de_x^\alpha}{1 - de_x^\alpha} \tag{5.16}$$

Plugging Equation (5.16) into Equation (5.15) yields:

$$\tan\frac{\alpha}{2} = \begin{cases} \dfrac{\left(\sqrt{1+\left(\frac{de_x^\alpha}{1-de_x^\alpha}\right)^2}-1\right)\cdot(1-de_x^\alpha)}{de_x^\alpha} & \text{if } de_x^\alpha \in (0,1) \\ 1 & \text{if } de_x^\alpha = 1 \\ 0 & \text{if } de_x^\alpha = 0 \end{cases} \tag{5.17}$$

Similarly, for β it can be written that:

$$\tan\frac{\beta}{2} = \begin{cases} \dfrac{\left(\sqrt{1+\left(\frac{de_x^\beta}{1-de_x^\beta}\right)^2}-1\right)\cdot(1-de_x^\beta)}{de_x^\beta} & \text{if } de_x^\beta \in (0,1) \\ 1 & \text{if } de_x^\beta = 1 \\ 0 & \text{if } de_x^\beta = 0 \end{cases} \tag{5.18}$$

Finally, by substituting Equations (5.17) and (5.18) in Equation (5.13), the associated de_x^Θ for the bisector can be computed.

Referring to Figure 5.14, it means that if the $de_x^{Q1}(\alpha)$ value of an incoming edge vector is, for example, between the $de_x^{Q1}(\alpha)$ values of the bisector `bsct_AB` and `bsct_BC`, then its $de_x^{Q1}(\alpha)$ value becomes that of the edge vector class `B`. Since only eight edge vector classes (`A`, `B`, `C`, `D`, `E`, `F`, `G`, `H`) are represented in the coverage mask LUT, it means that only 3 bits are needed to encode this value in the coverage mask LUT index. This 3-bit code is produced directly as a result of the edge vector class disambiguation with bisectors. In the coverage mask LUT being stored 256 coverage masks, 5 bits remain available (as in a previous implementation[78]) in the index to encode 32 L_1-norm distances $d_{L_1}(M)$ (coverage masks for 32 values of distances from the pixel center M to a particular edge slope can be stored). The rules for the edge vector class disambiguation with bisectors are presented in Table 5.1, column 1 and 2. It is needed to emphasize that the $de_x^{Q1}(\alpha)$ values associated with the bisectors represent constants to the algorithm which will be programmed in hardware and no computational effort is spent at rasterization time to compute them. The 3-bit code required to encode the disambiguated $de_x^{Q1}(\alpha)$ in the coverage mask LUT index is presented in Table 5.1, column 3. Referring to Figure 5.14, an exceptional case that has to be handled in a specific way appears for the disambiguation of any quadrant one edge vector class whose slope lies between

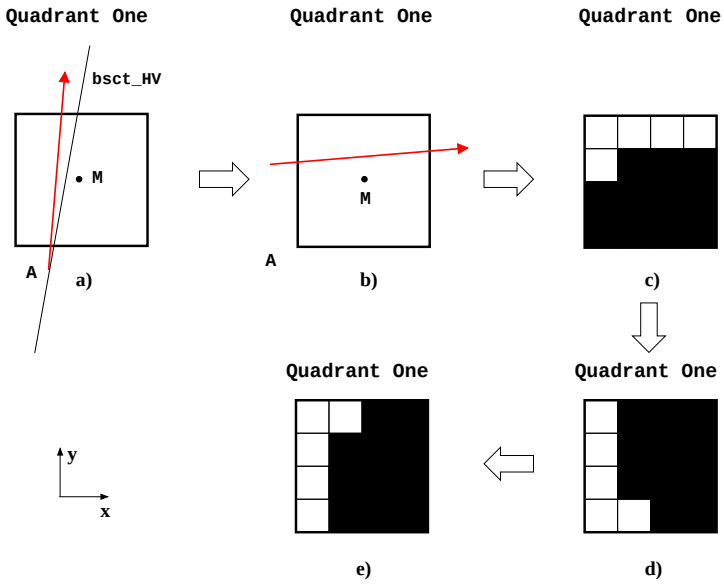

Figure 5.15: Coverage mask adjustment.

`bsct_HV` and the vertical. Normally, it will have to be disambiguated to the vertical edge vector class, but according to the assignment presented in Figure 5.7, this class belongs to the quadrant two. Instead, those exceptional edge vectors are disambiguated by wrapping around to the `A` edge vector class (last row in Table 5.1) and asserting a condition signal `wrap` (Table 5.1, column 4). The coverage mask is fetched from the coverage mask LUT, but before applying **step 7**, the correction described in **step 6** has to be performed if the condition signal is asserted. The equivalent underlying geometrical transformations for the coverage mask adjustment process are presented in Figure 5.15. The edge disambiguation rules presented in Table 5.1 can be implemented in two ways: sharing the gates for implementing the carry chains necessary for each required comparison or specifying the edge diambiguation rules in a logic table format with an entry for every possible $de_x^{Q1}(\alpha)$ value. Both approaches can be synthesized efficiently leading to a fast logic circuit, for example, considering 8-bit disambiguation constants, the resultant circuit complexity is slightly less than of a 16-bit adder.

Step 8 is required in order for the coverage mask lookup scheme to work with triangles with edges oriented clockwise or counter-clockwise, as required for OpenGL or Microsoft's DirectX-Direct3D compliance. The coverage masks

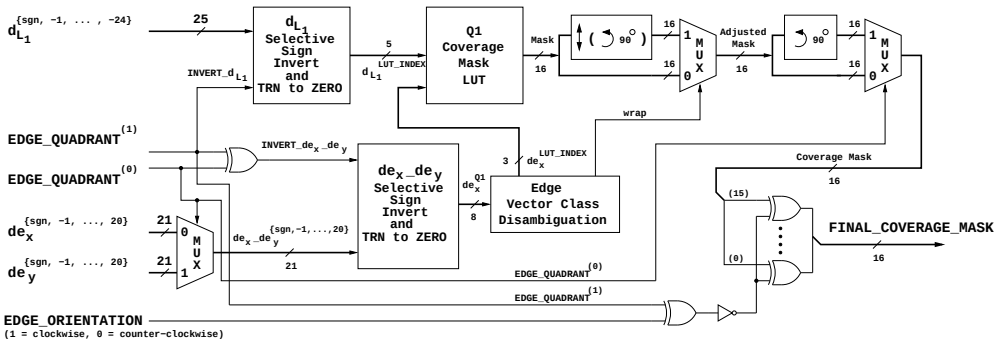

Figure 5.16: Coverage mask generation circuit diagram for one edge vector.

in the coverage mask LUT are computed only for clockwise orientation of triangle's edge vectors. For triangles with edges oriented counter-clockwise, the coverage mask obtained through the operations described so far has to be bitwise negated to deliver the final coverage mask. The orientation of the triangle's edges can be detected by computing in Equation (5.5) the sign of the edge function $E^{AB}(x_C, y_C)$, or equivalently, the normalized edge function $d_{L_1}^{AB}(x_C, y_C)$ (in any cyclic permutation of triangle's vertices A, B, C). Those computations are not specific to the antialiasing datapath, they being required mandatorily for the triangle interpolation setup, i.e., $\delta z/\delta x$, $\delta z/\delta y$ etc.

A diagram of the entire coverage mask generation circuit for one edge vector is presented in Figure 5.16. The diagram corresponds to Figure 5.1(b). To summarize, the proposed algorithm leads to efficient hardware implementations having a lower structural cost and requiring only computationally inexpensive operations.

5.3 The Additional Setup Required for Antialiasing

The required additional setup for the triangle antialiasing is described by the following relations:

$$
\begin{aligned}
mag\_x_{AB} &= |\Delta x_{AB}| \\
mag\_y_{AB} &= |\Delta y_{AB}| \\
mag\_x_{BC} &= |\Delta x_{BC}| \\
mag\_y_{BC} &= |\Delta y_{BC}| \\
mag\_x_{CA} &= |\Delta x_{CA}| \\
mag\_y_{CA} &= |\Delta y_{CA}| \\
mag\_sum_{AB} &= mag\_x_{AB} + mag\_y_{AB} \\
mag\_sum_{BC} &= mag\_x_{BC} + mag\_y_{BC} \\
mag\_sum_{CA} &= mag\_x_{CA} + mag\_y_{CA} \\
\\
Recip\_mag\_sum_{AB} &= \frac{1}{mag\_sum_{AB}} \\
\\
Recip\_mag\_sum_{BC} &= \frac{1}{mag\_sum_{BC}} \\
\\
Recip\_mag\_sum_{CA} &= \frac{1}{mag\_sum_{CA}} \\
\\
de_x^{AB} &= \Delta y_{AB} \cdot Recip\_mag\_sum_{AB} \\
de_x^{BC} &= \Delta y_{BC} \cdot Recip\_mag\_sum_{BC} \\
de_x^{CA} &= \Delta y_{CA} \cdot Recip\_mag\_sum_{CA} \\
de_y^{AB} &= \Delta x_{AB} \cdot Recip\_mag\_sum_{AB} \\
de_y^{BC} &= \Delta x_{BC} \cdot Recip\_mag\_sum_{BC} \\
de_y^{CA} &= \Delta x_{CA} \cdot Recip\_mag\_sum_{CA}
\end{aligned}
\tag{5.19}
$$

There are two strategies to compute the L_1 norm distances $d_{L_1}^{AB}$, $d_{L_1}^{BC}$, and $d_{L_1}^{CA}$ for a pixel M.

For the first one, it can be computed in the following way:

$$
\begin{aligned}
d_{L_1}^{AB}(x_M, y_M) &= E_{AB}(x_M, y_M) \cdot Recip\_mag\_sum_{AB} \\
d_{L_1}^{BC}(x_M, y_M) &= E_{BC}(x_M, y_M) \cdot Recip\_mag\_sum_{BC} \\
d_{L_1}^{CA}(x_M, y_M) &= E_{CA}(x_M, y_M) \cdot Recip\_mag\_sum_{CA}
\end{aligned}
\tag{5.20}
$$

by operating with the edge functions.

For the second one, it can be computed as follows:

$$\begin{aligned} d_{L_1}^{AB}(x_M, y_M) &= (x_M - x_A) \cdot de_x^{AB} - (y_M - y_A) \cdot de_y^{AB} \\ d_{L_1}^{BC}(x_M, y_M) &= (x_M - x_B) \cdot de_x^{BC} - (y_M - y_B) \cdot de_y^{BC} \\ d_{L_1}^{CA}(x_M, y_M) &= (x_M - x_C) \cdot de_x^{CA} - (y_M - y_C) \cdot de_y^{CA} \end{aligned} \quad (5.21)$$

by replacing the edge functions with L_1 normalized edge functions.

If the L_1 norm distance is written as a L_1 normalized edge function, then the distance d_{L_1} can be computed incrementally by simple addition for adjacent pixels:

$$\boxed{d_{L_1}(x+1, y) = d_{L_1}(x, y) + de_x} \quad (5.22)$$

$$\boxed{d_{L_1}(x, y+1) = d_{L_1}(x, y) - de_y} \quad (5.23)$$

Generalizing, the L_1 norm distance d_{L_1} for a pixel with the address $(x+\delta x, y+\delta y)$ can be computed from its value for the pixel (x, y) as:

$$\boxed{d_{L_1}(x+\delta x, y+\delta y) = d_{L_1}(x, y) + \delta x \cdot de_x - \delta y \cdot de_y} \quad (5.24)$$

using supplementary two multiplications and one addition.

Also, parallelizing a single L_1 normalized edge function computation is possible. By providing N hardware interpolators for the same L_1 normalized edge function, each interpolator can compute the L_1 normalized edge function in an interleaved fashion, for pixels a distance N away from a given pixel with the address $(x+i, y),\ i \in \{0, 1, \ldots, N-1\}$:

$$d_{L_1}(x+i+N, y) = d_{L_1}(x+i, y) + N \cdot de_x \quad (5.25)$$

where N must be a power of two for the reasons explained in Subsection 3.2.1.

5.4 Modifications of the Triangle Traversal Algorithm

The triangle traversal algorithm for aliased triangles was presented in Subsection 3.2.3. To accomodate antialiasing, only a modification of this algorithm is required by the new computing scheme to detect pixels (better said fragments) that are considered “interior” to the triangle. For these fragments, the associated data is mandatory to be computed (z, colors, and texture coordinates). The method to compute the associated data for the fragments remains unchanged.

The conditions required for a fragment (pixel) to be considered "interior" to the triangle are summarized in Table 5.2 for the two strategies presented in the Section 5.3.

For the first strategy, the aliased triangle's edge function approach is kept. This can lead to a shorter latency for the triangle rasterization setup. On the other side, the latency of the triangle rasterization will increase because, when the edges of the triangle will be encountered, in order to perform antialiasing at most three multiplications per fragment (pixel) will have to be additionally performed (it can be noticed with the aid of Equation (5.20)).

For the second strategy, in Section 5.3 it was suggested that the aliased triangle's edge function evaluation can be replaced completely by the L_1 normalized edge function evaluation. To compute Equations (5.21), (5.22), (5.23), and eventually (5.24), the same hardware as for the edge function evaluation can be employed. So what we suggest, is to use the edge functions for the aliased triangle rasterization, and the L_1 normalized edge functions (d_{L_1}) for the antialiased triangle rasterization. There are no concerns for the correctness of the algorithm, because all the algorithmic issues raised in the previous chapters hold equally for the L_1 normalized edge functions (the distance $d_{L_1}(x, y)$ is obtained from the edge function $E(x, y)$ by a division with a positive quantity). There is only one exception. For the triangle setup stage, presented in Chapter 3.2.2, the quantity $E_{AB}(x_C, y_C)$ has to be evaluated. For correctness, this quantity cannot be replaced with $d^{AB}_{L_1}(x_C, y_C)$. But this is not an issue, since the hardware required for the L_1 normalized edge function evaluation is the same as the hardware for the edge function evaluation. Under these conditions, the latency for the triangle rasterization setup will increase, but the latency of the triangle rasterization will decrease, by evaluating the L_1 normalized functions incrementally with additions for adjacent fragments (pixels).

It is true that, in most of the cases in computer graphics, only the throughput of the computations matters. There are a few exceptions and they are encountered only in the demanding real-time high-end graphics environments (e.g. interactive flight-simulators where the latency matters — the delayed visual feedback received by the user may lead to an overreaction from his part closing a positive feedback loop in the augmented system = user + graphics system, with negative consequences to the overall stability of the augmented system). For the rest of the low-end systems and in our case, however, this is not a crucial issue. Moreover, for a graphics system with limited interactivity, it is possible to hide out completely the latency for the triangle setup by overlapping the rasterization process of a triangle with the setup process for the next triangle,

Edge vector orientation	Condition for the "interior" fragment (pixel) M
$E_{AB}(x_C, y_C) > 0$	$\left[E_{AB}(x_M, y_M) > -\frac{1}{2} \cdot (\lvert\Delta x_{AB}\rvert + \lvert\Delta y_{AB}\rvert)\right]$ $\wedge \left[E_{BC}(x_M, y_M) > -\frac{1}{2} \cdot (\lvert\Delta x_{BC}\rvert + \lvert\Delta y_{BC}\rvert)\right]$ $\wedge \left[E_{CA}(x_M, y_M) > -\frac{1}{2} \cdot (\lvert\Delta x_{CA}\rvert + \lvert\Delta y_{CA}\rvert)\right]$ or $\left[d_{L_1}^{AB}(x_M, y_M) > -\frac{1}{2}\right]$ $\wedge \left[d_{L_1}^{BC}(x_M, y_M) > -\frac{1}{2}\right]$ $\wedge \left[d_{L_1}^{CA}(x_M, y_M) > -\frac{1}{2}\right]$
$E_{AB}(x_C, y_C) < 0$	$\left[E_{AB}(x_M, y_M) < +\frac{1}{2} \cdot (\lvert\Delta x_{AB}\rvert + \lvert\Delta y_{AB}\rvert)\right]$ $\wedge \left[E_{BC}(x_M, y_M) < +\frac{1}{2} \cdot (\lvert\Delta x_{BC}\rvert + \lvert\Delta y_{BC}\rvert)\right]$ $\wedge \left[E_{CA}(x_M, y_M) < +\frac{1}{2} \cdot (\lvert\Delta x_{CA}\rvert + \lvert\Delta y_{CA}\rvert)\right]$ or $\left[d_{L_1}^{AB}(x_M, y_M) < +\frac{1}{2}\right]$ $\wedge \left[d_{L_1}^{BC}(x_M, y_M) < +\frac{1}{2}\right]$ $\wedge \left[d_{L_1}^{CA}(x_M, y_M) < +\frac{1}{2}\right]$

Table 5.2: The condition that has to be satisfied for a fragment (pixel) to be considered "interior" to the triangle.

thus obtaining high throughput.

An additional comment is necessary to understand what a fragment (pixel) "interior" to a triangle means. A fragment (pixel) is "interior" to a triangle if the coverage value associated with that fragment in relationship with the triangle that is being rasterized is non-zero. This means that the triangle's edge(s) is touching that fragment. For this fragment, it is necessary to compute its associated data (z value, colors, and texture coordinates) for fragment occlusion test and color blending. Sometimes, the fragment's center may be outside of the triangle's boundary with at most half a pixel width and this may result in negative values or larger positive values (than in the case of the aliased triangle rasterization) for its associated data. This is the reason why some internal data formats described in Subsection 3.1 take into consideration two more bits (one for the sign).

5.5 A Qualitative Analysis of the Proposed Algorithm

The proposed antialiasing algorithm achieves all of the requirements of the OpenGL specification [80] (Chapter 3) that were quoted in Subsection 2.3.2. It fulfils triangle antialiasing.

The primary advantages of the algorithm are simplicity, cost and fill-rate performance. Only one sample is taken per pixel, so the frame buffer bandwidth and storage are essentially no different from a point-sampling approach (an additional alpha value is needed per pixel but this is not an issue due to the fact that an OpenGL-compliant rasterization engine have to be able to support the alpha channel). The controller hardware is also simple, meaning that the fill-rate performance need not necessarily be reduced when antialiasing mode is enabled.

Another advantage of the algorithm is that the intensity of a pixel covered by an edge changes gracefully, as the edge moves through the pixel. Thus, any jumps in intensity are limited only by the precision of the blending arithmetic. It avoids the artifacts manifested by the point supersampling antialiasing method regarding horizontal and vertical edges that were presented in Figure 2.16.

In addition, the algorithm, unlike the point supersampling antialiasing method, handles small objects very well. Each object's projection, no matter how small, contributes to those pixels that contain it, in strict proportion to the amount of the pixel's area it covers. Thus, the "popping" effect of small objects in movement, which is especially bothersome for the point supersampling antialiasing

method, is avoided.

Also, the "small triangle" case presented in Figure 2.12, which is a problem for the area sampling antialiasing method, is handled well by this algorithm. In addition, unlike the case of the point-supersampling algorithm where the end of a skinny triangle appears as a dashed line (Figure 2.17), in this case it appears smoothly.

To obtain realistic results for an antialiased scene containing many triangles, in OpenGL, the accumulation buffer (described in Subsection 2.3.2) can be employed, but this technique is very computation-intensive and therefore slower.

Also, instead of using the accumulation buffer method, one may choose to use the much more efficient procedure outlined in the OpenGL Programming Guide [93] (Chapter 6) (quoted in Subsection 2.3.2). A prerequisite for this procedure to succeed is that one has to perform a triangle sorting pass to send the triangles in a front-to-back order to the rasterization engine. This will not necessarily incur a performance penalty, because a tiling architecture (the case for the present architecture), requires a geometry sorting pass anyway at the software driver level to distribute the triangles per each tile. This means that the triangles can be sorted in a front-to-back order on the fly, as they are dumped one by one into the bin that keeps all of the triangles which will be sent to the rasterization engine for the current tile.

The only situation in a scene containing more than one triangle where the algorithm fails is the case of interpenetrating triangles of Figure 2.13. The algorithm works only on the polygon edges and does nothing about the interior of the polygons. However, intersecting triangles effectively create a high-frequency seam in the triangle interiors. Because the algorithm takes care only of one triangle at a time, it is incapable of detecting and smoothing such a seam as it would do for an edge. For this reason, when a scene with static objects is constructed, care has to be taken to avoid such "ill-behaved" triangles. Additionally, in a scene with dynamic objects, an algorithm for collision detection has to be put into place. These constraints, we believe, are not so drastic.

In computer graphics literature, there are solutions for full-scene antialiasing and order independent transparency, without requiring triangle sorting at the software driver level that can be adapted for our rasterization engine. Among them, we can mention [79], [90], [12] and [58].

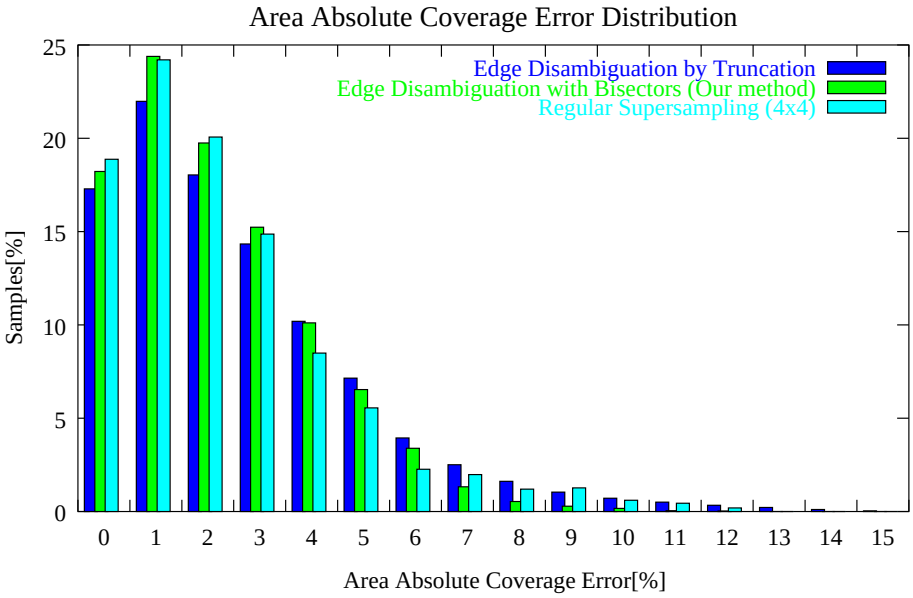

Figure 5.17: The area absolute coverage error distribution during hardware antialiasing employing the 8 edge vector classes, as proposed in [78] and presented in Figure 5.6.

Edge Vector Classes from [78]		
Method	**Maximum Error**	**Weighted Average Error**
Truncation	15.25%	2.64%
Edge disambiguation using bisectors	11.59%	2.23%
Regular Supersampling (4×4)	12.44%	2.34%

Table 5.3: The maximum area absolute coverage error and the weighted average of the area absolute coverage errors during hardware antialiasing employing the 8 edge vector classes, as proposed in [78] and presented in Figure 5.6.

5.6 The Computational Accuracy of the Proposed Antialiasing Scheme

We carried out investigations regarding the precision of the proposed antialiasing scheme. In particular, we studied the impact of the method chosen to

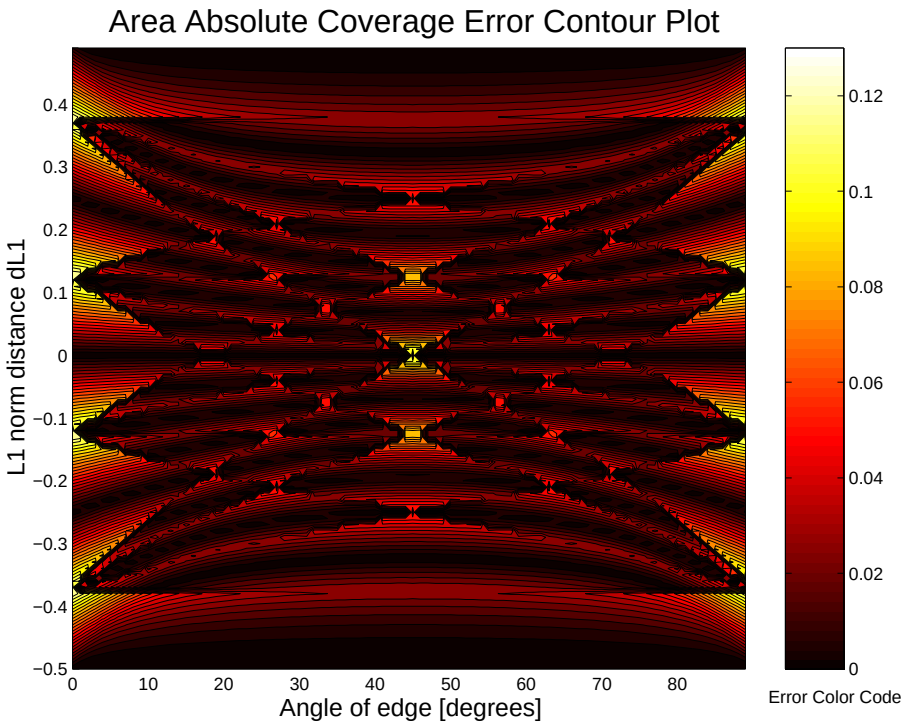

Figure 5.18: Contour plot of the distribution of the area absolute coverage error in the space angle(α) – L1 norm distance (d_{L1}) during hardware antialiasing employing 4×4 regular supersampling.

pre-compute the coverage masks (meant to be stored in the Coverage Masks LUT) for various sets of quadrant one edge vector classes on the area estimation accuracy. Additionally, we also investigated the most suitable position of these edge vector classes in the quadrant one space.

The proposed antialiasing scheme with coverage masks (256 16-bit coverage masks representing the pixel with 4×4 subpixel matrix) has two primary sources of errors: the first one — the *intrinsic* error — is given by the method chosen to pre-compute the coverage mask table content; the second one is given by the computations performed in the hardware antialiasing algorithm, i.e., the edge vector class disambiguation process.

In the following, the metric used to express quantitatively the error is the *area absolute coverage error* given by:

$$\varepsilon = \frac{|A^{exact}_{pixel\ covered} - A^{approx}_{pixel\ covered}|}{A_{pixel}} \tag{5.26}$$

where A_{pixel}, $A^{exact}_{pixel\ covered}$, and $A^{approx}_{pixel\ covered}$ represents the pixel area, the portion of the pixel area covered by the triangle's edge computed exactly, and the portion given by the output of the hardware antialiasing algorithm. All the results presented in this section take into consideration only one edge of the triangle, i.e., wherever a pixel exists over which two or three edges are intersecting (a very small triangle), the error given by the individual edges involved is additive.

Another metric of the precision of the hardware antialising scheme, hardly quantifiable, would be how closely the fetched coverage mask geometrically resembles the portion of the pixel covered by the triangle's edge. However, this matter was not neglected, the algorithm being designed and verified to produce close equi-morph coverage masks, thus making possible to implement triangle (Bartlett) filtering instead of box filtering as the last step in the hardware antialiasing algorithm.

5.6.1 An Intrinsic Error Reduction Method

Assuming a given set of quadrant one edge vector classes, the coverage masks that will be stored in the Coverage Mask LUT are pre-computed to minimize the area absolute coverage error for the instances of these edge vector classes. This is called in the following the intrinsic error of the antialiasing scheme.

The upper-bound of the intrinsic error was reduced by a coverage masks precomputation method adapted from [78]. The method is iterated for every instance edge vector of the quadrant one edge vector classes. For every instance edge vector, a coverage mask has to be computed. The method can be explained as follows. For a given instance edge vector, first the number of subpixels to be set in the coverage mask (n) is calculated with the formula:

$$n = \left\lfloor \frac{A^{exact}_{pixel\ covered}}{A_{pixel}} \times 16 + 0.5 \right\rfloor \tag{5.27}$$

Next, the subpixels of the coverage mask $m_0, \ldots, m_{15}$ are sorted from the most covered one to the least covered one (using the edge function for that instance edge vector). For subpixels with equal coverage, the subpixels are considered from the rightmost to the leftmost (to follow the heuristics we used

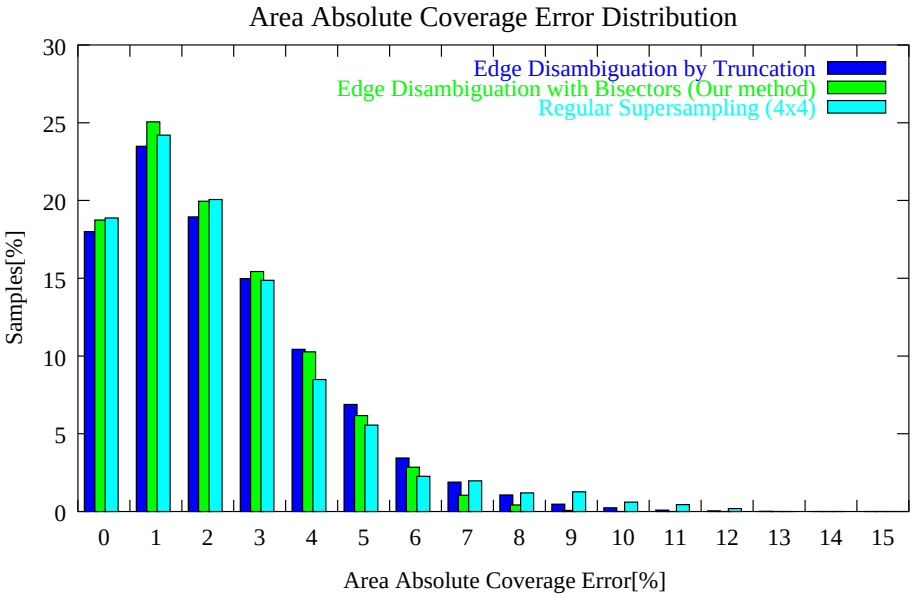

Figure 5.19: The area absolute coverage error distribution during hardware antialiasing employing 8 edge vector classes uniformly spread in the angular space of quadrant one.

8 Edge Vector Classes Uniformly Spread in the Angular Space of Q1		
Method	**Maximum Error**	**Weighted Average Error**
Truncation	12.55%	2.35%
Edge disambiguation using bisectors	8.99%	2.13%
Regular Supersampling (4×4)	12.44%	2.34%

Table 5.4: The maximum area absolute coverage error and the weighted average of the area absolute coverage errors during hardware antialiasing employing 8 edge vector classes uniformly spread in the angular space of the quadrant one.

so far to set subpixels for horizontal lines). Then, from this sorted list, the first n subpixels are set to 1 in the coverage mask, while all others are set to 0.

This method of pre-computing coverage masks guarantees that the upper-bound of the intrinsic error of the antialiasing scheme will be $1/32 \approx 3\%$,

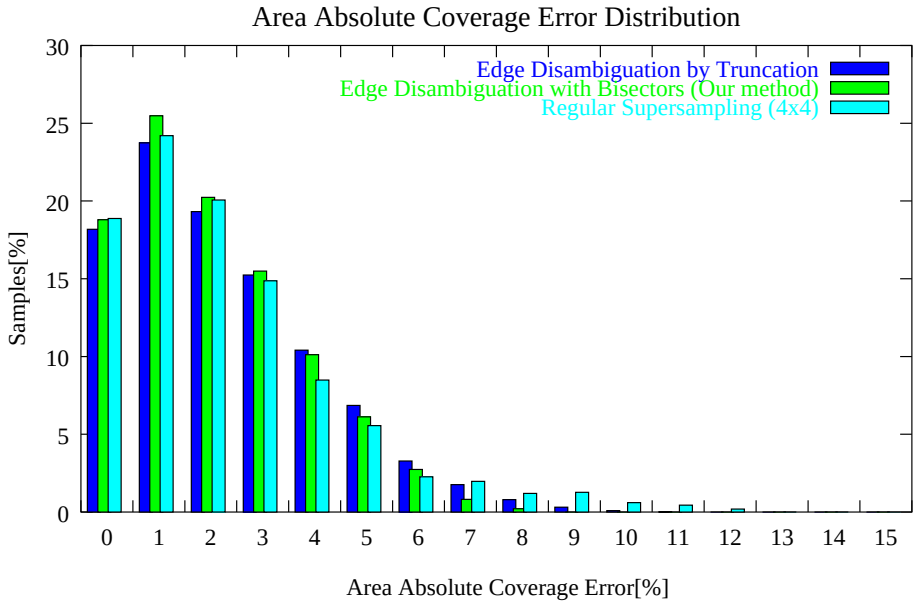

Figure 5.20: The area absolute coverage error distribution during hardware antialiasing, using 8 edge vector classes uniformly spread in quadrant one with regard to the hardware antialiasing algorithm input $de_x(\alpha)$.

8 Edge Vector Classes Uniformly Spread in the $de_x(\alpha)$ Space of Q1		
Method	**Maximum Error**	**Weighted Average Error**
Truncation	10.95%	2.28%
Edge disambiguation using bisectors	8.34%	2.09%
Regular Supersampling (4×4)	12.44%	2.34%

Table 5.5: The maximum area absolute coverage errors and the weighted average of the area absolute coverage errors during hardware antialiasing employing 8 edge vector classes uniformly spread in quadrant one with regard to the hardware antialiasing algorithm input $de_x(\alpha)$.

irrespective of the chosen set of quadrant one edge vector classes.

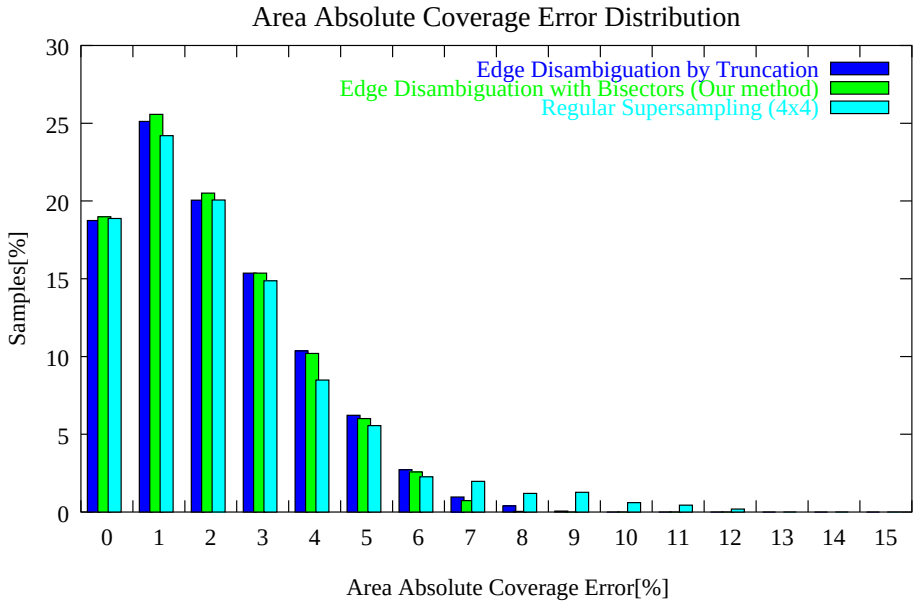

Figure 5.21: The area absolute coverage error distribution during hardware antialiasing employing 16 edge vector classes uniformly spread in the angular space of quadrant one.

5.6.2 Overall Error Reduction

Even if the intrinsic error of the antialiasing algorithm is minimized by the present method, the initial choice of the quadrant one edge vector classes still has a tremendous impact over the precision of the coverage mask look up process performed on the fly during rasterization.

Figure 5.6 presents the edge vector classes, as proposed in [78]. It can be seen that the edge vectors were chosen to pass through the center of subpixels (the pixel contour is emphasized in the figure) and, as it can be observed, they are not uniformly spread in the angular space (there is, for instance, a larger gap between edge vector classes A and B than the gap between edge vector classes B and C).

To investigate the overall area absolute coverage error using the antialiasing scheme during the actual rasterization process, a simple experiment was made. Edge vectors were generated by sweeping the entire space angle(α) – L1 norm distance (d_{L1}) for very small angular steps and very small distance increments

16 Edge Vector Classes Uniformly Spread in the Angular Space of Q1		
Method	**Maximum Error**	**Weighted Average Error**
Truncation	9.36%	2.12%
Edge disambiguation using bisectors	7.54%	2.06%
Regular Supersampling (4×4)	12.44%	2.34%

Table 5.6: The maximum area absolute coverage error and the weighted average of the area absolute coverage errors during hardware antialiasing employing 16 edge vector classes uniformly spread in the angular space of the quadrant one.

16 Edge Vector Classes Uniformly Spread in the $de_x(\alpha)$ Space of Q1		
Method	**Maximum Error**	**Weighted Average Error**
Truncation	8.05%	2.08%
Edge disambiguation using bisectors	7.49%	2.02%
Regular Supersampling (4×4)	12.44%	2.34%

Table 5.7: The maximum area absolute coverage error and the weighted average of the area absolute coverage errors during hardware antialiasing employing 16 edge vector classes uniformly spread in quadrant one with regard to the hardware antialiasing algorithm input $de_x(\alpha)$.

in the interval $[0, 90^\circ) \times (-0.5, 0.5)$. Every such generated edge vector was fed to the hardware antialiasing algorithm, and the coverage mask produced by the lookup process was compared with the exact area covered by the edge over the pixel. The results are presented in Figure 5.17 as an error distribution. The three superimposed histograms were generated 1) using the method from [78] with truncation, 2) the algorithm we proposed with edge disambiguation using bisectors, and 3) a pure 4×4 regular supersampling process. The maximum area absolute coverage error and the weighted average of the area absolute coverage errors over the entire space of the experiment are presented in Table 5.3. The regular supersampling process will serve as a reference point for our experiments.

Although from the results presented the regular supersampling process appears to behave better than the algorithm from [78], this is not the case because, (as

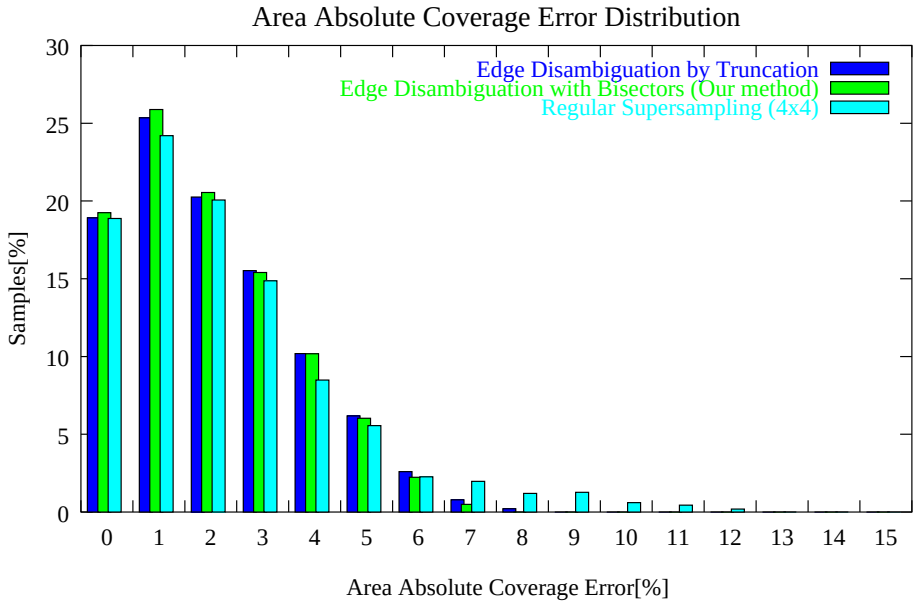

Figure 5.22: The area absolute coverage error distribution during hardware antialiasing using 16 edge vector classes uniformly spread in quadrant one with regard to the hardware antialiasing algorithm input $de_x(\alpha)$.

can be observed from the spatial distribution of the errors presented in Figure 5.18), the supersampling process behaves poorly for horizontal, vertical, and diagonal edges (the neighborhood of these particular edges serves as a collecting point for all the significant errors in the regular supersampling scheme).

In order to further reduce the errors manifested by our method, other edge vector class setups were chosen consistent with the observation made before that the initial edge vector classes were positioned non-uniformly in the angular space. The experiment was repeated choosing the edge vectors of the edge vector classes to be uniformly spread in the angular space of the quadrant one. We would like to stress out that, in this new comparison experiment and the rest of the experiments presented in the remainder of this section, the truncation method also utilizes the uniformly distributed set of edges that it is not any longer the algorithm presented in [78]. The histograms are presented in Figure 5.19, and the maximum area absolute coverage error and the weighted average of the area absolute coverage errors are presented in Table 5.4.

The error can be reduced further by repeating the experiment with the edge

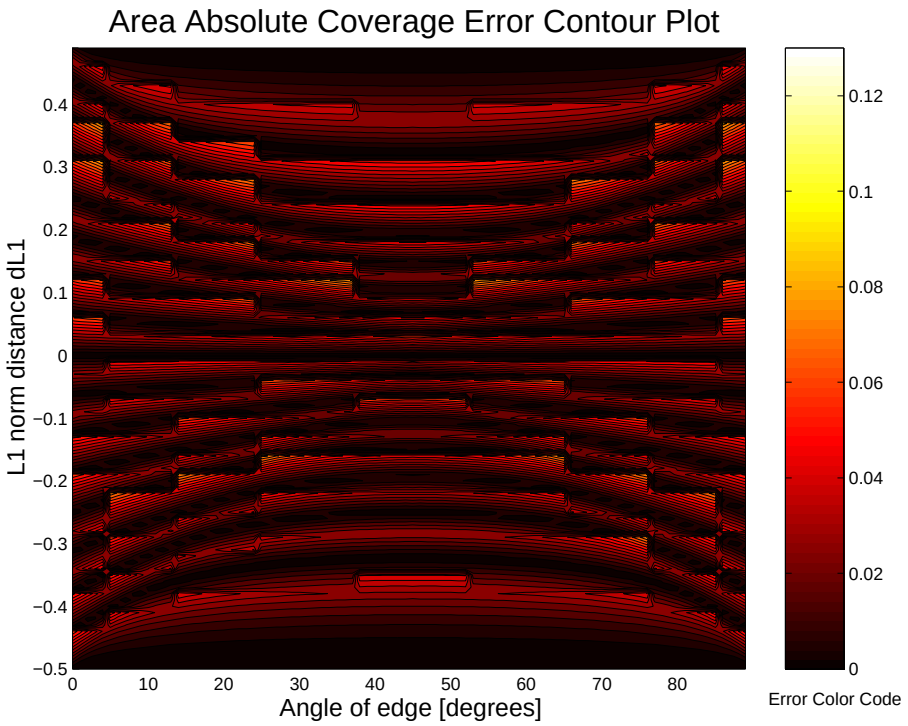

Figure 5.23: Contour plot of the distribution of the area absolute coverage error in the space angle(α) – L1 norm distance (d_{L1}) during hardware antialiasing using 8 edge vector classes uniformly spread in quadrant one with regard to the hardware antialiasing algorithm input $de_x(\alpha)$.

vector classes uniformly spread in quadrant one with regard to the hardware antialiasing algorithm input $de_x(\alpha)$. The histograms are presented in Figure 5.20, and the maximum area absolute coverage error and the weighted average of the area absolute coverage errors are presented in Table 5.5. The results in this case are better than the results of the previous case (edge vector classes spread uniformly in the angular space of the quadrant one).

For completness, the experiment was repeated by increasing the number of edge vector classes from 8 to 16, at the cost of doubling the coverage mask table size and increasing the complexity of the edge disambiguation unit. The results are presented in Figure 5.21, Table 5.6, Figure 5.22, and Table 5.7. The gains in accuracy are minor compared with the costs involved, even if we consider to implement the scheme with 16 edge vector classes uniformly

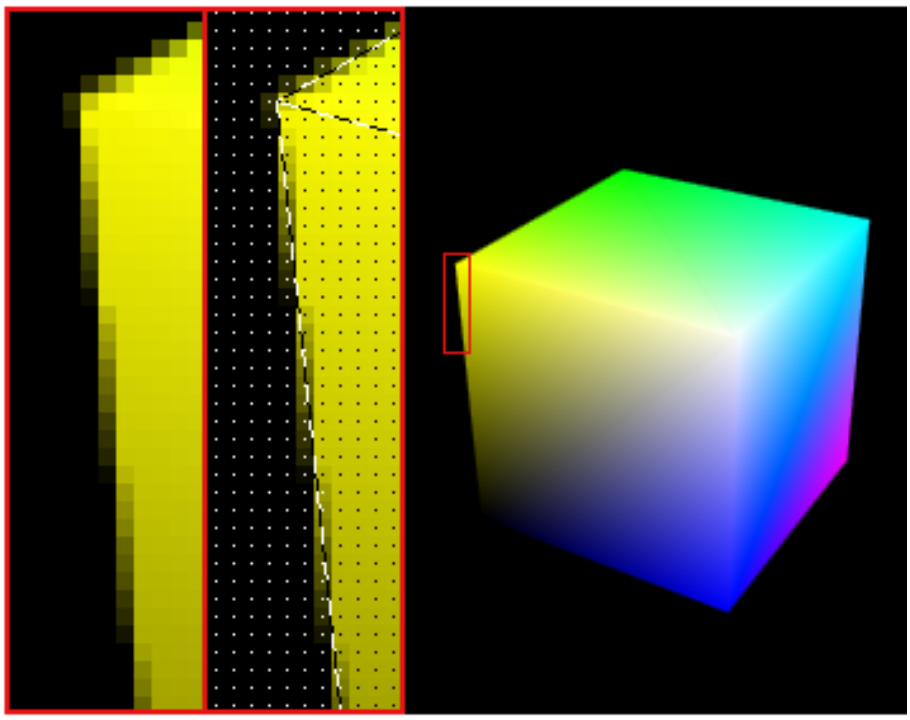

Figure 5.24: Antialiasing employing the proposed coverage mask generation hardware algorithm and implementation.

spread in the de_x space with truncation (without edge disambiguation with bisectors).

Given the fact that the schemes based on 16 edge vector classes do not provide a significant error reduction, we can conclude that the best candidate for the implementation in the rasterization engine is edge disambiguation employing bisectors with 8 edge vector classes uniformly spread in quadrant one with regard to the hardware antialiasing algorithm input $de_x(\alpha)$. The results for this scheme were presented in Figure 5.20 and Table 5.5. A contour plot of the distribution of the area absolute coverage error in the space angle(α) – L1 norm distance (d_{L1}) is presented in Figure 5.23. In comparison with the contour plot of the area absolute coverage error of the 4×4 regular supersampling approach depicted in Figure 5.18, it can be noticed that the errors are more uniformly distributed (also the maximal errors are smaller) and thus assuring a better antialiasing quality of edges at all angles.

IC Technology		Std. Cell Library
UMC *Logic18-1.8V/3.3V-1P6M*		VST *e*Si-Route/11
ED Latency	**ED Std. Cell No.**	**ED Cell Area**
0.5ns	42	$833\mu\text{m}^2$
Total Latency	**Total Std. Cell No.**	**Total Cell Area**
2.49ns	557	$12270\mu\text{m}^2$

Table 5.8: Hardware synthesis results for the coverage mask generation circuit for one edge vector.

5.7 Hardware Implementation and Simulation Results

A whole OpenGL-compliant 3D graphics rasterizer, including the proposed pixel coverage mask generation hardware algorithm (256 16-bit coverage masks), was modeled at RT-level in SystemC language [47]. Referring to the internal organization, the rasterizer adopts a tile-based rasterization approach. The tile size chosen for this particular implementation was set at 32×16 pixels, which implies that all the internal buffers (color buffer, depth buffer, and stencil buffer) composing the tile frame buffer have this size. The display size resolution was set at 320×240 pixels (QVGA), meaning that the display can be conceptually divided into 10×15 tiles. The rasterizer has only one pixel processing pipeline. The screen coordinates (X, Y) are represented on 9.4 bits (9 integer, 4 fractional), the color components (R,G,B,A) on 0.8 bits, the depth component (Z) on 0.24 bits, and the stencil component on 8.0 bits.

The ”aapoly” OpenGL application from [93] was executed on the virtual graphics hardware rasterizer. The generated image is presented in Figure 5.24. The antialiasing image quality can be seen in the detailed regions featuring pixel center and primitive geometry overlaid markings. The results of the hardware synthesis using Synopsys tools in a commercial 0.18μm IC manufacturing technology of the coverage mask generation circuit for one edge vector are presented in Table 5.8. Results are also provided for the edge vector class disambiguation circuit with bisectors. It is difficult to quantify the efficiency of the proposed implementation with respect to past solutions that produced coverage masks using normalized edge functions given that they do not provide details about their hardware implementation.

In an attempt to provide a fair comparison, we implemented the solution described in [78], and the results were an implementation with 8432 standard cells, an area of $270375\mu\text{m}^2$, and a latency of 4.2ns. This result indicates that our implementation is much more efficient; in addition, we managed to use

the saved area to implement the rest of the antialiasing hardware datapath [35] specified by Equation (5.5).

5.8 Conclusion

In this chapter we have proposed an efficient, high image quality run-time pixel coverage mask generation algorithm for embedded 3-D graphics antialiasing purposes, that is compatible with the above triangle traversal algorithm. The algorithm was implemented assuming 4×4 subpixel coverage masks and two's complement number representation. However, it has a higher degree of generality: it can be incorporated in any antialiasing scheme with pre-filtering that is based on algebraic representation of primitive's edges (for an illustration we have implemented the normalised distance proposed in [78]), it is independent of the underlying number representation, and it can be adapted to other coverage mask subpixel resolutions with the only prerequisite for the masks to be square. In addition, the proposed hardware algorithm represents a natural extension of the algorithm described in Chapter 3, for which a very efficient implemented will be presented in Chapter 6.

We have shown that precomputing the coverage masks for generator edges spread non-uniformly in the angular space of quadrant one, we have reduced the maximum error in coverage from 15.25% (assumed by previous state of the art implementations of similar antialiasing schemes) to 8.34% thus doubling the image quality, while reducing the implementation area significantly by an order of magnitude.

Chapter 6

An Efficient Tile-Based Traversal Algorithm Hardware Implementation

Tiling or chunking architectures [4] were proposed as a way to save memory bandwidth on framebuffer accesses (since an external memory access is typically one of the most energy-consuming operations) and to counteract the huge increase in storage of full-scene antialiasing. In a tiling architecture, the screen is divided in a number of non-overlapping regions, or tiles, which are processed serially. Every frame, primitive geometry is sorted first by screen location and dumped into one or more bins, one bin per tile. Geometry that overlaps a tile boundary is referenced in each tile that it is visible in. When all the primitive geometry has been specified, it is rendered from bin N to the tile N, before moving to the tile $N+1$. The advantage of the tile-based architectures is that all the data (colors, depth) can be maintained in on-chip tile-sized buffers, and accesses to external memories are required only to dump the tile color buffer content to the global off-chip frame buffer, when all the primitive geometry for the currently processed tile at the present frame was rasterized.

Although many algorithms [92, 66], based on edge functions [74], were proposed to rasterize efficiently primitives on traditional full-screen architectures, to the best of our knowledge, none was proposed for efficient rasterization in a tile-based architecture. All of the proposed algorithms are based on the following conceptual algorithm: while not all the positions inside the primitive are exhausted do 1) save the rasterization context, 2) move to a new rasteri-

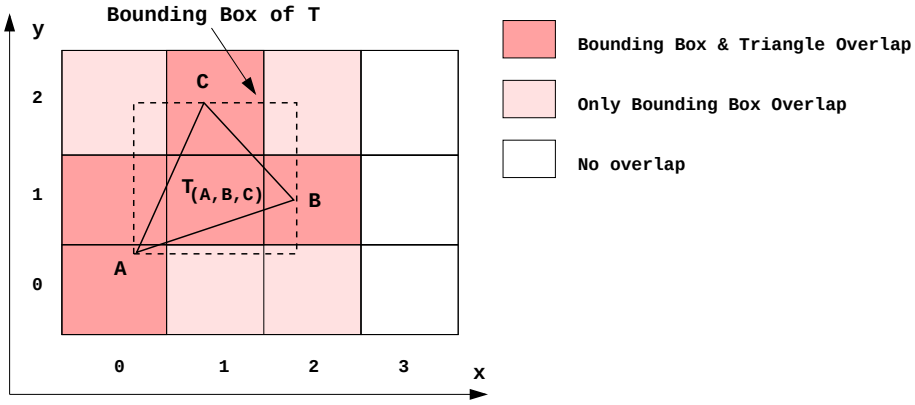

Figure 6.1: "Ghost" triangle for tiles $(0, 2)$, $(1, 0)$, $(2, 0)$, and $(2, 2)$.

zation position, 3) test the edge functions value for that position to see if the position is a hit, 4) if it is inside, communicate this hit position to the pixel processing pipelines and update the rasterization context or else restore the rasterization context, 5) based on the edge functions computed earlier, try to predict a new hit position. Computationwise, the main difficulty in tile-based rasterization with this algorithm is to find the first hit position in the to be rasterized primitive. To establish the overhead resulting from finding the first hit position, we performed experiments with heuristics. We included testing that determines if any of the primitive vertices or the primitive gravity center can be considered the starting rasterization position or the hit point. Our experiments indicated that the overhead can be between 50%-300% of the primitive rasterization time. In addition, there is always overhead associated with "ghost" primitives (depicted in Figure 6.1), primitives that are assigned to the current tile when they have nothing in common with it (this is due to the simplest algorithm in the software driver that assigns primitives to tiles based on a primitive bounding box test; other more complex tests in the software driver were envisaged eliminating the "ghost" primitive problem completely, but moving the costs to software). In full-screen rasterization, this overhead is inexistent due to the fact that a starting point inside the primitive can always be found, e.g., the gravity center. Apart of the overhead associated to locating the hit position, the traditional full-screen rasterization adapted for tile-based rasterization also exhibits random primitive pixel rasterization order. As several studies [50, 26, 56] indicate, the primitive pixel rasterization order is crucial for low-cost tile-based architectures that do not have dedicated texture memories (pull texture architectures) and are relying on a robust texture cache hit

ratio to reduce the latency and energy consumption of texel fetches from the external system memory. Also, a certain primitive pixel rasterization order may allow the interleaving of memory banks in the on-chip tile frame buffers. If this interleaving can be achieved then the dependencies introduced by the "read-modify-write" operation associated with the depth test and color blending can be removed. As a result the throughput of the system can be increased.

Therefore, in this chapter an efficient tile-based traversal algorithm hardware implementation to accelerate primitive traversal in 3-D graphics tile-based rasterizers is presented. The hardware implementation consists of two components: a systolic primitive scan-conversion subsystem [28] and a logic-enhanced memory [31]. The systolic primitive scan-conversion subsystem, using edge functions, works on a sliding window of 8×8 locations and outputs every clock cycle the primitive shape (encoded with one bit per location: ones represent tile pixels covered by primitive, zeros represent pixels not covered) for a different 4×4 pixel region inside the currently processed block. The window is moved to cover all the locations in the tile. The logic-enhanced memory works back-to-back with the systolic subsystem, contains the same number of bits as the number of pixels in the tile, and during rasterization time it is to be filled up in several clock cycles by the systolic primitive scan-conversion subsystem with the stencil of the primitive. Once the shape of the primitive has been coded inside the memory, the memory internal logic is capable of delivering on request in one clock cycle at least one and up to four hit positions to the pixel processing pipelines, signaling when all the hit positions are consumed.

The main contributions of the proposed tile-based traversal algorithm hardware implementation can be summarized by the following:

- the first hit position inside the primitives is found with no overhead,
- "ghost" primitives are efficiently handled, because they are discarded after a small constant delay, irrespective of the primitive size. This contrasts with the exhaustive search of the tile boundary required by tile-based rasterizers that adapt the full-screen rasterization approach.

Additionally, our proposal imposes a rasterization order with the following benefits:

- hit positions are communicated in a spatial pattern that has the potential to increase the hit ratio of texture caches in pull texture architectures;

- hit positions can always be mapped to different memory banks in the Z-buffer or color-buffer breaking the "read-modify-write" dependency associated with depth test and color blending.

Although outside of the scope of present thesis, another benefit of regularity of systolic architectures is its suitability to fault-tolerant design, and therefore increasing chip fabrication yields. A large body of literature on systolic arrays presents schemes of error detection and correction. The main avenues to achieve this are hardware and time redundancy. The former approach detects and corrects errors by introducing additional computing hardware (different circuits, where the numbers are encoded differently, or duplicating existing cells — therefore increasing hardware costs) [75], while the latter duplicates computations using the same hardware and votes on the results (and reduces the effective throughput if performed at run-time) [94][23]. Most of the schemes are hybrid and borrow the benefits of both approaches [81][95][36][20]. An alternative to run-time error detection or correction is the offline mode where the chip is put in diagnostic mode in the silicon fab, and upon finding an error a new spare cell is reconfigured in place of the faulty one [71][23].

The rest of the chapter is organized as follows. The pixel rasterization order is introduced in Section 6.1. The systolic primitive scan-conversion subsystem is presented in Section 6.2. The logic-enhanced memory architecture is described in Section 6.3. Hardware implementation results are presented in Section 6.4.

6.1 Efficient Pixel Rasterization Order

For clarity of explanation and without loss of generality, we assume a standard QVGA display size (with a resolution of 320×240) used in mobile terminals, divided in tiles with a size of 32×16 pixels. The screen coordinates x and y of the primitive vertices for a QVGA display are represented as unsigned fixed-point numbers in the format 9.4 (meaning 9 integer bits and 4 fractional bits). We assume that the arithmetic computations are performed in two's complement notation.

The quest to an efficient hardware algorithm for rasterization has to start from finding a suitable pixel rasterization order. In Figure 6.2, the pixel grid of the tile around the origin of the tile coordinate system is depicted and a proposed space-filling path indicated with arrows starting from the origin is presented. Space-filling paths are known to improve the texel coherency generating high

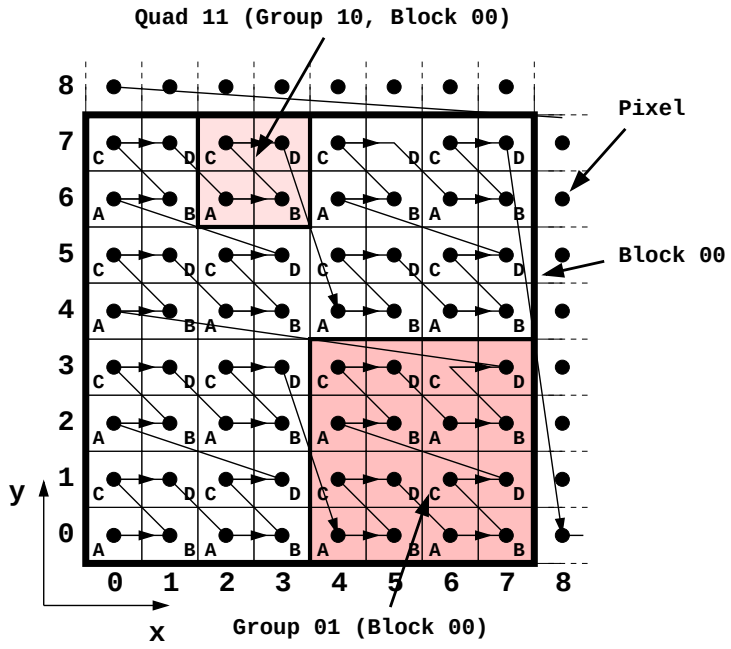

Figure 6.2: Proposed pixel rasterization order in tile.

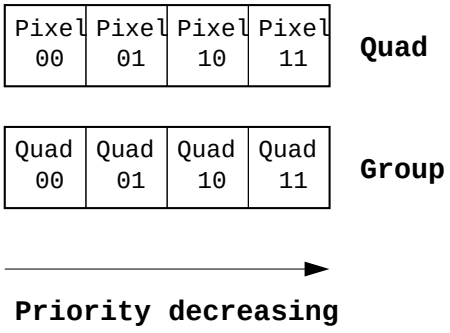

Figure 6.3: Pixel and Quad coding.

hit-ratio in texture caches [4]. In addition, if 2×2 regions of fragments can be generated during rasterization they can be mapped on different memory banks A, B, C, and D. Supposing that the shape or stencil of a triangle has been already coded in a memory representing the bi-dimensional tile, now hit locations have to be forwarded to the pixel processing pipeline. The only way to select among many hit locations according to the space-filling path traversal order is via priority encoding. After the hit location is communicated, the bit for

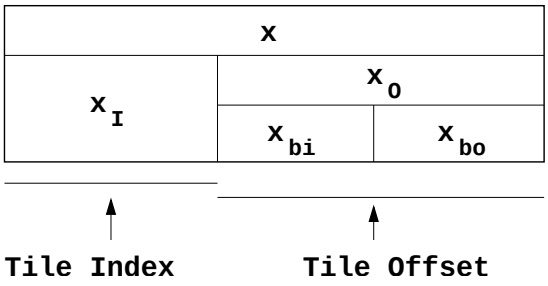

Figure 6.4: Fields of the x screen coordinate.

that location has to be reset in order for a priority encoding scheme to work further. Referring to Figure 6.2, an (x, y) offset position can be encoded in terms of block positions (8×8 fragment regions), group positions (4×4 fragment regions), quad positions (2×2 fragment regions), and positions in quad. Assuming a 32×16 pixel tile, the location $(x, y) = (x_4x_3x_2x_1x_0, y_3y_2y_1y_0)$ can be encoded as $(Block, Group, Quad, Pos) = (y_3x_4x_3, y_2x_2, y_1x_1, y_0x_0)$. With this encoding, priority can be restated hierarchically: hit locations in a block (respectively group, quad) encountered earlier on the space-filling path have a higher priority than any hit locations in a block (respectively group, quad) encountered later on the path (see Figure 6.3). The way the hierarchical priority encoding scheme is implemented in hardware is presented in the next sections.

6.2 Systolic Computation of the Primitive Stencil

The systolic primitive scan-conversion subsystem, using edge functions, works on a sliding window of 8×8 locations (a block) and outputs every clock cycle the primitive shape (encoded with one bit per location: ones represent tile pixels covered by primitive, zeros represent pixels not covered) for a different 4×4 pixel region (a group) inside the currently processed block. The sliding window is moved according to the space-filling path traversal order presented in Figure 6.2 until all the tile locations are exhausted, and also the groups generated by the systolic subsystem on the block-sized window are output according to the space-filling path traversal order. The systolic subsystem sliding window size is designed to lead to hardware costs that match the hardware size of a full-screen rasterizer, therefore larger sizes, although beneficial from a performance viewpoint, are considered too costly. For the current tile size of 32×16 pixels, the computations for an entire tile will take 32 clock cycles.

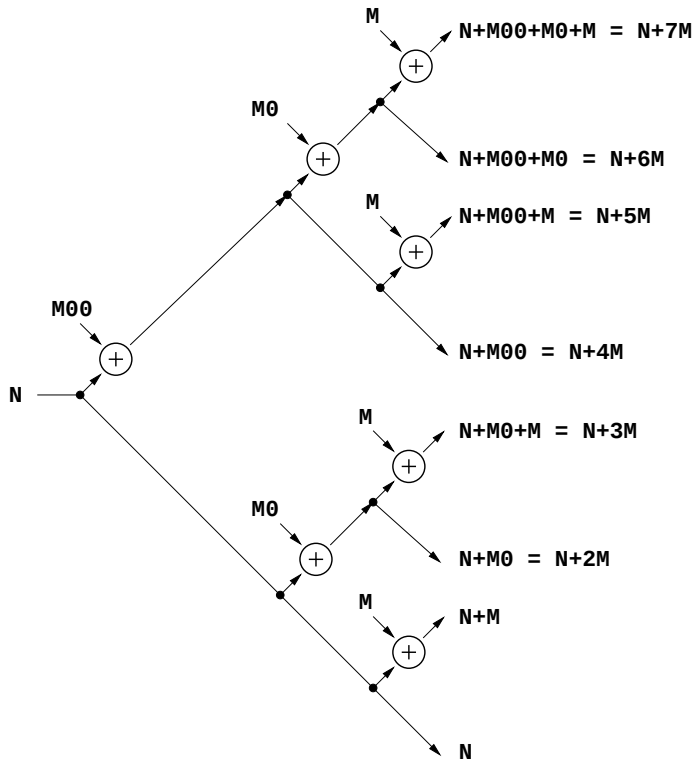

Figure 6.5: Parallel computation graph of $x_{bo} \cdot M + N$ for every $x_{bo} \in [0, 7]$.

A primitive is rasterized using edge functions. In a tile-based rasterizer, for an edge vector $\overrightarrow{AB}$ the edge function can be reformulated as follows:

$$\begin{aligned}
E_{AB}(x,y) &= (x - x_A) \cdot \Delta y_{AB} - (y - y_A) \cdot \Delta x_{AB} \\
&= (x_I + x_O - x_A) \cdot \Delta y_{AB} - (y_I + y_O - y_A) \cdot \Delta x_{AB} \\
&= x_O \cdot \Delta y_{AB} - y_O \cdot \Delta x_{AB} + (x_I - x_A) \cdot \Delta y_{AB} \\
&\quad -(y_I - y_A) \cdot \Delta x_{AB} \\
&= x_O \cdot \Delta y_{AB} - y_O \cdot \Delta x_{AB} + E_{AB}(x_I, y_I) \\
&= (x_{bi} \cdot 8 + x_{bo}) \cdot \Delta y_{AB} - (y_{bi} \cdot 8 + y_{bo}) \cdot \Delta x_{AB} \\
&\quad +E_{AB}(x_I, y_I) \\
&= x_{bo} \cdot \Delta y_{AB} - y_{bo} \cdot \Delta x_{AB} + (x_{bi} \cdot \Delta y_{AB} \\
&\quad -y_{bi} \cdot \Delta x_{AB}) \cdot 8 + E_{AB}(x_I, y_I) \\
&= x_{bo} \cdot M + y_{bo} \cdot P + N + Q \\
&= (x_{bo} \cdot M + N) + (y_{bo} \cdot P + Q)
\end{aligned} \tag{6.1}$$

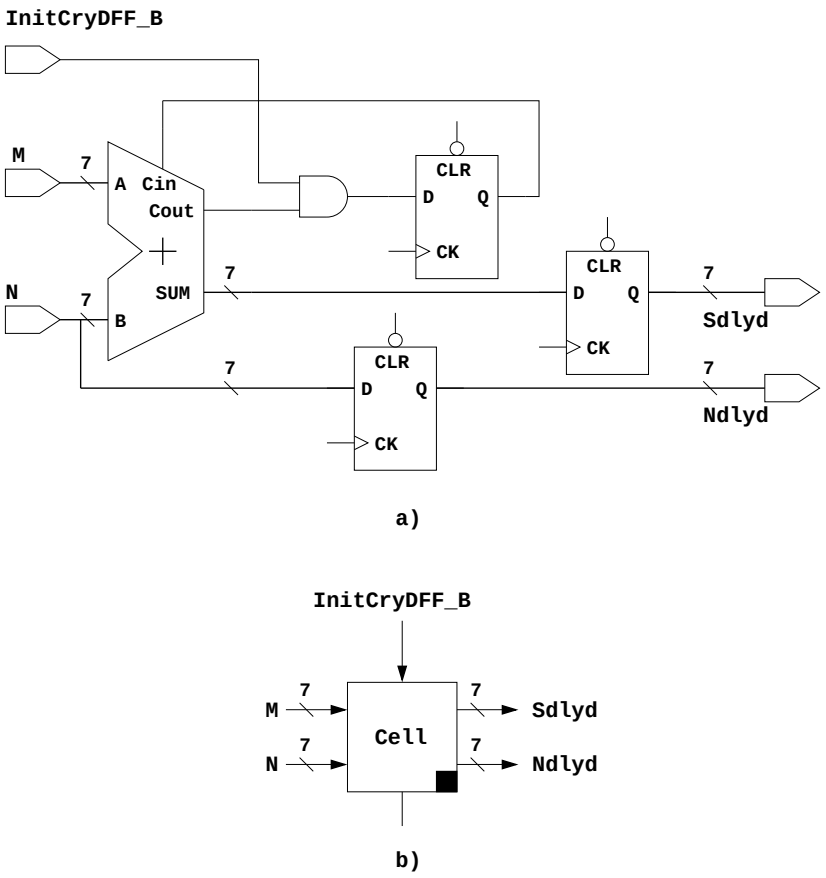

Figure 6.6: *Cell* processing element circuit diagram.

where (x_I, y_I) represent tile coordinates on the screen, (x_O, y_O) represent offset coordinates in a tile, (x_{bi}, y_{bi}) are the block coordinates in the tile, and (x_{bo}, y_{bo}), $x_{bo} \in [0, 7]$, $y_{bo} \in [0, 7]$ represent pixel offsets in the block (see Figure 6.4). The values Δx_{AB}, Δy_{AB} and the quantity $E_{AB}(x_I, y_I)$ are computed at primitive setup time. The quantity $(x_{bi} \cdot \Delta y_{AB} - y_{bi} \cdot \Delta x_{AB}) \cdot 8$ is computed before any computations are started on a new block window and the computation can be performed efficiently as multi-operand addition (carry-save addition followed by carry-propagate addition). Therefore, the last two quantities can be regarded as constants N and Q, and what we propose is to compute in parallel the expression $(x_{bo} \cdot M + N) + (y_{bo} \cdot P + Q)$ for every $x_{bo} \in [0, 7]$, $y_{bo} \in [0, 7]$ (when antialiasing is considered the required normalized edge functions, as described in Chapter 5 Equation 5.5, can be obtained

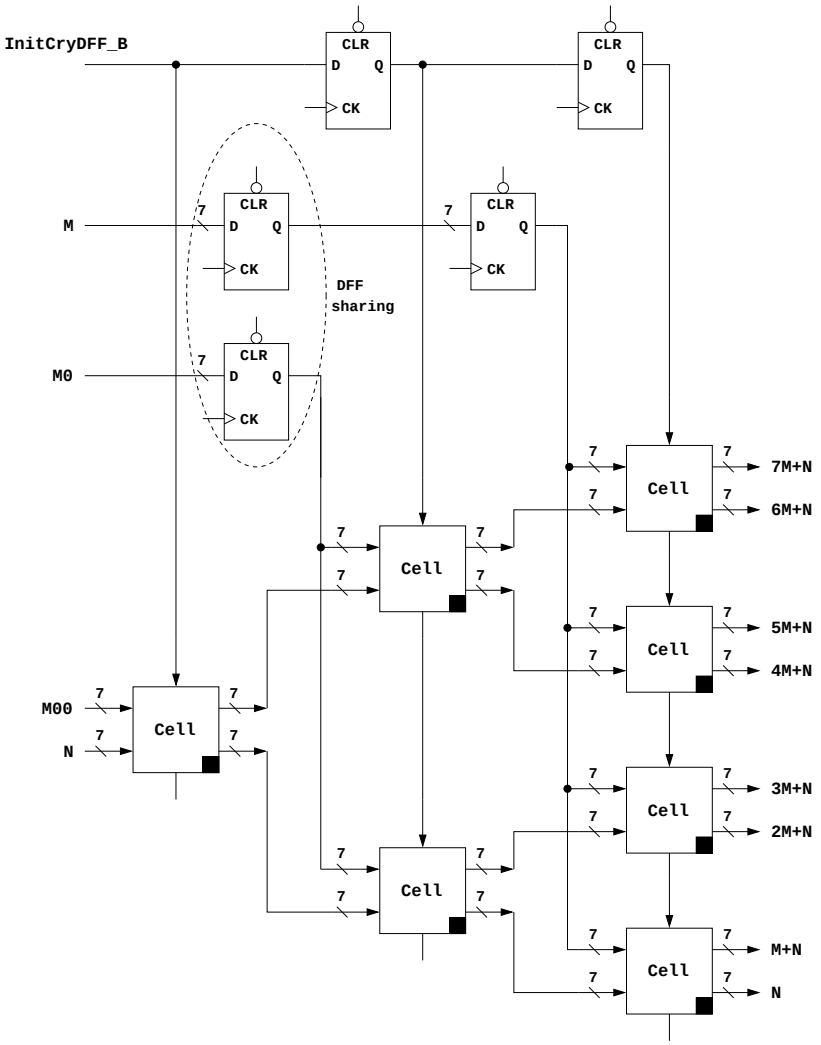

Figure 6.7: Systolic computation of $x_{bo} \cdot M + N$ where $x_{bo} \in [0, 7]$.

with a correction of the constant Q with the term $1/2 \cdot (|\Delta x_{AB}| + |\Delta y_{AB}|)$.

The first solution is to compute the expression $x_{bo} \cdot M + N$ and $y_{bo} \cdot P + Q$ for every $x_{bo} \in [0, 7]$, $y_{bo} \in [0, 7]$ using a direct hardware mapping of the graph depicted in Figure 6.5. In the tree, $M0$ and $M00$ are denoting left-shifted value of M with one position and two positions, they will be derived from M value by some multiplexers outside the tree circuit. The costs for the $E_{AB}(x, y)$ computation in the current block will be prohibitive in both area and latency

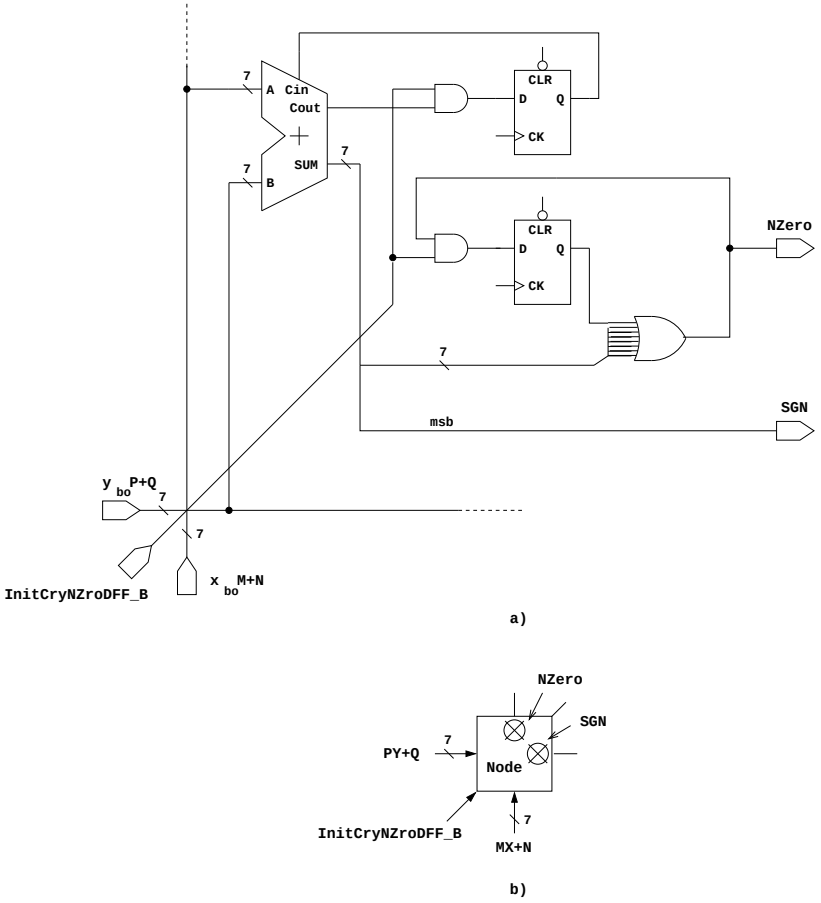

Figure 6.8: *Node* processing element circuit diagram.

for this method: it will require 78 28-bit adders (for all three edge function computation, this requires 234 28-bit adders) and the critical path will span 4 28-bit adders.

We are proposing a second solution that is more economical in cost and has a very low latency. This is the systolic subsystem described in the following. First, the expresions $x_{bo} \cdot M + N$ and $y_{bo} \cdot P + Q$ can be computed in parallel for every $x_{bo} \in [0, 7]$, $y_{bo} \in [0, 7]$ using a tree of *Cell* processing elements. The *Cell* processing element is depicted in Figure 6.6 and contains a 7-bit ripple-carry adder, and three D flip-flops: one in which the carries are stored between additions, one to store the result of the current addition, and one to delay with one clock cycle one of the operands. The systolic tree computing $x_{bo} \cdot M + N$ is

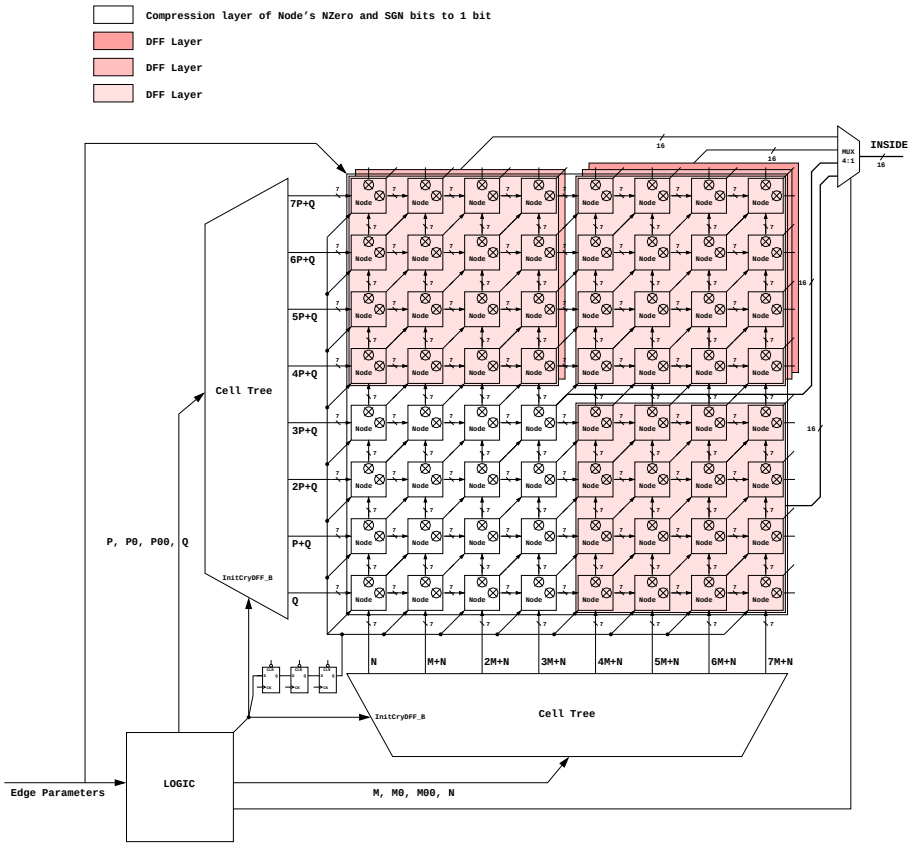

Figure 6.9: Systolic computation of the edge function for an 8×8 pixel window.

presented in Figure 6.7. Every clock cycle 7 bits of the 28-bit result are output by the systolic tree starting with the least significant 7 bits. In the tree, left-shifted values of M with one position and two positions are required and those can be provided from M value by some multiplexers outside the tree circuit.

As 7-bit slices of the values $x_{bo} \cdot M + N$ and $y_{bo} \cdot P + Q$, $x_{bo} \in [0, 7]$, $y_{bo} \in [0, 7]$ are generated every clock cycle, they are combined by *Node* processing elements arranged in a 8×8 matrix. The *Node* processing element is depicted in Figure 6.8 (for drawing purposes, the outputs in the symbol are drawn with crosses meaning that they are perpendicular on the page). It takes two partial 7-bit results and combines them with a 7-bit ripple-carry adder outputing only delayed versions of the edge function sign bit and the edge function not zero flag (to compute the primitive stencil, the values of the edge functions are not

interesting per se but their relationship with 0). In addition, another D flip-flop is again required to store the generated carry.

The entire systolic subsystem for an edge function is presented in Figure 6.9. With additional information (edge quadrant, primitive edge orientation convention, antialiasing enabled or not), the two outputs of every *Node* processing element can be compressed in only one signal (the compression layer depicted in Figure 6.9). The 8×8 matrix of *Node* elements generates results for a different sliding window of 8×8 locations every 4 clock cycles. With the addition of the D flip-flop layers depicted in Figure 6.9, results for a different group (4×4 locations) in the window are generated every clock cycle and with proper multiplexing they are available on the `Inside` output port. Assuming an identical systolic subsystem for each primitive edge, the `Inside` signals for each edge are combined by an AND operation and each bit in the 16-bit result indicates whether that particular location in the group is covered by the primitive (one for inside the primitive, zero for outside). Every clock cycle, the groups generated are written in the logic-enhanced memory described in the next section.

6.3 Logic-enhanced Memory Architecture

In the following, a logic-enhanced memory architecture based on a hierarchical priority encoding scheme supporting a tile size of 32×16 pixels is presented. The memory is used in conjunction with the systolic primitive scan-conversion subsystem described in the previous section. The logic-enhanced memory has a word line width equal to the size of a group (16 bits), and is capable of working back-to-back with the systolic subsystem meaning that every clock cycle the primitive shape for a group of locations is transfered inside the memory. The memory contains 32 wordlines (1tile$=8$blocks$\times 4$groups) and will be filled up by the systolic subsystem with the primitive shape in 32 clock cycles. Once the shape of the primitive has been completely transferred to the logic-enhanced memory, quads which contain at least a hit location (less than four hit locations, if the quad is situated on the primitive edges) will be output at a rate of one per cycle to the pixel processing pipeline in the proper space-filling rasterization order, until all the hit locations are exhausted. The memory output interface includes the individual bits of the quad, the encoding of the quad in the $(Block, Group, Quad)$ format, and a signal that indicates whether all the hit locations were transferred out. The $(Block, Group, Quad)$ format is suitable for computations in the subsequent pixel processing stages

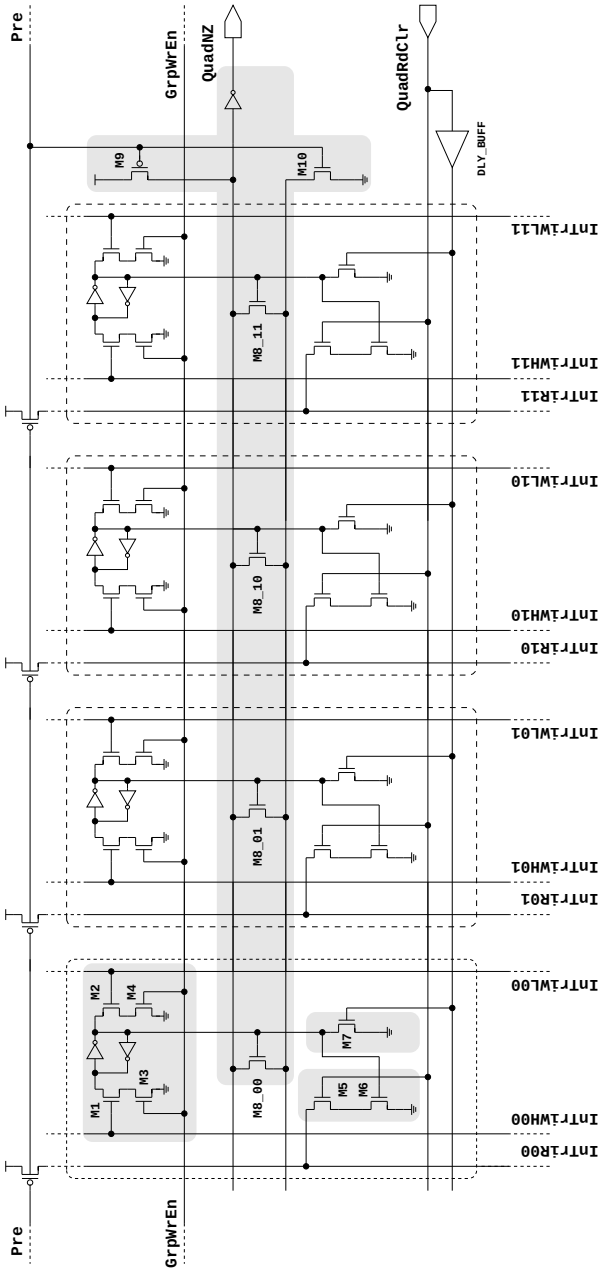

Figure 6.10: Quad cell.

resulting only in multi-operand additions that can be implemented efficiently, as described in [27].

The logic-enhanced memory architecture is presented in Figures 6.10, 6.11, and 6.13. The memory contains 32 wordlines, each wordline contains a group, each group contains four quads, and each quad contains four locations bits. An entire group can be written per clock cycle, but only one quad can be read out per clock cycle. For write operations the memory behavior is identical to any CMOS SRAM read/write memory and will not be described in detail; further, the differential bit lines used for writing are omitted from the drawings. When a quad containing hit locations is requested, the priority tokens of all quad words are transmitted to the group priority encoder, and all group priority tokens are transmitted to the global priority encoder. After a decision is taken by the global priority encoder on the highest priority quad containing hit locations, information is bounced back only to the group word line containing the quad, the quad being read out and its non-zero bits reset to prepare the memory for the next read. The location information is stored using static RAM bit cells, but the logic circuitry is implemented in domino dynamic logic clocked by the precharge signal `Pre`. The priority encoders are similar to the high-speed low-power *n*-type domino logic design described in [55]: the 4-bit group priority encoder has one-level lookahead (see Figure 6.12) and the 32-bit global priority encoder is constructed from 8-bit priority encoders connected through the third-level lookahead signals.

In Figure 6.10, the Quad Cell circuit diagram is presented where four locations bit cells are depicted. Each bit cell consists of a storage cell (transistors M1, M2, M3, M4 and the two cross-coupled inverters), one of the four parallel transistors (M8) of a distributed domino four-input OR gate (that includes additionally transistors M9 and M10), the conditional read circuitry (transistors M5, M6), and the reset transistor M7. For write operation, when the signal `GrpWrEn` is asserted, one of the two storage nodes is pulled down, and the other is pulled up. This requires that the pullup in both inverters to be weaker than the series pulldown transistors. The storage cell is write-only because the conditional read signal is formed internally based on the content of the storage cell and the signal `QuadRdClr` formed outside the Quad Cell (see Figure 6.11). The role of the OR gate is to detect if hit locations are stored in the quad cell; the signal `QuadNZ` will participate in priority encoding in the Group Cell. If based on the priority encoding scheme the quad contains the hit locations with the highest priority in the memory, the signal `QuadRdClr` will be asserted for the read and clear operation on the quad bits. The static delay buffer `DLY_BUFF` insures that there is enough separation in time between

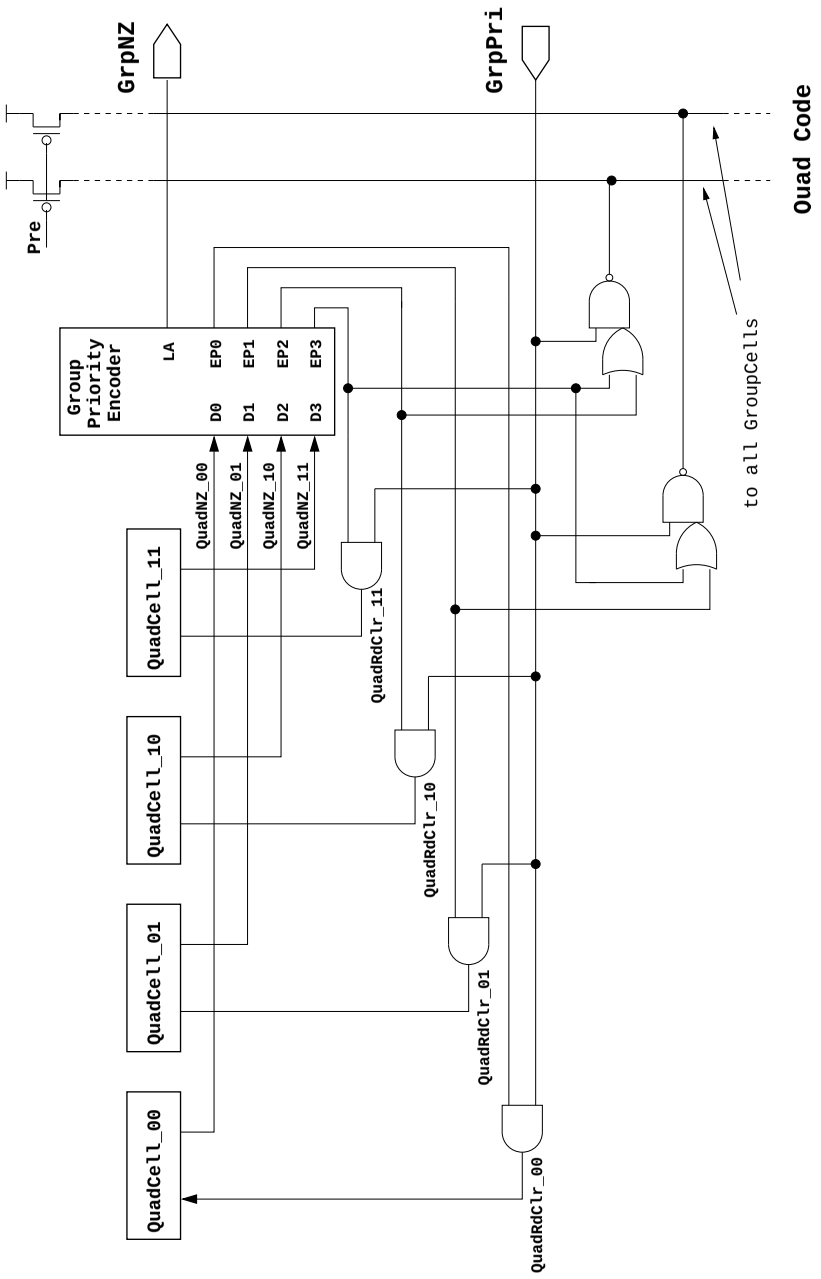

Figure 6.11: Group cell.

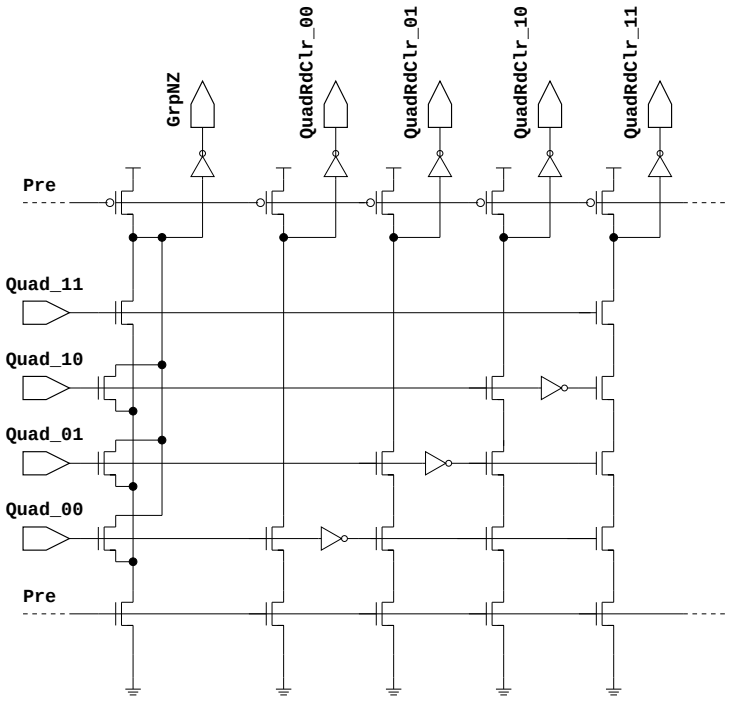

Figure 6.12: Dynamic priority encoder with one-level of lookahead.

reading the quad bits (the precharged read bit line `InTriR` can be discharged enough to be detected as logic 0 by the charge-redistribution amplifiers) and clearing the quad bits. The size required for `DLY_BUFF` is small because the memory has only 32 word lines.

The role of the Group Cell additional logic circuitry presented in Figure 6.11 is to pass forward the `QuadNZ` signals from the four Quad Cells to the group priority encoder. `GrpNZ` is connected to the lookahead ouput port of the group priority encoder and signals that at least one of the Quad cells contains hit locations and this is input in the global priority encoder. If the global priority encoder decides that the Group Cell has the highest priority among the other Group Cells, the signal `GrpPri` will be asserted. `GrpPri` together with the priority encoded lines `EP` from the group priority encoder will be anded using four two-input domino AND gates forming `QuadRdClr` signals for the Quad Cells. In addition, two domino OAI gates are used to form the quad code — if a quad having the highest priority exists in the Group Cell, then two precharged bit lines are discharged broadcasting the quad code to the memory output.

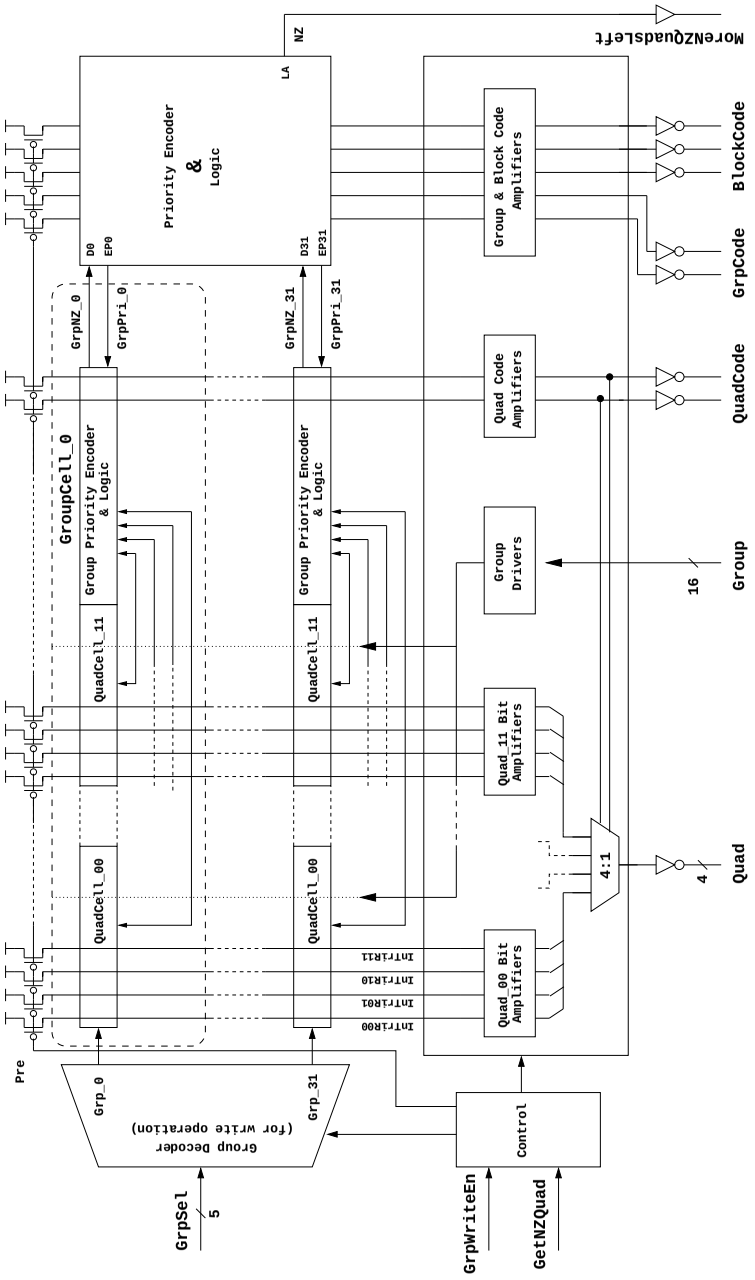

Figure 6.13: Logic-enhanced memory architecture.

Finally, in Figure 6.13, the block diagram of the logic-enhanced memory is presented. The signals from the Group Cells are input in the global priority encoder that decides which one of the Group Cells has the highest priority. `MoreNZQuadsLeft` is connected to the lookahead ouput port(`LA`) of the global priority encoder and signals that at least one of the Quad cells contains hit locations, therefore indicating outside the memory if there are any hit locations left. The quad code returned is used for multiplexing the highest priority quad bits from the highest priority Group Cell, and logic similar to that presented in Figure 6.11 is used to generate the block code and group code outputs. The memory input interface contains the `GetNZQuad` signal that has to be asserted in order for a quad with hit locations to be read out. The rest of the circuitry is identical to any CMOS SRAM read/write memory and will not be described.

6.4 Hardware Implementation Results

The systolic subsystem sliding window size was designed to lead to hardware costs that match the hardware size of a functionally equivalent full-screen scan-conversion unit. Larger sizes, although they may provide benefits from a performance viewpoint, were considered too costly for mobile terminals and were not implemented. We performed the hardware synthesis using Synopsys tools in a commercial $0.18\mu\mathrm{m}$ IC manufacturing technology. The results for the systolic primitive scan-conversion subsystem for all three edge functions including the required control are presented in Table 6.1. The critical path of the unit can be clocked at a frequency of at least 200 MHz when reasonable clock uncertainty is taken into account. The latency is one primitive stencil computed every 32 clock cycles, but the stencil computation is completely hidden by the logic-enhanced memory operation that feeds the pixel processing pipelines. The power consumption was estimated assuming random vectors on the inputs and is presented also in Table 6.1. It should be noted that in reality the actual figure of power consumption may be somewhat lower due to existing signal correlations that are not accounted for in our estimation. The system we are describing is already modeled in SystemC as part of GRAAL [32], a full-fledged 3-D graphics OpenGL-compliant tile-based hardware rasterizer. The performance figures presented in Table 6.1 are computed for typical triangles with an average area of 160 pixels [7] and indicate the performance of the triangle setup stage and the maximum theoretical pixel fill rate (it does not account for texture cache miss penalty inducing stalls upstream) that can be

IC Technology	UMC *Logic18-1.8V/3.3V-1P6M*
Std. Cell Library	VST *e*Si-Route/11
Critical Path [ns]	2.155
Area [μm^2]	269964
Std. Cell Number	
D Flip-Flops (DFF area = 81.3μm^2)	1413
Full Adders (FA area = 65μm^2)	1638
Control Circuitry Gates	7071
Total	10122
Power Consumption [mW]	33
Energy Consumption [mW/MHz]	0.165
Performance	
Rendering Rate [triangles/s]	2.44×10^6
Fill Rate [pixels/s]	460×10^6

Table 6.1: Systolic scan-conversion hardware implementation results.

IC Technology	UMC *Logic18-1.8V/3.3V-1P6M*
Critical Path Latency [ns]	2.387
Area[μm^2]	
Bit Cell	144
Quad Cell	636
Group Cell	2894
Total(including peripheral circuitry)	118985

Table 6.2: Logic-enhanced memory hardware implementation results.

achieved with the proposed systolic subsystem.

It would have been of interest to compare our scheme with other designs. Unfortunately, implementation details that regard what we have developed (the primitive scan-conversion hardware algorithm) are not available from the existing literature (see for example [2]).

The logic-enhanced memory was designed at the physical level in the same 0.18μm IC manufacturing technology. After the parasitics were extracted from layout the annotated circuits composing the critical path (starting in Quad Cell 11 of the Group Cell 31, going through the global priority encoder then back to the originating cell, then to the Quad_11 Bit Amplifiers, and finally reaching the `Quad` output port) were simulated using the HSPICE circuit simulator.

The results are reported in Table 6.2. The critical path latency translates in a maximum clock frequency of 200 MHz assuming that the precharge and the evaluation phases take half of the clock cycle.

6.5 Conclusion

In this chapter we have described an efficient tile-based traversal algorithm hardware implementation to accelerate primitive traversal in 3-D graphics tile-based rasterizers. The hardware implementation consists of two components: a systolic primitive scan-conversion subsystem, using edge functions, and a logic-enhanced memory, which is filled in several clock cycles with the shape of a new triangle by the systolic subsystem. The systolic primitive scan-conversion subsystem has a throughput of 16 pixels per clock cycle. The memory internal logic is then capable of delivering up to 4 pixels per clock cycle to the pixel processing pipelines, in a spatial pattern which is very advantageous for texture caching and for reducing bank clashes in multi-banked SRAM tile buffers, used for read-modify-write operations associated with depth test and colour blending. The ghost primitives generated by trivial triangle tile binning implementations in tile-based rasterization systems are also discarded very fast in 4 clock cycles, reducing significantly the impact on the triangle throughput in such systems.

Special considerations were given 1) not to compromise the operational noise margins of the circuitry and 2) the enhancing logic(arithmetic) cells to have a layout with a similar pitch to the data storage cells in order to facilitate high cell integration densities. Therefore, in the logic-enhanced memory, the storage cells were implemented with traditional SRAM circuitry (two cross-coupled inverters generating the storing latch and two NMOS pass transistors for access), but the logic cells were implemented in a domino dynamic logic style.

The hardware implementation has shown that such a design could be clocked at a frequency of at least 200Mhz with reasonable cost and power consumption figures.

Chapter 7

Primitive List Hardware Acceleration for Embedded 3D Graphics

Tiling architectures have been proposed for graphics as a way to save memory bandwidth on frame buffer accesses, and to counteract the huge increase in storage of full-scene antialiasing (only on-chip tile-sized frame buffers are maintained). In a tiling architecture, the screen is divided in a number of non-overlapping regions, or tiles, and for every frame, primitive geometry is sorted first by screen location and dumped into one or more bins, one bin per tile. Each bin is then rasterized sequentially.

Tile-based rendering appears to be promising from a low-power perspective, because it decomposes a scene into smaller tiles. By rendering the tiles one-by-one, a small memory integrated on the graphics accelerator can be used to store the color components and depth (z) values of one tile. This implies that most accesses are local, on-chip accesses, which consume significantly less power than accesses to the off-chip frame and z buffers.

Figure 7.1 presents GRAAL, our proposed hardware rasterizer, in the system-on-chip context. Adopting a bus-centric view, the system contains bus masters such as the host processor, GRAAL rasterizer's memory transactors, the scan converter memory engine, and bus slaves (such as the external memory interface, GRAAL rasterizer's register blocks, and the scan converter register interface). In order to rasterize a 3-D graphics scene, from a functional and transactional (memory) perspective, the following sequence of events must be

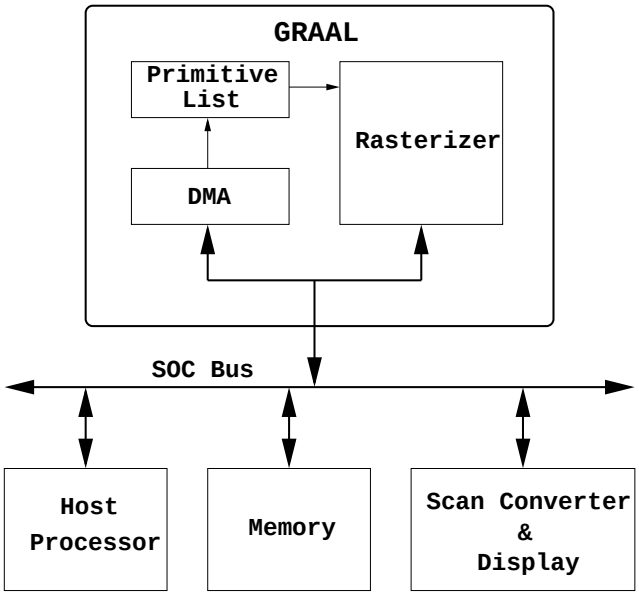

Figure 7.1: GRAAL integrated in a system on chip.

triggered in the afore-mentioned system.

1. the triangle bins are written to the system memory by the host processor, usually in linked-list memory blocks;

2. the host processor issues to the GRAAL register interface a series of commands, memory base addresses, and configuration settings, it enables interrupts from GRAAL, and finally, writes to a special GRAAL kicker register which signals the start of render;

3. on receiving the kick on its slave interface, GRAAL, via its primitive list DMA and programmed base addresses, fetches (without any host processor involvment) the tiles with triangles from the system memory written in the linked-list memory blocks;

4. for each tile, the triangles are rasterized and written to the framebuffer location in the system memory;

5. when the last tile for the current render is finished, GRAAL will raise an interrupt;

6. the host processor will be interrupted, and, as a result, it will communicate to the scan converter the address of the framebuffer location in

system memory, i.e., it will program the address in the scan converter registers, it will enable interrupts from the scan converter, then the host processor will kick the scan converter and exit the interrupt routine;

7. the scan converter will use its own DMA to fetch the framebuffer content from the system memory and display it to the screen, then will interrupt again the host processor;

8. the host processor will be interrupted again, and, by examining some status registers, it will know that the scene has been rasterized to the screen, thus it can unlock and reuse the memory addresses used for the frame buffer.

As described above, there are two important data transfers performed in the system for the purpose of the rasterization, and both of them have the system memory as an intermediary: the first transfer is from the host processor to the hardware rasterizer, and the second transfer is from the rasterizer to the framebuffer location in memory. GRAAL, being an embedded hardware rasterizer, has to reduce to a minimum the memory bandwidth consumed for rasterization, because any embedded system has available only a limited memory bandwidth budget. This objective is achieved by using a tile-based rasterization approach employing several tiling algorithms we presented in [59], as demonstrated in Section 7.1. However, a memory bandwidth reduction in itself is not enough because, in an embedded system, the host processor has a general purpose role, and the computational requirements of the algorithms described in Section 7.1 are still too high. Therefore, the tiling list computation has to be uploaded to the hardware rasterizer as much as possible. This would be the second objective, achieved by using a hardware primitive list accelerator as described in Section 7.2. At the same time, the hardware primitive list accelerator achieves the first objective too.

7.1 The SW Tiling Algorithm

The tiling engine is responsible for sorting the primitives into bins and sending them to the rasterizer in tile-based order. As mentioned before, tiling appears to be promising for low-power implementations because it reduces the memory bandwidth required between the rasterizer and the (external) frame and z buffers. Since accesses to external memory often dissipate more energy than the datapaths and the control units, reducing them can provide significant energy savings.

In Subsection 7.1.1, we investigate how much external data traffic is saved by a tile-based renderer when compared to a traditional renderer. The sorting step, however, also requires memory bandwidth and in Subsection 7.1.2 we evaluate several algorithms for performing this step. Finally, Subsection 7.1.3 investigates how the state change information can be reduced.

7.1.1 Memory Bandwidth Requirements Of Tile-Based Rendering

In this section, we examine how the amount of external data traffic varies with the tile size to identify the tile size that yields the best trade-off between data traffic volume and area needed for on-chip buffers. Furthermore, we measure how much external data traffic is saved by a tile-based renderer. Previous studies have not presented such measurements [54] or have focused only on the overlap [24], defined as the average number of tiles covered by a primitive.

The GraalBench benchmark suite [7] has been used. Q3L and Q3H are traces of the popular Quake III game. Tux is a freely available game that runs on Linux. AW is the AWadvs-04 test that is part of the SPECViewperf $6.1.2$ package [84]. ANL, GRA, and DIN are VRML scenes which were chosen based on their diversity and complexity. All workloads use VGA resolution (640×480), except Q3L which uses QVGA resolution (320×240). Furthermore, the first 3 traces consist of around 1400 frames, while the latter 4 consist of about 600 frames.

Table 7.1 depicts the number of triangles transferred from the tiling engine to the rasterizer for various tile sizes. The last row shows the number of triangles transferred if the tile size is equal to the window size. The overlap can therefore be obtained by dividing the number of triangles transferred for a specific tile size by the number given in the last row.

Obviously, if the tile size increases, the number of transferred triangles decreases, since there is less overlap. In our design, however, it is important to use as little internal memory as possible and, consequently, a trade-off needs to be made. The results show that using tiles smaller than 32×32 increases the number of triangles transferred significantly. For example, if a tile size of 16×16 is employed instead, the amount of geometrical data sent to the rasterizer increases by $2.02\times$ for the Q3H benchmark, and by $1.97\times$ for the Tux benchmark. On average, using the geometric mean, a tile size of 16×16 increases the number of triangles sent by $1.62\times$, when compared to 32×32 tiles. Moreover, employing tiles larger than 32×32 reduces the amount of geomet-

Tile size	Benchmarks					
	Q3H	Tux	AW	ANL	GRA	DIN
16×16	21,300	8,204	15,627	18,731	9,416	9,416
16×32	15,600	6,101	14,464	13,850	8,215	7,905
16×64	13,009	5,143	13,911	11,555	7,624	7,142
32×16	14,662	5,539	14,187	14,823	7,183	7,954
32×32	10,526	4,148	13,090	10,689	6,217	6,591
32×64	8,671	3,576	12,567	8,745	5,742	5,904
64×16	11,360	4,225	13,480	12,910	6,071	7,216
64×32	8,006	3,245	12,416	9,150	5,223	5,928
64×64	6,518	2,813	11,908	7,308	4,807	5,278
640×480	3,404	1,822	10,768	4,321	3,603	4,083

Table 7.1: Number of triangles transferred as a function of the tile size.

rical data only marginally. For example, using 64×64 instead of 32×32 tiles reduces the data by $1.35\times$ (geometric mean). This indicates that for the considered workloads a tile size of 32×32 is the best trade-off between the number of triangles sent to the rasterizer and the internal buffer size. We remark that the resolution might affect the optimal tile size for non-scalable applications. In practice, however, most applications scale their content complexity based on the resolution and, therefore, the optimal tile size cannot be substantially affected by a resolution change. If each element is represented by 32 bits for RGBA color, 24 bits for depth, and 8 bits for stencil, 64Kbits are required to implement 32×32 tile buffers. In an SRAM implementation, this corresponds to about 96K equivalent gates. This number can be compared to current gate budgets available for mobile graphics accelerators, which are around 200K to 500K gates.

Figure 7.2 depicts the total amount of external data traffic produced by the conventional and the tile-based renderer for a tile size of 32×32. The total traffic has been divided into data sent from the tiling engine to the rasterizer and data transferred between the rasterizer and the external frame and z buffers and texture memory. Since the latter component is much larger than the former, the tile-based renderer reduces the total external traffic volume significantly, by $1.96\times$ on average (geometric mean). For some workloads, however, the advantage of tile-based rendering is marginal. For example, for the Aw benchmark, it is only 23.4%. By defining the overdraw as the number of pixels written to a buffer divided by the buffer size in pixels, this is because tile-based rendering

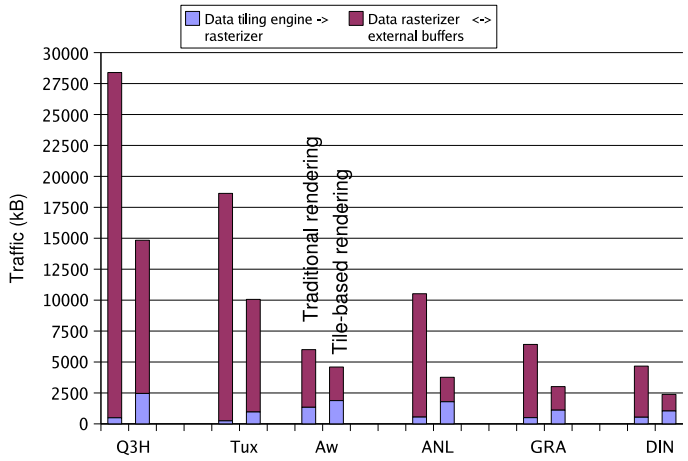

Figure 7.2: Total external data transferred (KB) per frame for a tile-based and a traditional architecture.

decreases the data transferred between the rasterizer and the external frame and z buffers and texture memory (depending on the overdraw), but increases the amount of data sent from the tiling engine to the rasterizer (depending on the overlap). Hence, tile-based renderers are more suitable than traditional renderers for workloads with low overlap and high overdraw, a trend foreseen for the future. For workloads with high overlap and low overdraw, on the other hand, tile-based renderers do not reduce the total amount of external data traffic significantly.

7.1.2 Scene Management Algorithms

The primitives need to be sent to the rasterizer in tile-based order. Several of such scene management algorithms were developed and their computational cost and memory requirements were measured. A comprehensive description of the algorithms is presented in [9].

An important part of the scene management algorithm is the test that determines if a triangle overlaps a tile. Commonly employed is the so-called bounding box (BBOX) test, which checks if the axis-aligned bounding box of a triangle intersects the tile, as illustrated in Figure 7.3. It is commonly employed because of its cost; it requires only four comparisons. The BBOX test, however, is imprecise because it might be that the BBOX intersects with a tile, while the triangle does not. For example, in Figure 7.3 the BBOX overlaps the

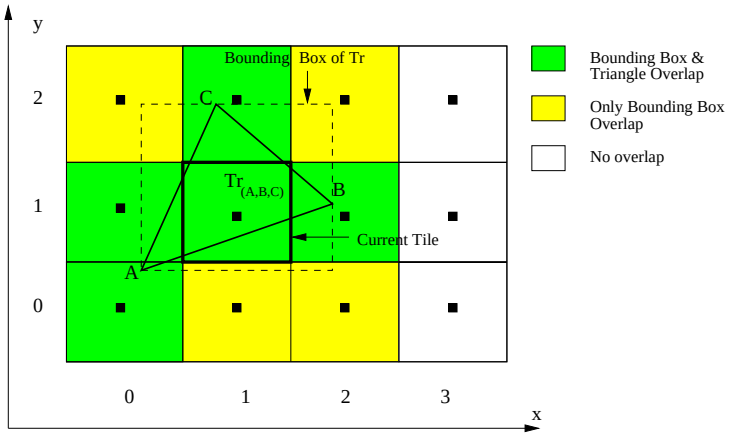

Figure 7.3: Triangle to tile BBOX test.

yellow tiles, but the triangle does not.

This is the main reason why we have developed an exact test, called the Linear Edge Test (LET). This test has been proposed before, but in a different context [78], and we have adapted it, so that no coverage mask is needed. The LET test, however, is computationally more expensive than the BBOX test.

We have proposed and evaluated the following scene management algorithms:

DIRECT: This algorithm simply scans all primitives for each tile and sends the primitives that (potentially) overlap the current tile to the rasterizer. We have used the BBOX test, but other tests can also be employed. By defining the scene buffer as the buffer that holds the initial scene geometry and state changes and optionally the geometry and state changes sorted per tile, the main advantage of the algorithm is that it requires no memory in addition to the scene buffer.

Let $tiles$ and $triangles$ be the number tiles and triangles, respectively. The time complexity of the algorithm DIRECT is

$$\begin{aligned} C &= t_{buf} \cdot triangles + \\ & \quad t_{bbox-total} \cdot tiles \cdot triangles + \\ & \quad t_{send} \cdot triangles \cdot bbox\_overlap \end{aligned} \tag{7.1}$$

where t_{buf} is the cost of placing a triangle in the scene buffer, $t_{bbox-total}$ is the cost of computing the bounding box of a triangle and determining if a bounding

box and a tile overlap, t_{send} is the cost of sending a triangle to the rasterizer, and $bbox\_overlap$ is the overlap factor if the bounding box test is employed.

TWO_STEP: This algorithm consists of two phases. In the first phase, the BBOX of each triangle is computed and stored in a buffer. This avoids having to recompute the BBOX for each triangle/tile tuple. In the second phase, all triangles are scanned for each tile and the triangles whose BBOX overlap the current tile are sent to the rasterizer.

The time complexity of this algorithm is

$$\begin{aligned} C &= (t_{buf} + t_{bbox-compute}) \cdot triangles + \\ &\quad t_{bbox-test} \cdot tiles \cdot triangles + \\ &\quad t_{send} \cdot triangles \cdot bbox\_overlap \end{aligned} \tag{7.2}$$

where $t_{bbox-compute}$ is the cost of computing the bounding box of a triangle and $t_{bbox-test}$ is the cost of testing if a bounding box of a triangle and a tile overlap (so $t_{bbox-total} = t_{bbox-compute} + t_{bbox\_test}$. It was assumed that the cost of storing a bounding box is negligible since a storing operation is performed by default while computing the bounding box components. However, even if a separate bounding box storing cost is added, its contribution to the total cost (complexity) is negligible.

The amount of additional memory required by the algorithm TWO_STEP is $triangles \cdot sizeof(bbox)$, where $sizeof(bbox)$ is the size of a bounding box structure, i.e., 4 integers.

TWO_STEP_LET: This algorithm is identical to TWO_STEP except that in the second phase the LET test is employed. Since the LET test contains the BBOX test, the main LET test is applied only to the triangles that passed the BBOX test.

The time complexity of this algorithm is

$$\begin{aligned} C &= (t_{buf} + t_{bbox-compute}) \cdot triangles + \\ &\quad t_{bbox-test} \cdot tiles \cdot triangles + \\ &\quad t_{let-test} \cdot triangles \cdot bbox\_overlap + \\ &\quad t_{send} \cdot triangles \cdot let\_overlap \end{aligned} \tag{7.3}$$

where $t_{let-test}$ is the cost of testing if a triangle and a tile overlap using LET test, and $let\_overlap$ is the LET overlap factor.

While the TWO_STEP_LET algorithm takes more time than the TWO_STEP algorithm, the number of triangles sent to the rasterizer by the TWO_STEP_LET algorithm ($triangles \cdot let\_overlap$) is lower than or equal to the number of triangles sent to the rasterizer in the TWO_STEP algorithm ($triangles \cdot bbox\_overlap$), since the LET test is accurate while the BBOX test is conservative. By sending less triangles to the hardware rasterizer this algorithm reduces the computational overhead at the rasterizer.

The amount of additional memory required by the algorithm TWO_STEP_LET is the same as for the TWO_STEP algorithm.

SORT: In this algorithm there is a buffer for each tile with pointers to the primitives that overlap the tile according to the BBOX test. For each tile, only the primitives that have a pointer in the corresponding buffer are sent to the rasterizer.

The time complexity of this algorithm is

$$\begin{aligned} C &= (t_{buf} + t_{bbox-compute}) \cdot triangles + \\ &\quad t_{insert} \cdot triangles \cdot bbox\_overlap + \\ &\quad t_{tiletrav} \cdot tiles + \\ &\quad t_{send} \cdot triangles \cdot bbox\_overlap \end{aligned} \tag{7.4}$$

where t_{insert} is the cost of inserting a pointer to a triangle in the buffer of the tile. There is no need to add a $t_{bbox-test}$ cost since there is no BBOX test performed (it can be determined from the bounding box coordinates in which tiles to insert pointers). The $t_{tiletrav}$ is the cost to traverse a tile.

The amount of additional memory required by the SORT algorithm is $triangles \cdot bbox\_overlap \cdot 2 \cdot sizeof(pointer) + tiles \cdot 2 \cdot sizeof(pointer)$, where $sizeof(pointer)$ denotes the size of a pointer (4bytes). In our current implementation, we use a (prealocated) linked list of pointers to primitive blocks.

SORT_LET: This algorithm is identical to SORT, except that in the second phase the LET test is used.

Parameter	Benchmarks						
	Q3L	Q3H	Tux	AW	ANL	GRA	DIN
$t_{bbox-compute}$	14.02	14.00	13.98	14.25	14.25	14.25	14.20
$t_{bbox-test}$	2.08	1.99	1.78	1.86	1.93	1.77	1.78
$t_{let-test}$	50.50	43.40	48.93	59.15	52.90	54.66	57.79

Table 7.2: Time complexity parameters for each workload.

Statistics	Benchmarks						
	Q3L	Q3H	Tux	AW	ANL	GRA	DIN
number of frames	1,379	1,379	1,363	603	600	599	600
average triangles/frame	3,350	3,436	1,825	11,053	4,455	3,681	4,150
max. triangles/frame	7,074	7,170	2,980	14,102	14,236	6,907	4,313
max. bbox/frame	19,288	53,154	11,789	18,822	37,771	11,233	9,858
max. let/frame	14,175	26,542	8,478	18,465	33,469	10,109	9,088
bbox_overlap	3.06	7.03	4.17	1.34	4.08	2.28	2.06
let_overlap	2.39	4.32	3.05	1.32	3.4	1.98	1.95
max. scene buffer memory required	594k	602k	250k	1,185k	1,196k	580k	362k

Table 7.3: Relevant characteristics of the benchmarks.

The time complexity of this algorithm is

$$\begin{aligned} C \;=\; & (t_{buf} + t_{bbox-compute}) \cdot triangles + \\ & t_{let-test} \cdot triangles \cdot bbox\_overlap + \\ & t_{insert} \cdot triangles \cdot let\_overlap + \\ & t_{tiletrav} \cdot tiles + \\ & t_{send} \cdot triangles \cdot let\_overlap \end{aligned} \tag{7.5}$$

The amount of additional memory required by the SORT_LET algorithm is $triangles \cdot let\_overlap \cdot 2 \cdot sizeof(pointer) + tiles \cdot 2 \cdot sizeof(pointer)$.

As described in [8], some of the parameters used to estimate the complexity of the algorithms ($t_{bbox-compute}$, $t_{bbox-test}$, $t_{let-test}$) can vary across the workloads. In order to reduce the errors obtained by estimating them statistically, programs were written to compute the average number of elementary operations needed to implement each test and obtained particular values for each workload. The results are presented in Table 7.2. It can be seen that the obtained $t_{bbox-compute}$ and $t_{bbox-test}$ parameters are quite uniform across

Ops. number	Benchmarks						
	Q3L	Q3H	Tux	AW	ANL	GRA	DIN
DIRECT	8.7M	34M	17.6M	108M	44.2M	35.8M	40.7M
TWO_STEP	1.7M	5.3M	2.4M	13.6M	6.2M	4.5M	5.0M
TWO_STEP_LET	1.8M	5.4M	2.4M	14M	6.4M	4.6M	5.2M
SORT	0.7M	1.3M	0.47M	1.4M	1.13M	0.6M	0.66M
SORT_LET	1.1M	2.0M	0.75M	2.3M	1.9M	1.0M	1.1M

Table 7.4: Number of elementary operations per frame for each scene management algorithm.

the workloads, while $t_{let-test}$ has a larger variation. Other parameters of the workloads such as the average or maximum number of triangles per frame are presented in Table 7.3. The *average triangles/frame* statistics represent the average number of triangles sent from the OpenGL library to our driver after backface culling. This number is actually the $triangles$ parameter used to compute the complexity of the algorithms. The *max. triangles* represents the maximum number of triangles sent for one frame. This number can be used to determine the maximum amount of memory required to buffer the triangles for one frame. This number can also be used to determine the computational power required for real-time operation.

For the other parameters, the following assumptions were employed: $t_{buf} = 50$, $t_{insert} = 6$ (two additions, three assignments, and one comparison), $t_{tiletrav} = 4$ (two comparisons, one assignment, and one increment), $t_{send} = 40$ (the number of memory writes currently used to transfer the data for a triangle in our simulator).

The average time taken by each scene management algorithm to process one frame of every benchmark is presented in Table 7.4, while Figure 7.4 depicts the time taken by each algorithm relative to the amount of time taken by algorithm DIRECT. As expected, algorithm DIRECT requires the largest number of operations by far, while SORT takes the least amount of time. On average, across all benchmarks, SORT is 44x faster than DIRECT. Even if the TWO_STEP algorithm also scans the entire scene buffer for each tile, it has reasonable performance. On average, it is 6x slower than SORT. Furthermore, TWO_STEP_LET is hardly slower than TWO_STEP and, therefore, preferable, since it sends fewer triangles to the rasterizer, which means that the computational load on the rasterizer is reduced. SORT_LET, on the other hand, is on average 1.6x slower than SORT.

The amount of memory required by each algorithm, in addition to the scene

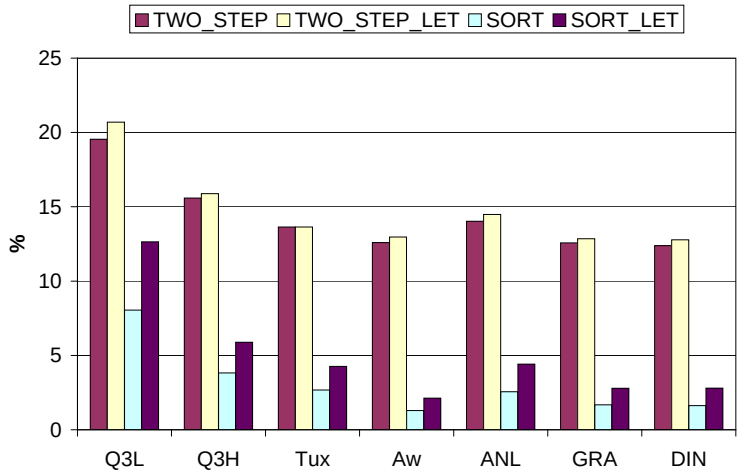

Figure 7.4: Estimated time taken by each scene management algorithm relative to the amount of time taken by algorithm DIRECT.

Max. memory	Benchmarks						
	Q3L	Q3H	Tux	AW	ANL	GRA	DIN
DIRECT	0	0	0	0	0	0	0
TWO_STEP/ TWO_STEP_LET	56.6k	57.4k	23.8k	112.8k	113.9k	55.26k	34.5k
SORT	155k	430k	99k	155k	306k	94k	83k
SORT_LET	115k	217k	72k	152k	273k	86k	78k

Table 7.5: Additional maximum memory requirements (bytes) per frame for each scene management algorithm.

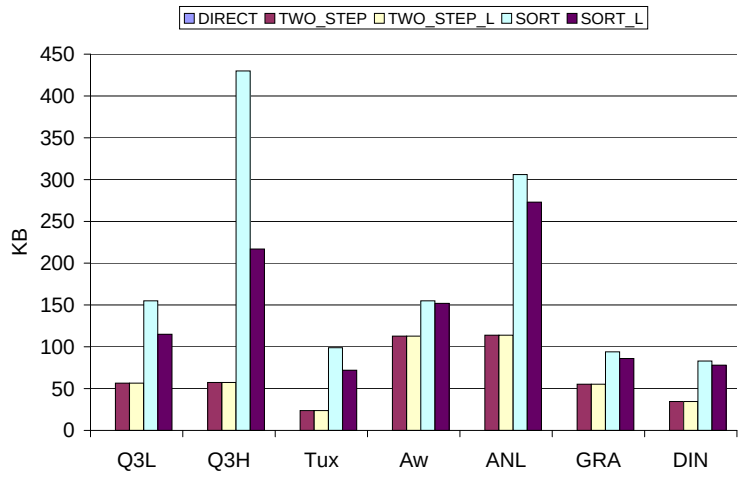

Figure 7.5: Memory requirements of the scene management algorithms.

buffer needed to store the primitives, is presented in Table 7.5 and Figure 7.5. As explained before, DIRECT does not require any additional memory. Furthermore, as expected, SORT needs the most additional memory, since it is proportional to the number of triangles and the BBOX overlap factor. Because the LET test is exact while the BBOX test is not, SORT_LET requires less memory than SORT. However, the difference is significant only for one benchmark (Q3H) for which SORT needs almost twice as much additional memory as SORT_LET. For the other benchmarks the difference is much smaller ($1.17\times$ on average). The reason is that the BBOX test is fairly exact for all benchmarks except Q3H. The TWO_STEP and TWO_STEP_LET algorithms require the same amount of memory. On average, TWO_STEP requires $3.2\times$ less additional memory than SORT. Notice that this difference depends strongly on the benchmark. For benchmarks with a small overlap factor, e.g., Aw, the difference is hardly significant, while for benchmarks with a large overlap factor (in particular Q3H), it is considerable.

7.1.3 State Management Algorithms

Tile-based rendering reduces the memory traffic between the rasterizer and the off-chip frame and z buffers. It increases, however, the amount of state change information such as enable/disable z testing and create/delete texture commands that need to be sent from the tiling engine to the rasterizer, because it may be necessary to send the same state change operation several times to the rasterizer. Consider, for example, the following instruction stream:

```
EnableDepth
Triangle(1)
DisableDepth
Triangle(2)
EnableDepth
Triangle(3)
```

Assume that tile 1 intersects with triangle 2 and 3 and that tile 2 intersects with triangle 1 and 3. If the tile-based driver duplicates the state change operations for each tile, then the following instruction stream is generated:

```
Tile 1
   EnableDepth
   DisableDepth
   Triangle(2)
```

```
   EnableDepth
   Triangle(3)

Tile 2
   EnableDepth
   Triangle(1)
   DisableDepth
   EnableDepth
   Triangle(3)
```

The italicized state change operations can be removed. For example, the first EnableDepth command may be removed because it is immediately followed by DisableDepth.

Determining which state change operations can be removed and when they should be removed is not always trivial. For instance, if a DeleteTexture command is encountered while rendering the current tile, the texture can be safely deleted only when all primitives (from all tiles) that use this texture are rendered or when multiple copies of the texture are kept in memory. Including all state change operations to each tile is not practical since it requires duplicating large amounts of state variables, e.g., texture objects. In some cases, the state change operations account for 63% of the data sent to the rasterizer.

In the following paragraphs, we describe two algorithms that handle the state information correctly when using tile-based rendering. The first, partial rendering, handles instructions with side-effects correctly, but may incur significant overhead. The second, delayed execution, reduces this overhead.

Partial Rendering Algorithm In this algorithm, whenever an instruction with side effects, e.g., DeleteTexture, is encountered, the driver renders all previously buffered instructions and then executes the instruction. While this is a solution to rendering commands with side effects, it may introduce significant rendering overhead. For each partial rendering, the introduced overhead consists of saving and reloading the contents of the enabled tile buffers, e.g., color, depth, and stencil, from the global buffers, plus the state information save and reload operations.

Delayed Execution Algorithm In this algorithm, when a command with side effects is encountered, the driver postpones its execution until all primitives depending on it have been rendered or until the end of the current frame is reached.

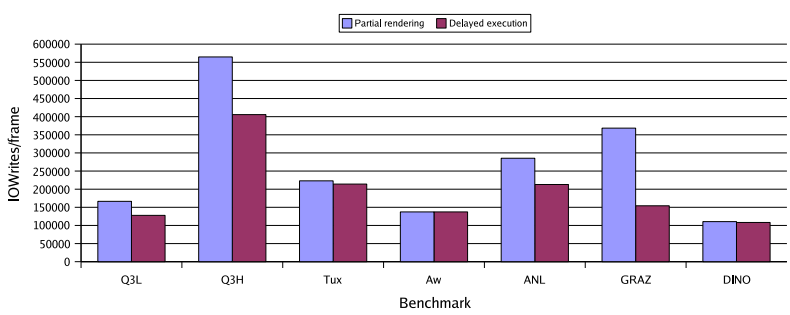

Figure 7.6: Average number of state information writes to the accelerator per frame.

Figure 7.6 depicts the amount of state change information sent to the rasterizer using partial rendering and delayed execution. The tile size is 32×32. The delayed method reduces the number of writes to the accelerator by filtering the state information and eliminating unnecessary writes. The amount of state information for the Q3L, Q3H, GRAZ, and ANL components is significantly reduced, by 23% to 58%. The state traffic for the Aw, Tux, and DINO workloads, however, is not decreased substantially, since tiling does not introduce many unnecessary state changes for these workloads.

7.2 Primitive List Hardware Acceleration

In an embedded system, the host processor has a general-purpose role and orchestrates the entire activity of the peripherals. Therefore, the scene management algorithm (for primitive tiling/binning) must not be a computational burden for the host processor. As much as possible, this computation has to be uploaded to the hardware rasterizer. In this section, we propose a hardware primitive list accelerator that will reduce the effort required to generate the tiling lists.

The primitive list hardware accelerator is able to store a number of the primitives on-chip and to perform tile binning based on the primitive bounding box test. This is achieved by a CAM memory with priority encoders on the outputs, using static RAM bit cells for storage, but dynamic domino logic for the arithmetic circuits to save area. The storage includes information related to global scene primitive vertex data and tags to global scene rasterization state, and the arithmetic circuits are able to perform primitive bounding box intersection

tests against the current tile boundaries. As the global scene rasterization data contains state changing commands, i.e, color shading, occlusion tests, color blending modes, and primitives in a strict sequential order, parallel queries in CAM are made using rasterization state tags and the current tile coordinates. The result is the sequence of rasterization state changing commands and the primitives local for the current tile that are sequentially transferred to the rest of the rasterization system for rendering, a tile at a time.

Hardware synthesis in a commercial 0.18um technology has indicated that the hardware implementation can be clocked at a frequency of 200MHz, and the rendering and the fill rate achieved are 2.4 million triangles/s and 460 million pixels/s for graphics scenes with typical average triangle area of 160 pixels.

7.2.1 The Baseline Algorithm

A sequence of graphic calls interleaves geometry and state, as follows:

Rasterization State S_0

Primitive P_0^0

$\vdots$

Primitive P_0^k

Rasterization State S_1

Primitive P_1^0

$\vdots$

Primitive P_1^l

$\vdots$

Rasterization State S_n

Primitive P_n^0

$\vdots$

Primitive P_n^m

The baseline scene management algorithm that we propose will have two components: one of them will be performed on the host processor, and another will be mapped on the proposed primitive list hardware accelerator. The part that will be executed on the host processor (Algorithm 7.2.1) is directly derived from TWO_STEP, with the exception that the scene will not be split in tiles, therefore the tile/triangle bounding box intersections are not performed. The triangles, as received from the application, and their bounding boxes are submitted directly to the primitive list hardware accelerator. In the next step, performed in hardware (Algorithm 7.2.2), all the tiles that are covered by triangles will be processed sequentially, one by one. For the current tile considered, the primitive list hardware accelerator does the current tile/triangle bounding box intersections in parallel on all triangles submitted by the host processor, and then sends only the triangles that are intersecting the current tile to the rasterizer automatically, one by one, preserving their submission order. When all the triangles for the current tile have been processed, the primitive list hardware accelerator can be queried again with the next tile, and the process repeats itself until all the tiles are processed.

The part that is executed on the host processor has the following pseudo-code:

Algorithm 7.2.1: FILLPRIMITIVELISTMEMORY()

$x_L \leftarrow +\infty$
$x_R \leftarrow -\infty$
$y_B \leftarrow +\infty$
$y_T \leftarrow -\infty$
for each state tag $S_j \in GlobalStateList$
{
  for each primitive tag $P_j^i \in PrimitiveList\,(S_j)$
  {
    Write $\left(S_j, P_j^i, x_{L_j}^i, x_{R_j}^i, y_{B_j}^i, y_{T_j}^i\right)$
    to primitive list memory
    Update the screen coverage box SCB
    {
      $x_L \leftarrow min\left(x_L, x_{L_j}^i\right)$
      $x_R \leftarrow max\left(x_R, x_{R_j}^i\right)$
      $y_B \leftarrow min\left(y_B, y_{B_j}^i\right)$
      $y_T \leftarrow max\left(y_T, y_{T_j}^i\right)$
    }
  }
}

Ops. number	Benchmarks						
	Q3L	Q3H	Tux	AW	ANL	GRA	DIN
TWO_STEP (SW)	1.7M	5.3M	2.4M	13.6M	6.2M	4.5M	5.0M
TWO_STEP (HW 132KB)	0.36M	0.37M	0.19M	1.2M	0.48M	0.39M	0.44M
TWO_STEP (HW 8KB)	0.47M	0.55M	0.26M	1.4M	0.63M	0.47M	0.53M

Table 7.6: Number of elementary operations per frame for the scene management algorithm with and without hardware primitive list acceleration.

Max. memory	Benchmarks						
	Q3L	Q3H	Tux	AW	ANL	GRA	DIN
TWO_STEP (SW)	56.6k	57.4k	23.8k	112.8k	113.9k	55.26k	34.5k
TWO_STEP (HW 132KB)	8.18k	8.39k	4.45k	26.98k	10.87k	8.98k	10.13k
TWO_STEP (HW 8KB)	9.81k	10.06k	5.35k	32.38k	13.05k	10.78k	12.16k

Table 7.7: Additional maximum memory requirements (bytes) per frame for each scene management algorithm on the host processor, with and without hardware primitive list acceleration.

The pseudo-code illustrates that for each possible primitive a record with the following fields will be transferred to the system memory: the state id or tag, the primitive index, and the primitive bounding box coordinates. Also the accumulated bounding box will be kept. The records, named by us the *primitive control stream*, will be subsequently transferred via DMA and stored in the arithmetic-enhanced CAM array contained in the primitive list hardware accelerator depicted in Figure 7.9.

With the notations introduced in Section 7.1, the time complexity of this algorithm is

$$\begin{aligned} C &= (t_{buf} + 1.2 \cdot t_{bbox-compute}) \cdot triangles + \\ &\quad t_{send} \cdot triangles \end{aligned} \tag{7.6}$$

The factor 1.2 is an empirical factor introduced to fit the experimental observations related to the bounding box computation expressed in tile coordinates.

The amount of additional memory required by the algorithm is $triangles \cdot sizeof(bbox)$, where $sizeof(bbox)$ is the size of a bounding box structure expressed in tile coordinates, therefore 4 5-bit values.

Under the legend key TWO_STEP(HW 132KB), the average time taken by the proposed scene management algorithm to process one frame of every benchmark on the host processor is presented in Table 7.6, while Figure 7.7 depicts

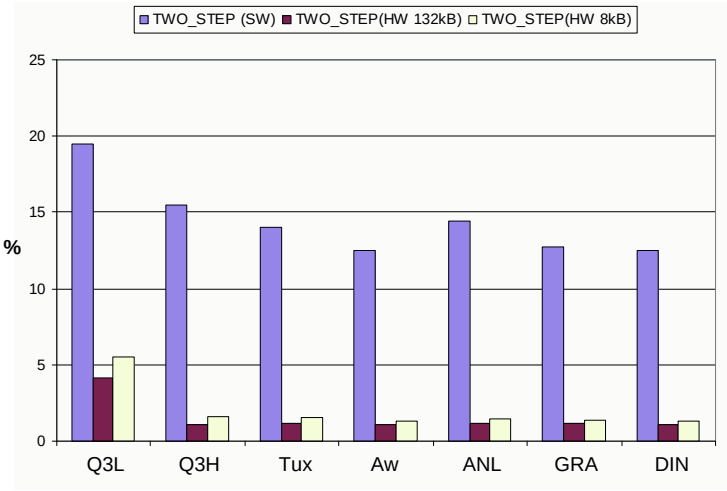

Figure 7.7: Time taken by each scene management algorithm on the host processor, relative to the amount of time taken by algorithm DIRECT with and without hardware primitive list acceleration.

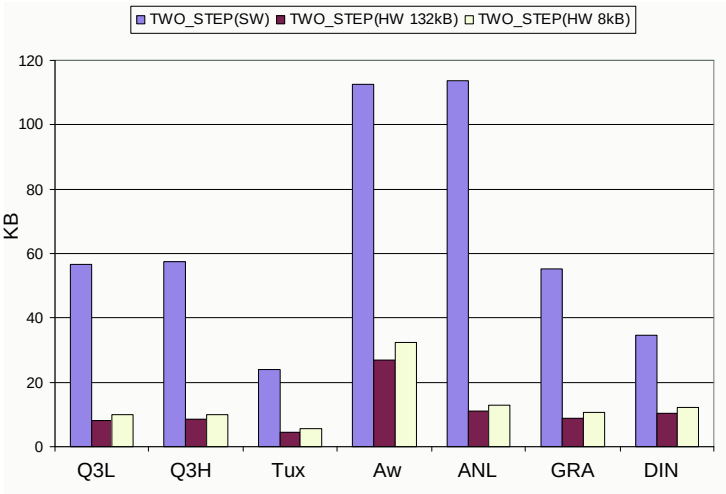

Figure 7.8: Memory requirements of the scene management algorithm on the host processor, with and without hardware primitive list acceleration.

the time taken against the amount of time taken by algorithm DIRECT (for cross reference purpose with Figure 7.4). By examining Table 7.4, it can be seen that the present algorithm consumes between 5–11 times less instructions than the original TWO_STEP algorithm, and between 1.3–2 times less instructions than the SORT algorithm (the fastest software algorithm). The host processor, therefore, will be freed to perform other tasks in the system, increasing the system responsiveness.

The amount of memory required by each algorithm, in addition to the scene buffer needed to store the primitives, is presented in Table 7.7 and Figure 7.8. When compared to Table 7.5 and Figure 7.5, this algorithm has a memory footprint 3–7 times smaller than the best SW algorithm in this category. This is less significant due to the fact that anyway the memory consumed is insignificant in the total bandwidth breakdown.

As it has been already mentioned, the primitive control stream (the tiling lists) from the host processor will be read from memory by the DMA and passed down to the primitive list hardware accelerator. Then, the accelerator, for each processed tile, will select and send triangles down one by one to the rasterizer. The primitive list hardware accelerator functionality can be summarized by the following pseudo-code:

Algorithm 7.2.2: READPRIMITIVELISTMEMORY()

for each $tile\,(X_C, Y_C) \in SCB\,(X_L, X_R, Y_B, Y_T)$
  for each state tag $S_j \in GlobalStateList$
    $\mathbf{ParallelQuery}\,(\mathbf{S_j}, \mathbf{X_C}, \mathbf{Y_C})$
      for each primitive list location $i \in [0, M-1]$
        $Hit\,(i) \leftarrow \left(X_C \in \left[{x_L}^i, {x_R}^i\right]\right) \bigwedge \left(Y_C \in \left[{y_B}^i, {y_T}^i\right]\right)$
        $\bigwedge Valid\,(i) \bigwedge (S_j = S_i)$
      $addr \leftarrow PriorityEncode\,(\{Hit(i)\})\,, i \in [0, M-1]$
      $P \leftarrow FetchPrimitiveTag\,(addr)$
      $SendToRasterizer\,(P)$
      $Valid\,(addr) \leftarrow$ **false**

The explanation of the algorithm described in pseudo-code is provided in the followings. For each tile in the accumulated bounding box and for each state tag whose state command has been applied, fetch sequentially, in the application submission order, the triangles whose bounding box are intersecting the current tile. Then apply a new state changing command and repeat the process,

until all the primitives for the current tile have been procesed. The algorithm continues with the selection of the next tile in the accumulated bounding box, until all tiles, and respectively all primitives have been processed. The searches are performed in parallel with the aid of the arithmetic-enhanced CAM depicted in Figure 7.9.

The inputs to the arithmetic-enhanced CAM are mainly the state tag and the coordinates of the currently processed tile. The parallel searches involve a state tag equality comparison and four comparisons between the current tile coordinates and the primitive bounding box coordinates. The resulting hits are priority encoded in order for the selected triangles to be generated in the application submission order.

The arithmetic-enhanced CAM is implemented similarly to a CMOS SRAM circuit. For write operations, the memory behaviour is identical to any CMOS SRAM read/write memory and will not be described (the memory function is realized statically with regenerative cross-coupled inverters). The logic circuitry in the comparators is implemented in domino dynamic logic, and the priority encoders are similar to the high-speed low-power *n*-type domino logic design described in [55].

The hit condition for the current tile coordinates to be within the primitive bounding box coordinates can be written as:

$$Hit = (x_L \leq x_C) \wedge (x_R \geq x_C) \wedge (y_B \leq y_C) \wedge (y_T \geq y_C) \tag{7.7}$$

where x_C and y_C are the current tile coordinates, and x_L, x_R, y_B, and y_T are the coordinates of the primitive bounding box.

Equation (7.7) can be rewritten as:

$$Hit = \overline{(x_L > x_C) \vee (x_R < x_C) \vee (y_B > y_C) \vee (y_T < y_C)} \tag{7.8}$$

which is a form amenable to wired OR implementation (the logic function will be a wired NOR in fact), and therefore suitable to domino logic implementation.

A row of the arithmetic-enhanced CAM is depicted in Figure 7.10. It contains two “Greater Than” cells, two “Less Than” cells for primitive bounding box comparison against the current tile, and an “Equal” cell for state tag comparison. If the comparison will result in a hit, the hit will be registered and taken into consideration by the priority encoder, and if the row is selected by the priority encoder, then the primitive index will be returned on `PrimTagBus` bus. The primitive index will be used to fetch the primitive pointer that looks

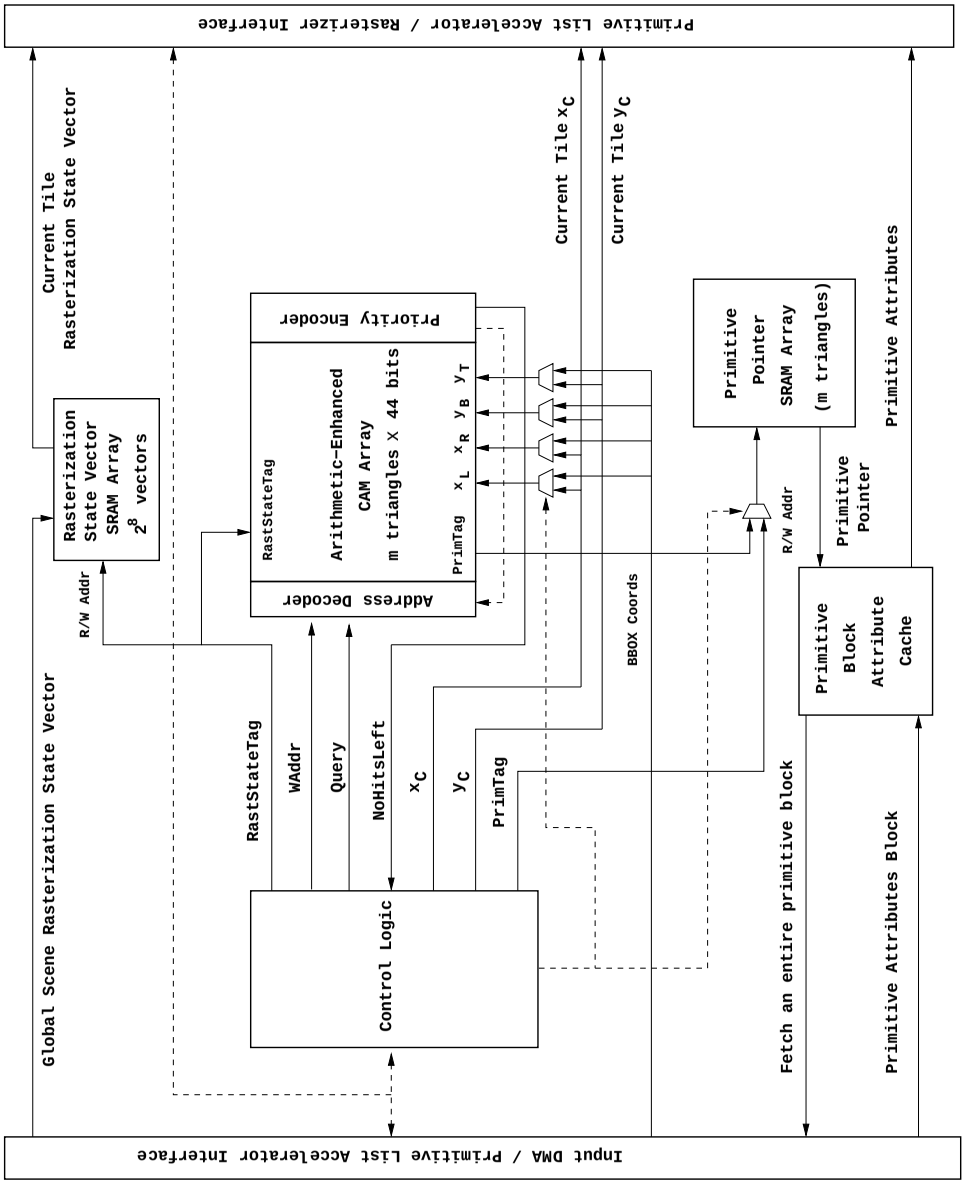

Figure 7.9: Primitive list accelerator block diagram

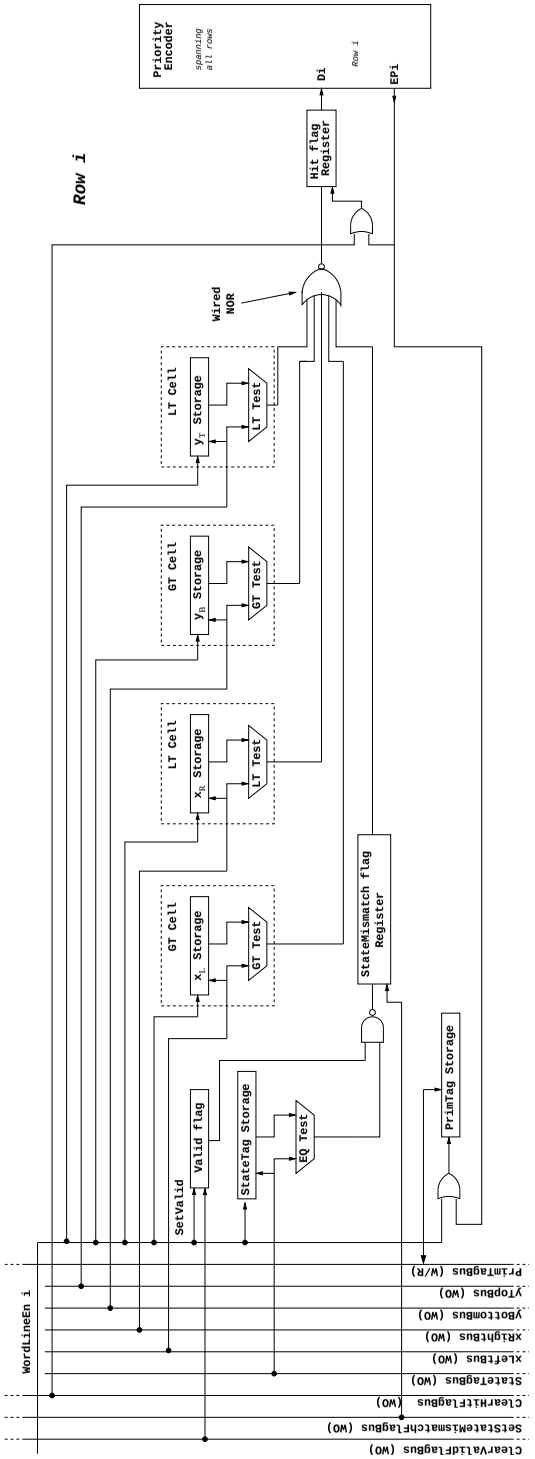

Figure 7.10: Arithmetic-enhanced CAM row

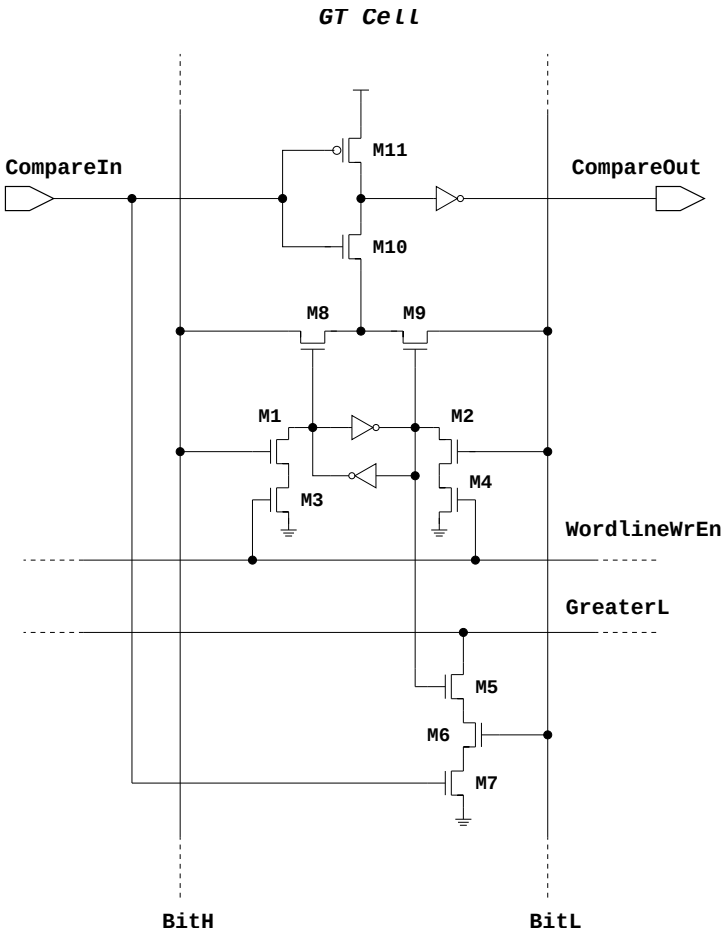

Figure 7.11: Comparator cell implementing a “Greater Than” function

up the primitive attributes from a primitive block attribute cache (presented in Figure 7.9).

The comparators will be described in the followings. Please note that the "Equal" cell is a frequently encountered cell in CAM memory design and there is no need to be described here again. The novel "Greater Than"comparator for a singular bit is depicted in Figure 7.11. The bit of the primitive bounding box coordinate is stored within the cross-coupled inverter and it is written with the aid of transistors M1, M2, M3, and M4 connected to the data signals `BitH` and `BitL`, and control signal `WordlineWrEn`. The floating inverter formed with the transistors M10, M11 will be conditionally enabled by the transistors M8 and M9 and will propagate the comparison enable signal `CompareIn` to `CompareOut` if the comparison could not be decided in this cell. The comparison function is ensured by transistors M5, M6, and M7 that are able to discharge the output signal `GreaterL`. Initially, the primitive bounding box is written within the cell by presenting the data bit on `BitH` and `BitL` and enabling `WordlineWrEn` signal. The stored value will be then available on the drain of the transistor M2. The aritmetic operation starts by presenting the current tile coordinate bit on `BitH` and `BitL` and precharging the output `GreaterL` to VDD. Then, the comparison enable signal `CompareIn` is set to 1 and the evaluation phase starts. Depending on the input and the stored value bits, there are four possible modes of operation:

1. Input=0 Storage=0 Transistors M6, and M7 will be on, but M5 will be off and, therefore, the `GreaterL` signal will float at precharged levels. Transistor M8 will be on and will enable the invertor M10 and M11 that will transfer the comparison enable signal to `CompareOut`, thus enabling the lesser significant bit cells because the comparison could not be decided;

2. Input=0 Storage=1 Transistors M5, M6, and M7 will be on and the GreaterL signal will be discharged to 0, meaning that the stored value is greater than the input and, therefore, the comparison is decided. Both transistors M8 and M9 will be off, so the invertor M10 and M11 will be disabled, as a result the `CompareOut` signal will not be able to follow the signal `CompareIn`, in effect disabling comparisons in the lesser significant bit cells;

3. Input=1 Storage=0 Transistor M7 will be on, but M5 and M6 will be off and, therefore, the `GreaterL` signal will float at precharged levels. Both transistors M8 and M9 will be off, so the invertor M10 and M11

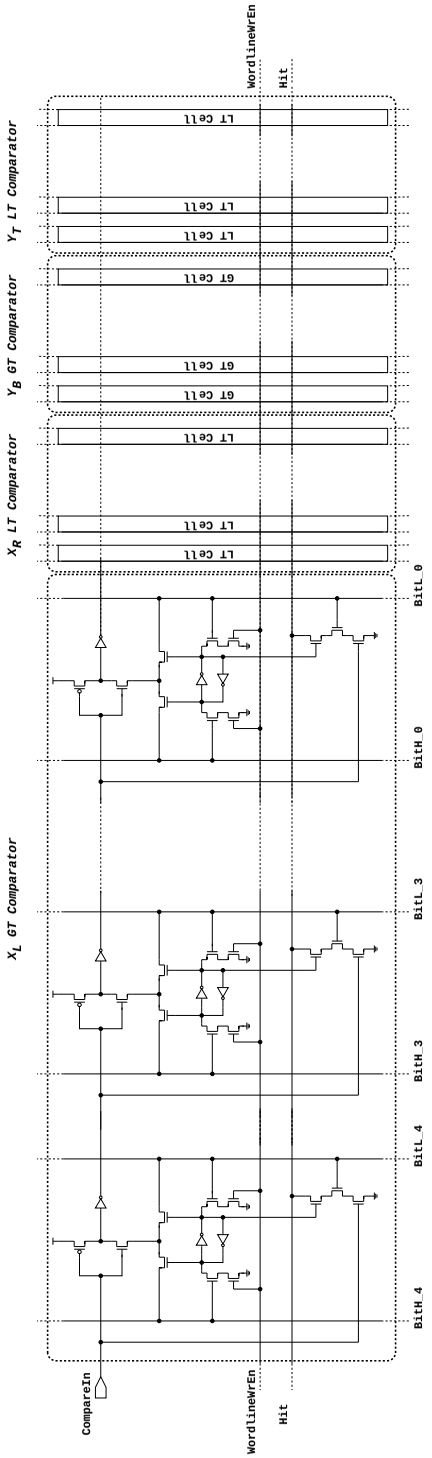

Figure 7.12: Abuttment of comparator cells in a wired NOR configuration

will be disabled, and consequently the `CompareOut` signal will not be able to follow the signal `CompareIn`, in effect disabling comparisons in the lesser significant bit cells. The comparison has been decided, meaning that the input value is greater than the stored value;

4. Input=1 Storage=1 Transistor M5, and M7 will be on, but M6 will be off and, therefore, the `GreaterL` signal will float at precharged levels. Transistor M9 will be on and will enable the invertor M10 and M11 that will transfer the comparison enable signal to `CompareOut`, thus enabling the lesser significant bit cells because the comparison couldn't be decided;

The end result of chaining comparator bit cells is that the signal `GreaterL` will be discharged to 0 only when the stored value is greater than the input value.

Similarly, a "Less Than" comparator cell can be described, the only difference in topology being that the connection of the transistors M5, M6, and M7 is mirrored along a vertical symmetry axis crossing the cell centre, with the operating principle being identical.

In order to implement Equation (7.8), different comparator cells in the CAM row can be abutted, as presented in Figure 7.12, where it can be noticed that the outputs and control signals are shared on the horizontal inteconnect lines across the row.

The hits generated with the afore-mentioned circuits will be registered and sent to the prority encoder. The implementation of the priority encoder and the selection of the rows is in principle similar to what has been described in Section 6.3.

Employing the described circuits, the data transferred per frame by the rasterizer to the frame buffer is presented under the legend key TWO_STEP(HW 132KB) in Figure 7.13.

The first iteration of the hardware organization has clear disadvantages:

- the arithmetic CAM accomodates 2^{13} primitives, so the size is $2^{13} \cdot 44$ bits, which is the equivalent of a 132KB SRAM — this is very large for an embedded system
- the priority encoder input is large amounting to 2^{13} bits - it implies a very large multi-stage priority-encoder that has a very long latency.

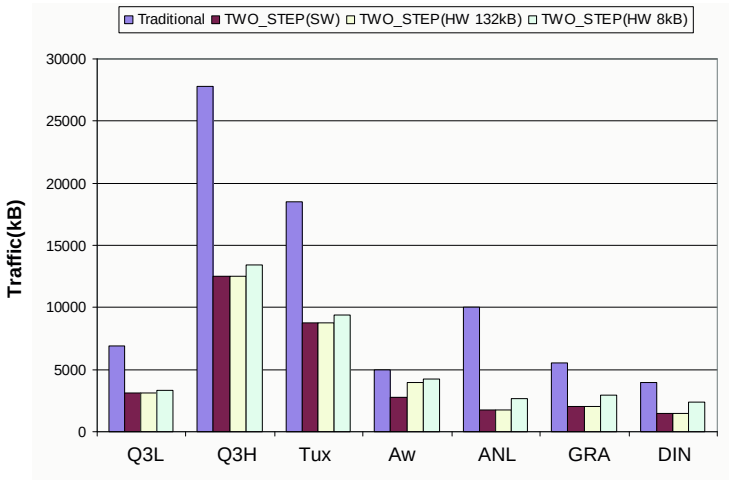

Figure 7.13: Comparison of data transferred (KB) per frame to the frame buffer by a traditional rasterizer, and a tile-based rasterizer with and without hardware primitive list acceleration.

7.2.2 A Feasible Algorithm

Reducing the CAM storage only to 512 triangles at a time, the CAM size will be $512 \cdot 44$ bits, which is the equivalent of a 8KB SRAM; this is feasible in an embedded system.

The scene management algorithm this time will split the screen in 16 coarse-grained tiles and will apply TWO_STEP.

With the notations introduced in Section 7.1, the time complexity of this algorithm is

$$\begin{aligned} C &= (t_{buf} + 1.2 \cdot t_{bbox-compute}) \cdot triangles + \\ &\quad t_{bbox-test-coarse} \cdot tiles/16 \cdot triangles + \\ &\quad t_{send} \cdot triangles \cdot bbox\_overlap\_coarse \end{aligned} \tag{7.9}$$

where $t_{bbox-test}$ is the cost of testing if a bounding box of a triangle and a coarse-grained tile overlap, and $bbox\_overlap\_coarse$ is the overlap factor for coarse-grained tiles.

The amount of additional memory required by the algorithm is $triangles \cdot ((sizeof(bbox) + coarse\_grained\_tile\_id)$, where $sizeof(bbox)$ is the size of a bounding box structure expressed in tile coordinates, therefore 4 5-bit

values, and $coarse\_grained\_tile\_id$ is the index of a coarse-grained tile, i.e., a 4-bit value.

Under the legend key TWO_STEP(HW 8KB), the average time taken by the proposed scene management algorithm to process one frame of every benchmark on the host processor is presented in Table 7.6, while Figure 7.7 depicts the same time relative to the amount of time taken by algorithm DIRECT (for cross reference purpose with Figure 7.4). By examining Table 7.4, it can be seen that the present algorithm consumes between 4–9 times less instructions than the original TWO_STEP algorithm, and between 1.2–1.9 times less instructions than the SORT algorithm (the fastest software algorithm). Consequently, the host processor will be freed to perform other tasks in the system, increasing the system responsiveness.

The amount of memory required by each algorithm, in addition to the scene buffer needed to store the primitives, is presented in Table 7.7 and Figure 7.8. When compared to Table 7.5 and Figure 7.5, this algorithm has a memory footprint 2.8–6.4 times smaller than the best SW algorithm in this category. This is less significant due to the fact that the memory consumed is insignificant in the total bandwidth breakdown.

The data transferred per frame by the rasterizer to the frame buffer is presented under the legend key TWO_STEP(HW 8KB) in Figure 7.13. The figure once again re-emphasizes the feasibility of the proposed CAM with a reduced size equivalent to an 8KB SRAM, which performs almost as well as the larger CAM, but without the implementation costs of the later.

7.3 Conclusion

In this chapter we have presented a hardware primitive list accelerator that lowers the effort on the host processor required to generate the tiling lists and reduces the external memory traffic at the same time. The primitive list hardware accelerator is able to store a number of the primitives on-chip and to perform tile binning based on the primitive bounding box test. This is achieved by a CAM memory with priority encoders on the outputs, using static RAM bit cells for storage, but dynamic domino logic for the arithmetic circuits to save area. The storage includes information related to global scene primitive vertex data and tags to global scene rasterization state, and the arithmetic circuits are able to perform primitive bounding box intersection tests against the current tile boundaries. As the global scene rasterization data contains state changing commands, i.e., colour shading, occlusion tests, color blending modes,

and primitives in a strict sequential order, parallel queries in CAM could be performed using rasterization state tags and the current tile coordinates. The result is the sequence of rasterization state changing commands and the primitives local for the current tile that are sequentially transferred to the rest of the rasterization system for rendering a tile at a time. We have shown that by using the proposed hardware primitive list accelerator the number of instructions needed on the host processor for primitive tile binning was reduced by 4–9 times and the memory footprint was reduced by 3–6 times for our embedded benchmark suite GraalBench, when compared to the software driver implementation alone. We also have indicated that the cost of the hardware primitive list accelerator could be accomodated by an embedded rasterizer being no more than the equivalent of an 8KB SRAM memory macro.

Chapter 8

Conclusions

In this dissertation, we presented a framework for developing embedded rasterization hardware for mobile devices, meant to accelerate real-time 3-D graphics applications. In particular, within this framework, we proposed a novel design for an embedded *tile-based* rasterizer called GRAphics AcceLerator (GRAAL). GRAAL is an OpenGL compliant rasterizer to be used in a tile-based rasterization scenario, designed to be low-cost, potentially low-power, having relatively high-performance, and delivering good quality image results. We have focussed on several key problem areas for tile-based rasterization such as: rasterization and triangle traversal, antialiasing, and primitive list sorting. The research activity called for devising new hardware algorithms in the afore-mentioned areas, and so included algorithm research, high-level architecture design, and hardware design involving high-level circuit synthesis and full-custom ASIC design at layout and circuit level. A significant effort has been made for the creation of targeted hardware/software co-design toolboxes and design flows for embedded graphics, in order to fully assess the merits of the proposed design. Overall, we have shown that a hardware implementation using an IC technology node of 0.18μm clocked at a frequency of 200MHz could achieve a rendering and a fill rate of 2.4 million triangles/s and 460 million pixels/s for typical 3-D graphics workloads, with costs and levels of power consumption suitable for mobile graphics.

This chapter summarizes our overall investigations and achievements. It is organized in three sections. Section 8.1 discusses the overall conclusions. Section 8.2 presents the major contributions. Section 8.3 proposes further research directions.

8.1 Summary

In this dissertation, we considered and solved a number of issues associated with tile-based rasterization. Our overall achievements can be summarized by the following.

In Chapter 2, a generic 3-D graphics pipeline was overviewed and the main operations performed were described by laying emphasis on the perspective-correct rasterization from a theoretical point of view. GRAAL, our proposed hardware rasterization engine, has implemented, with some variations, all the operations decribed in there as they are fundamental. The chapter also presented a brief description of the anti-aliasing theory and a classification of the existing hardware developments to cope with the aliasing problem. Ample references were made to the OpenGL specification, the 3-D graphics library chosen to be hardware accelerated by GRAAL, thus outlining the OpenGL embodiments of the theoretical aspects presented there-in.

In Chapter 3, an algorithmic view of a potential OpenGL-compliant tile-based hardware rasterization engine was described. In this context, the term *potential* referred to the proposal that constitutes a platform to build on towards full OpenGL compliance. This can be achieved only by a combination of software driver-level techniques and hardware algorithms implemented by the rasterization engine. The chapter focussed on the algorithms implemented in hardware, whereas the software driver-level issues that help augmenting the hardware capabilities were mentioned only when it was absolutely necessary. The rasterization engine described is mainly oriented on three-dimensional triangle rasterization, as all the other graphics primitives, e.g., points, lines, and general polygons, can be reduced to triangles at the software driver-level. The described rasterization engine is capable to perform well with a multiplicity of triangle rasterization methods, e.g., filled flat- or Gouraud-shaded, both aliased or antialiased, with the employed algorithms being selected to strike a balance between cost, power consumption, performance, and image quality.

Chapter 4 presented the GRAAL framework, a versatile hardware/software co-simulation and co-design tool for embedded 3-D graphics accelerators developed by us. Written in SystemC, it includes an extensive library of parameterizable graphics pipeline components that can be assembled in a graphics rasterizer and plugged together with third-party SystemC models of other various components (microprocessors, memories, and peripherals) to create an entire virtual simulation platform. GRAAL framework incorporates tools to assist in the visual debugging of the graphics algorithms implemented in hardware and

to estimate the performance in terms of throughput, power consumption, and area. We have used the framework extensively and effectively throughout the project to assess the merits of various proposed hardware implementations and software/hardware partitioning algorithms.

In Chapter 5, an efficient low-cost, low-power hardware implementation of a run-time pixel coverage mask generation algorithm for embedded 3-D graphics antialiasing purposes was presented. The area sampling algorithm exploits the quadrant symmetry property allowing the storage of only the coverage mask information for a few representative edges in one of the quadrants of the plane, the rest of the information being derived on the fly via computationally inexpensive operations. In addition, precomputing the coverage masks for generator edges spread non-uniformly in the angular space of quadrant one, we reduced the maximum error in coverage from 15.25% (assumed by previous state of the art implementations of similar antialiasing schemes) to 8.34%, while reducing the implementation area significantly by an order of magnitude.

Chapter 6 described an efficient tile-based traversal algorithm hardware implementation to accelerate primitive traversal in 3-D graphics tile-based rasterizers. The hardware implementation consists of two components: a systolic primitive scan-conversion subsystem, using edge functions, and a logic-enhanced memory, which is filled in several clock cycles with the shape of a new triangle by the systolic subsystem. The memory internal logic is then capable of delivering up to four pixels per clock cycle to the pixel processing pipelines, in a spatial pattern which is very advantageous for texture caching and for reducing bank clashes in multi-banked SRAM tile buffers, used for read-modify-write operations associated with depth test and colour blending. The ghost primitives generated by trivial triangle tile binning implementations in tile-based rasterization systems are also discarded very fast in 4 clock cycles, reducing significantly the impact on the triangle throughput in such systems. The hardware implementation has shown that such a design could be clocked at a frequency of at least 200Mhz with reasonable cost and power consumption figures.

In Chapter 7, we presented a hardware primitive list accelerator that lowers the effort on the host processor required to generate the tiling lists and reduces the external memory traffic at the same time. The primitive list hardware accelerator is able to store a number of the primitives on-chip and to perform tile binning based on the primitive bounding box test. This is achieved by a CAM memory with priority encoders on the outputs, using static RAM bit cells for storage, but dynamic domino logic for the arithmetic circuits to save

area. The storage includes information related to global scene primitive vertex data and tags to global scene rasterization state, and the arithmetic circuits are able to perform primitive bounding box intersection tests against the current tile boundaries. As the global scene rasterization data contains state changing commands, i.e., colour shading, occlusion tests, color blending modes, and primitives in a strict sequential order, parallel queries in CAM could be performed using rasterization state tags and the current tile coordinates. The result is the sequence of rasterization state changing commands and the primitives local for the current tile that are sequentially transferred to the rest of the rasterization system for rendering a tile at a time. We showed that by using the proposed hardware primitive list accelerator, the number of instructions needed on the host processor for primitive tile binning was reduced by 4–9 times and the memory footprint was reduced by 3–6 times, for our embedded benchmark suite GraalBench, when compared to the software driver implementation alone. We also showed that the cost of the hardware primitive list accelerator could be accomodated by an embedded rasterizer being no more than the equivalent of an 8KB SRAM memory macro. Hardware synthesis has also indicated that the hardware implementation clocked at a frequency of 200MHz could sustain a rendering and fill rate of 2.4 million triangles/s and 460 million pixels/s for typical 3-D graphics scenes.

8.2 Contributions

The major contributions of this study can be summarized as follows:

- We have proposed a versatile hardware/software co-simulation and co-design tool framework for 3-D graphics accelerators. The tool framework offers a coherent development methodology based on an extensive library of parametrizable graphics pipeline components modelled at RT-level in SystemC. The framework is an open system, allowing integration with other third-party SystemC models to enable an entire embedded platform simulation if desired. The framework incorporates tools to assist in the visual debugging of the graphics algorithms implemented in hardware, and to estimate the performance in terms of throughput, power consumption, and area.

- We have presented a complete mathematical formalism that could be applied to any tile-based raterization engine. We have described how, after an initial computational stage called triangle setup, which is relative to

the current tile and current triangle, operations could be performed to each pixel (or pixel block), in parallel to other pixels (or pixel blocks), to generate the triangle stencil or the attributes that are required by the pixel processing pipelines. Also, we have presented how values, for neighbouring pixels ocurring within the same pixel block, could be derived using only two-operand additions, which are cheaper to implement in hardware than multiplications.

- We have proposed an efficient hardware triangle traversal algorithm to accelerate primitive traversal in 3-D graphics tile-based rasterizers that has a maximum ghost triangle overhead of 4 clock cycles, a latency of several clock cycles and could deliver a throughput of up to four pixels per clock cycle to the pixel pipelines for each triangle.

- We have proposed an efficient, high image quality run-time pixel coverage mask generation algorithm for embedded 3-D graphics antialiasing purposes, that is compatible with the above triangle traversal algorithm. The algorithm was implemented assuming 4×4 subpixel coverage masks and two's complement number representation. However, it has a higher degree of generality: it can be incorporated in any antialiasing scheme with pre-filtering that is based on algebraic representation of primitive's edges, it is independent of the underlying number representation, and it can be adapted to other coverage mask subpixel resolutions with the only prerequisite for the masks to be square. For the presented hardware implementation, the costs are reduced by an order of magnitude and the image quality almost doubles when compared to prior state-of-the-art implementations.

- We have designed the afore-mentioned proposed algorithms to deliver the pixels to the pixel pipelines in a special spatial pattern, i.e., Morton order, one of the space filling curves, that increases the hit ratio of texture caches and allows for the four pixels, generated simultaneously, to always be mapped to different memory banks in the local tile framebuffers thus breaking the read-modify-write dependencies associated with depth test and colour blending. As a result, the power consumption is reduced and the performance is increased.

- We have proposed a novel and efficient hardware primitive list sorting algorithm that lowers on the one hand the effort of the host processor required to generate the primitive tiling lists and reduces on the other hand the external memory traffic. For an implementation footprint similar to

an 8KB SRAM memory macro, the number of instructions on the host processor for tiling list generation was lowered by 4–9 times and the memory cost by 3–6 times, for our embedded benchmark suite GraalBench, when compared to the software driver implementation alone.

- We have designed novel hardware circuitry to implement, in a very efficient manner, the algorithms presented above. Driven by the ever increasing delays in the interconnect networks with each technology node, we have adopted modern implementation techniques for embedded design, that not so long ago were the attributes of high-performance computing: high-throughput circuitry, computation units and data storage interwoven together, and a re-compute rather than a compute-once distribute-and-reuse-many-times strategy. Therefore, the triangle traversal algorithm uses a systolic primitive scan-conversion subsystem that has a throughput of 16 pixels per clock cycle. In addition, as a part of the same triangle traversal algorithm, and for the primitive list sorting algorithm, a logic(arithmetic)-enhanced memory is employed. Special considerations were given 1) not to compromise the operational noise margins of the circuitry and 2) the enhancing logic(arithmetic) cells to have a layout with a similar pitch to the data storage cells in order to facilitate high cell integration densities. Therefore, in the logic(arithmetic)-enhanced memory, the storage cells were implemented with traditional SRAM circuitry (two cross-coupled inverters generating the storing latch and two NMOS pass transistors for access), but the logic(arithmetic) cells were implemented in a domino dynamic logic style that enabled all the features described above.

8.3 Proposed Research Directions

As a continuation of the research we suggest the following:

- An interesting emerging technology for flexible display, from DIMES, paving the way to truly ”smart”displays (i.e., packing logic functions and display functions together in a layered sandwich under the screen pixel) is the Single Grain TFT technology [77]. While existing OLED technologies for flexible displays provide very slow TFTs (thin film transistors) unsuitable even for medium-speed logic implementations, the SG TFT technology has the merit of providing TFTs with increased carrier mobility. The SG TFT technology from DIMES currently makes

steady advances, promising in a number of years to close the performance gap to bulk silicon, and therefore starts becoming an attractive proposition for integrating a miscellanea of logic functions under the display proper. As presented in a recent feasibility study [82], I also believe that moving rasterizer stage functionality to the ”smart” displays using DIMES SG TFT technology would be a natural and logic step to pursue, spawning an interesting class of new hardware architectures and circuit designs, including customized logic enhanced memories akin to the ones proposed in this thesis.

- In this dissertation we addressed an area sampling antialiasing method, and full scene antialiasing methods such as supersampling or multisampling with intra-pixel sample resolve case could be readily added to the rasterizer, as they are fully understood (in these cases the on-chip tile buffer stores samples instead of pixels and filters them to pixels prior to the transfer to the external framebuffer). However, we have not addressed the multisampling scheme with *inter*-pixel sample resolve case for tile-based rasterizers (in which pixels have to share, for final filtering, samples with neighbouring pixels), although it generates the same reported image quality but it only requires half of the number of samples and consequently it will require half the storage and computation. Future investigations could find the scheme worthwhile with some trade-offs, but for the current work, we have considered that the scheme has some apparent difficulties at the tile edges introducing inter-dependencies in the tile processing and therefore adding to the implementation costs, and also potentially having an adverse knock on effect on external memory traffic between the rasterizer and system memory.

- One of the logic(arithmetic)-enhanced memory problems is that it requires rather significant decoupling capacitors in the voltage supply rails and a careful layout of the voltage supply network, in order to prevent a supply voltage drop, that degrades noise margins, due to transient currents and IR effects (when a new computation with a new tile is requested at the circuit inputs). For this reason, a possible research direction would be to investigate various trade-offs for a new memory controller, which can stagger in time the start of the internal computations and alleviate the afore-mentioned shortcomings.

- In this thesis, we have considered that the individual triangles are written by the host processor into the primitive lists in memory in a linear fashion. We recognize that this strategy does not make efficient use

of the system bus, which is a high-latency, high-throughput device for which optimal transfers are performed by bursting blocks of data. At the same time, the triangles are seldom sent by the OpenGL library as individual entities and more often they are clustered using various compact topologies such as triangle meshes and triangle fans, defined by vertex and index buffers. As future work, it would be interesting to investigate formats for linked block structures in the system memory, which could store the vertex and the index buffers and also make efficient usage of the system bus. In this way, further reductions in the external memory traffic could be envisaged.

- Whenever the number of triangles is too large to be handled by the hardware implementation, the rasterizer has to perform a series of partial renders. The implication is that, after each tile which is rasterized, the local tile buffers have to be written to the system memory in order for the next partial render (that may rasterize to the same tile) to be able to continue as if the partial renders have not existed at all. As the synthesized images exhibit in general a degree of colour continuity, a new research direction would be to study loseless compression algorithms, with random access for decompression, suitable for writing the local tile buffers to memory, and therefore reduce the external memory traffic even further.

- In this dissertation we have addressed only the fixed function rasterization pipeline. An interesting research area would be to add full programability to the pixel pipelines, by designing a pixel pipeline processor that could map efficiently the existing shading languages to a new instruction set, and that can make use of deferred shading techniques (exploitable in the tile-based rasterization context) for further energy savings.

Bibliography

[1] Wikipedia. *http://en.wikipedia.org*.

[2] ARM MBX HR-S 3D Graphics Core — Technical Overview. ARM Ltd. and Imagination Technologies Ltd., 2003.

[3] Timo Aila, Ville Miettinen, and Petri Nordlund. Delay Streams for Graphics Hardware. *ACM Trans. Graph.*, 22:792–800, July 2003.

[4] T. Akenine-Möller and E. Haines. *Real-Time Rendering*. A K Peters, Ltd., 2002.

[5] Tomas Akenine-Möller and Jacob Ström. Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones. *ACM Trans. Graph.*, 22:801–808, July 2003.

[6] Alliance tools, URL: http://www-asim.lip6.fr/recherche/alliance/.

[7] I. Antochi, B. Juurlink, S. Vassiliadis, and P. Liuha. GraalBench: A 3D Graphics Benchmark Suite for Mobile Phones. In *Proceedings of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04)*, June 2004.

[8] I. Antochi, B.H.H. Juurlink, S. Vassiliadis, and P. Liuha. Scene Management Models and Overlap Tests for Tile-Based Rendering. In *Proc. EUROMICRO Symp. on Digital System Design*, 2004.

[9] Iosif Antochi. *Suitability of Tile-Based Rendering for Low-Power 3D Graphics Accelerators*. PhD thesis, Delft University of Technology, October 2007.

[10] ARM Limited. *ARM Developer Suite version 1.1*, 1999.

[11] Arm Ltd. AMBA Specification. 1999.

[12] A.C. Barkans. High-Quality Rendering Using the Talisman Architecture. In *1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 79–88, 1997.

[13] Andrew C. Beers, Maneesh Agrawala, and Navin Chaddha. Rendering from Compressed Textures. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 373–378, New York, NY, USA, 1996. ACM.

[14] J.F. Blinn. Hyperbolic Interpolation. *IEEE Computer Graphics and Applications*, 12(4):89–94, July/August 1992.

[15] J.F. Blinn. W Pleasure, W Fun. *IEEE Computer Graphics and Applications*, 18(3):78–82, May/June 1998.

[16] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 83–94, Vancouver, BC, June 2000.

[17] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report Nr. 1342, University of Wisconsin-Madison Computer Sciences Department, June 1997.

[18] Bloomberg Businessweek. PWC: Video Game Industry to Drive Entertainment Sector. *http://www.businessweek.com*, October 2005.

[19] L. Carpenter. The A-Buffer, an Antialiased Hidden Surface Removal Method. *ACM SIGGRAPH '84 Conference Proceedings*, 18:103–108, 1984.

[20] Shek-Wayne Chan and Chin-Long Wey. The Design of Concurrent Error Diagnosable Systolic Arrays for Band Matrix Multiplications. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 7(1):21–37, 1988.

[21] Milton Chen, Gordon Stoll, Homan Igehy, Kekoa Proudfoot, and Pat Hanrahan. Simple Models of the Impact of Overlap in Bucket Rendering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, HWWS '98, pages 105–112, New York, NY, USA, 1998. ACM.

[22] P. Christie and D. Stroobandt. The Interpretation and Application of Rent's Rule. *IEEE Trans. Very Large Scale Integr. Syst.*, 8(6):639–648, 2000.

[23] Ronald J. Cosentino. Concurrent Error Correction in Systolic Architectures. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 7(1):117–125, 1988.

[24] M. Cox and N. Bhandari. Architectural Implications of Hardware-Accelerated Bucket Rendering on the PC. In *Proc. 1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 25–34. ACM Press, 1997.

[25] M. Cox, N. Bhandari, and M. Shantz. Multi-Level Texture Caching for 3D Graphics Hardware. In *Proceedings of ISCA 98: International Symposium on Computer Architecture*, pages 86–97, 1998.

[26] Michael Cox, Narendra Bhandari, and Michael Shantz. Multi-Level Texture Caching for 3D Graphics Hardware. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 86–97. IEEE Press, 1998.

[27] D. Crisu, S. Cotofana, and S. Vassiliadis. A Proposal of a Tile-Based OpenGL Compliant Rasterization Engine - Progress Report. Technical Report (2004-01), Computer Engineering Laboratory, EEMCS, Delft University of Technology, March 2004.

[28] D. Crisu, S. D. Cotofana, S. Vassiliadis, and P. Liuha. 3D Graphics Tile-Based Systolic Scan-Conversion. In *Thirty-Eighth Asilomar Conference on Signals, Circuits, and Systems*, pages 517 – 521, November 2004.

[29] D. Crisu, S. D. Cotofana, S. Vassiliadis, and P. Liuha. GRAAL - A Development Framework for Embedded Graphics Accelerators. In *Proceedings of Design, Automation and Test in Europe (DATE'04)*, pages 1366–1367, February 2004.

[30] D. Crisu, S. D. Cotofana, S. Vassiliadis, and P. Liuha. High-Level Energy Estimation for ARM-Based SOCs. In *Lecture Notes in Computer Science (Proceedings of the Third International Workshop on Computer Systems: Architectures, Modeling and Simulation SAMOS III)*, pages 168–177, November 2004.

[31] D. Crisu, S. D. Cotofana, S. Vassiliadis, and P. Liuha. Logic-Enhanced Memory for 3D Graphics Tile-Based Rasterizers. In *Proceedings of the 2004 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS 2004)*, pages II–237 – II–240, July 2004.

[32] D. Crisu, S.D. Cotofana, and S. Vassiliadis. A Proposal of a Tile-Based OpenGL-Compliant Rasterization Engine. Technical report, Computer Engineering Laboratory, Delft University of Technology, Deliverable no. (2002)–02, 2002.

[33] D. Crisu, S.D. Cotofana, S. Vassiliadis, and P. Liuha. Efficient Hardware for Antialiasing Coverage Mask Generation. In *Proceedings of Computer Graphics International Conference 2004 (CGI 2004)*, pages 257–264, June 2004.

[34] D. Crisu, S.D. Cotofana, S. Vassiliadis, and P. Liuha. Determining a Coverage Mask for a Pixel. In *US Patent No: 7,006,110 B2*, February 2006.

[35] D. Crisu, S. Vassiliadis, S. D. Cotofana, and P. Liuha. Low Cost and Latency Embedded 3D Graphics Reciprocation. In *Proceedings of 2004 IEEE International Symposium on Circuits and Systems (ISCAS 2004)*, pages II–905 – II–908, May 2004.

[36] M. O. Esonu, A. J. Al-Khalili, S. Hariri, and D. Al-Khalili. Design Techniques for Fault-Tolerant Systolic Arrays. *The Journal of VLSI Signal Processing*, 11:151–168, 1995. 10.1007/BF02106828.

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

[38] Simon Fenney. Texture Compression Using Low-Frequency Signal Modulation. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '03, pages 84–91, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[39] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[40] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principles and Practice, Second Edition in C*. Addison-Wesley, 1996.

[41] A. Fountain, J. Huxtable, P. Ferguson, and D. Heller. *The Definitive Guides to the X Window System, Vol. 6A — Motif Programming Manual for Motif 2.1*. O'Reilly & Associates, Inc., 2001.

[42] Richard Fromm, Stylianos Perissakis, Neal Cardwell, Christoforos Kozyrakis, Bruce McGaughy, David Patterson, Tom Anderson, and Katherine Yelick. The Energy Efficiency of IRAM Architectures. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 327–337, New York, NY, USA, 1997. ACM.

[43] H. Fuchs, J. Goldfeather, J.P. Hultquist, S. Spach, J.D. Austin, F.P. Brooks, J.G. Eyles, and J. Poulton. Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes. *Computer Graphics (ACM SIGGRAPH '85 Conference Proceedings)*, 19(3):111–120, 1985.

[44] Tohru Furuyama. Trends and Challenges of Large Scale Embedded Memories. In *Proceedings of the IEEE 2004 Custom Integrated Circuits Conference*, pages 449–456, 2004.

[45] G.E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, (38):114–117, April 1965.

[46] G.M.Amdahl and G.A. Blaauw and F.P.Brooks. Architecture of the IBM System/360. *IBM Journal of Research and Development*, (8(2)):87–101, 1964.

[47] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[48] Khronos Group. OpenGL ES - The Standard for Embedded Accelerated 3D Graphics. *http://www.khronos.org/opengles/*, 2008.

[49] P. Haeberli and K. Akeley. The Accumulation Buffer: Hardware Support for High-Quality Rendering. *Computer Graphics*, 24(4):309–318, August 1990.

[50] Ziyad S. Hakura and Anoop Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 108–120. ACM Press, 1997.

[51] Paul S. Heckbert. Fundamentals of Texture Mapping and Image Warping. Master's thesis, University of California, Berkeley, 1989.

[52] A. Herrera. Technology and Solutions for Antialiasing of Computer Graphics. Technical report, Jon Peddie Associates, 2000.

[53] Hans Horten-Lund. *Design for Scalability in 3D Computer Graphics Architectures*. PhD thesis, Technical University of Denmark, July 2001.

[54] Emile Hsieh, Vladimir Pentkovski, and Thomas Piazza. ZR: a 3D API Transparent Technology for Chunk Rendering. In *Proceedings of the 34th annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 284–291, Washington, DC, USA, 2001. IEEE Computer Society.

[55] C.-H. Huang, J.-S. Wang, and Y.-C. Huang. Design of High-Performance CMOS Priority Encoders and Incrementers/Decrementers Using Multilevel Lookahead and Multilevel Folding Techniques. *IEEE Journal of Solid-State Circuits*, 37(1):63–76, January 2002.

[56] Homan Igehy, Matthew Eldridge, and Kekoa Proudfoot. Prefetching in a Texture Cache Architecture. In *Proceedings of the 1998 EUROGRAPHICS/SIGGRAPH Workshop on Graphics Hardware*, pages 133–142. ACM Press, 1998.

[57] Joseph A. Fisher and Paolo Faraboschi and Cliff Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers, and Tools*. Morgan Kaufmann Publishers, 2005.

[58] N.P. Jouppi and C.-F. Chang. Z^3: An Economical Hardware Technique for High-Quality Antialiasing and Transparency. In *SIGGRAPH Eurographics 1999 Hardware Workshop in Computer Graphics*, 1999.

[59] Ben Juurlink, Iosif Antochi, Dan Crisu, Sorin Cotofana, and Stamatis Vassiliadis. GRAAL: A Framework for Low-Power 3D Graphics Accelerators. *IEEE Computer Graphics and Applications*, 28(4):63–73, July/August 2008.

[60] Masatoshi Kameyama, Yoshiyuki Kato, Hitoshi Fujimoto, Hiroyasu Negishi, Yukio Kodama, Yoshitsugu Inoue, and Hiroyuki Kawai. 3D Graphics LSI Core for Mobile Phone "Z3D". In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '03, pages 60–67, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[61] B. Kapoor. Low Power Memory Architectures for Video Applications. In *Proceedings of 8th Great Lakes Symposium on VLSI*, pages 2–7, 1998.

[62] Paul Landman. High-Level Power Estimation. In *International Symposium on Low Power Electronics and Design*, pages 29–35, Monterey CA, 1996.

[63] E. Lapidous and G. Jiao. Optimal Depth Buffer for Low-Cost Graphics Hardware. In *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 67–73, 1999.

[64] O. Lathrop, D. Kirk, and D. Voorhies. Accurate Rendering by Subpixel Addressing. *IEEE Computer Graphics and Applications*, 10(5):45–53, September/October 1990.

[65] Y. Li and J. Henkel. A Framework for Estimating and Minimizing Energy Dissipation of Embedded HW/SW Systems. In *Proceedings of Design Automation Conference*, pages 188–193, 1998.

[66] J. McCormack and R. McNamara. Tiled Polygon Traversal Using Half-Plane Edge Functions. In *Proceedings of the 2000 SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 15–21, 2000.

[67] The MESA 3D Graphics Library, URL: http://www.mesa3d.org.

[68] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[69] Microsoft DirectX, URL: http://www.microsoft.com/windows/directx/.

[70] Steven Molnar, John Eyles, and John Poulton. PixelFlow: High-Speed Rendering Using Image Composition. In *Computer Graphics*, pages 231–240, 1992.

[71] R. Negrini, M. Sami, and R. Stefanelli. Fault Tolerance Techniques for Array Structures Used in Supercomputing. *Computer*, 19:78–87, 1986.

[72] Gartner Newsroom. Gartner Says Spending on Gaming to Exceed 74 Billion in 2011. *http://www.gartner.com/it/page.jsp?id=1737414*, July 2011.

[73] The Open SystemC Initiative (OSCI), URL: http://www.systemc.org.

[74] J. Pineda. A Parallel Algorithm for Polygon Rasterization. *Computer Graphics*, 22(4):17–20, August 1988.

[75] Vincenzo Piuri. Fault-Tolerant Systolic Arrays: An Approach Based Upon Residue Arithmetic. In *IEEE Symposium on Computer Arithmetic*, pages 230–238, 1987.

[76] Kari Pulli. New APIs for Mobile Graphics. 2008.

[77] Vikas Rana. *Single Grain Si TFTs and Circuits Based on the μ-Czochralski Process*. PhD thesis, Delft University of Technology, October 2006.

[78] A. Schilling. A New Simple and Efficient Antialiasing with Subpixel Masks. *Computer Graphics*, 25(4):133–141, July 1991.

[79] A.G. Schilling and W. Straßer. EXACT: Algorithm and Hardware Architecture for an Improved A-Buffer. *Computer Graphics*, pages 85–91, 1993.

[80] M. Segal and K. Akeley. *The OpenGL Graphics System: A Specification (Version 1.2.1)*. Silicon Graphics, Inc., 1999.

[81] Edwin Hsing-Mean Sha and Kenneth Steiglitz. Error Detection in Arrays via Dependency Graphs. *VLSI Signal Processing*, 4(4):331–342, 1992.

[82] Ankur Sharma. Flexible Smart Display with Integrated Graphics Rasterizer Using Single Grain TFTs. Master's thesis, Delft University of Technology, February 2012.

[83] D. Shreiner. *OpenGL Reference Manual, Third Edition, The Official Reference Document to OpenGL, Version 1.2*. Addison-Wesley, 2000.

[84] SPECviewperf 6.1.2, URL: http://www.specbench.org.

[85] W.R. Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1993.

[86] Jacob Ström and Tomas Akenine-Möller. iPACKMAN: High-Quality, Low-Complexity Texture Compression for Mobile Phones. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '05, pages 63–70, New York, NY, USA, 2005. ACM.

[87] Synopsys Inc., URL: http://www.synopsys.com.

[88] V. Tiwari, S. Malik, and A. Wolfe. Power Analysis of Embedded Software: A First Step toward Software Power Minimization. *IEEE Transactions on VLSI Systems*, 2:437–445, December 1994.

[89] V. Tiwari, S. Malik, A. Wolfe, and M. Lee. Instruction Level Power Analysis and Optimization of Software. *Journal of VLSI Signal Processing Systems*, 13(2–3):223–238, 1996.

[90] J. Torborg and J.T. Kajiya. Talisman: Commodity Realtime 3D Graphics for the PC. In *SIGGRAPH 96 Conference Proceedings*, pages 353–364, 1996.

[91] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye. Energy-Driven Integrated Hardware-Software Optimizations Using SimplePower. *ISCA 2000*, 2000.

[92] M.D. Waller, J.P. Ewins, M. White, and P.F. Lister. Efficient Primitive Traversal Using Adaptive Linear Edge Function Algorithms. *Computer & Graphics*, 23:365–375, 1999.

[93] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL Programming Guide, Third Edition, The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley, 1999.

[94] C.-C. Wu and T.-S. Wu. Concurrent Error Correction in Unidirectional Linear Arithmetic Arrays. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 136–141, 1987.

[95] Chang Nian Zhang, Hon Fung Li, and R. Jayakumar. A Systematic Approach for Designing Concurrent Error-Detecting Systolic Arrays Using Redundancy. *Parallel Computing*, 19(7):745–764, 1993.

List of Publications

Patents

1. **D. Crisu**, S.D. Cotofana, S. Vassiliadis, and P. Liuha, "Determining a Coverage Mask for a Pixel", Applicant: Nokia Corporation, *US Patent US* 7006110 *B*2 (Feb. 28, 2006), *European Patent EP*1614071 *A*2 (Nov. 1, 2006), *International Patent WO/*$2004/093012$ (Oct. 28, 2004).

Journal Papers

2. B.H.H. Juurlink, I. Antochi, **D. Crisu**, S. D. Cotofana, and S. Vassiliadis, "GRAAL: A Framework for Low-Power 3D Graphics Accelerators", in *IEEE Computer Graphics and Applications*, July 2008, pp. 63–73.

International Conferences

3. **D. Crisu**, S. D. Cotofana, S. Vassiliadis, and P. Liuha, "3D Graphics Tile-Based Systolic Scan-Conversion", in *The* 38^{th} *Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, USA, November 2004, pp. 517–521.

4. **D. Crisu**, S. D. Cotofana, S. Vassiliadis, and P. Liuha, "Logic-Enhanced Memory for 3D Graphics Tile-Based Rasterizers", in *Proceedings of the 2004 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS 2004)*, Hiroshima, Japan, July 2004, pp. II-237–II-240.

5. **D. Crisu**, S. D. Cotofana, S. Vassiliadis, and P. Liuha, "Efficient Hardware for Antialiasing Coverage Mask Generation", in *Proceedings of Computer Graphics International Conference 2004 (CGI 2004)*, Crete, Greece, June 2004, pp. 257-264.

6. **D. Crisu**, S. Vassiliadis, S. D. Cotofana, and P. Liuha, "Low Cost and Latency Embedded 3D Graphics Reciprocation", in *Proceedings of 2004 IEEE International Symposium on Circuits and Systems (ISCAS 2004)*, Vancouver, Canada, May 2004, pp. II-905–II-908.

7. **D. Crisu**, S. D. Cotofana, S. Vassiliadis, and P. Liuha, "GRAAL - A Development Framework for Embedded Graphics Accelerators", in *Proceedings of Design, Automation and Test in Europe (DATE 04)*, Paris, France, February 2004, pp. 1366–1367.

8. **D. Crisu**, S. D. Cotofana, S. Vassiliadis, and P. Liuha, "High-Level Energy Estimation for ARM-Based SOCs", in *Lecture Notes in Computer Science (Proceedings of the Third International Workshop on Computer Systems: Architectures, Modeling and Simulation SAMOS III)*, Samos, Greece, November 2004, pp. 168–177.

9. **D. Crisu**, "An Architectural Survey and Modelling of Data Cache Memories in Verilog HDL", in *Proceedings of the* 22^{nd} *International Semiconductor Conference CAS '99*, Sinaia, Romania, October 1999, pp. 139–143.

10. S. A. Spanoche, S.-M. Popescu, M. Bodea, **D. Crisu**, C. Gavrilescu, R. Ionita, M. Padure, A. Popa, "Electrical Transient Noise Modelling in SPICE-like Simulators", in *Proceedings of the* 21^{st} *Edition of Control Systems and Computer Science*, Bucharest, Romania, May 1999, pp. 117–120.

11. C. Andreev, R. Ionita, A. Popa, **D. Crisu**, and C. Dan, "βTDA2003 — 10W Monolithic Audio Amplifier", in *Proceedings of the* 21^{st} *International Semiconductor Conference CAS'98*, Sinaia, Romania, October 1998, pp. 481–484.

12. **D. Crisu**, and C. Dan, "An Auto-Scaling Ruler for the L-Edit Layout Editor Implemented Using L-Edit/UPI Subroutine Library", in *Proceedings of the* 21^{st} *International Semiconductor Conference CAS'98*, Sinaia, Romania, October 1998, pp. 493–496.

13. V. Muresan, **D. Crisu**, and X. Wang, "From VHDL to FPGA. A Case Study of a Fuzzy Logic Controller", in *Proceedings of the International Conference for Young Lecturers and PhD Students*, Miskolc, Hungary, August 1997, pp 127–130.

Local Conferences

14. **D. Crisu**, S. D. Cotofana, S. Vassiliadis, and P. Liuha, "Efficient Hardware for Tile-Based Rasterization", in *Proceedings of* 15^{th} *Annual Workshop on Circuits, Systems, and Signal Processing (ProRISC 2004)*, Veldhoven, The Netherlands, November 2004, pp. 352–357.

15. L. Huang, **D. Crisu**, and S. D. Cotofana, "Heuristic Algorithms for Primitive Traversal Acceleration in Tile-Based Rasterizers", in *Proceedings of* 15^{th} *Annual Workshop on Circuits, Systems, and Signal Processing (ProRISC 2004)*, Veldhoven, The Netherlands, November 2004, pp. 408–414.

16. **D. Crisu**, S. D. Cotofana, S. Vassiliadis, and P. Liuha, "Design Tradeoffs for an Embedded OpenGL-Compliant Hardware Rasterizer", in *Proceedings of* 14^{th} *Annual Workshop on Circuits, Systems, and Signal Processing (ProRISC 2003)*, Veldhoven, The Netherlands, November 2003, pp. 49–55.

17. **D. Crisu**, S. D. Cotofana, and S. Vassiliadis, "A Hardware/Software Co-Simulation Environment For Graphics Accelerator Development in ARM-based SOCs", in *Proceedings of the* 13^{th} *Annual Workshop on Circuits, Systems, and Signal Processing (ProRISC 2002)*, Veldhoven, The Netherlands, November 2002, pp. 255–267.

18. **D. Crisu**, S. D. Cotofana, and S. Vassiliadis, "An Energy-Aware Architectural Exploration Tool for ARM-Based SOCs", in *Proceedings of the* 12^{th} *Annual Workshop on Circuits, Systems, and Signal Processing (ProRISC 2001)*, Veldhoven, The Netherlands, November 2001, pp. 327–337.

Technical Reports

19. **D. Crisu**, S.D. Cotofana, and S. Vassiliadis, "A Tile-Based OpenGL-Compliant Hardware Rasterization Engine - Progress Report", in *Deliverable no. (2002)-03, Computer Engineering Laboratory*, Faculty of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology, 2002.

20. **D. Crisu**, S.D. Cotofana, and S. Vassiliadis, "A Proposal of a Tile-Based OpenGLCompliant Rasterization Engine", in *Deliverable no. (2002)-*

02, Computer Engineering Laboratory, Faculty of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology, 2002.

21. **D. Crisu**, S.D. Cotofana, and S. Vassiliadis, "An Energy-Aware Architectural Exploration Tool for ARM-Based SOCs", in *Deliverable no. (2001)-02, Computer Engineering Laboratory*, Faculty of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology, 2001.

22. **D. Crisu**, I. Antochi, S.D. Cotofana, B. Juurlink, and S. Vassiliadis, "Low-Power Techniques and 2D/3D Graphics Architectures", in *Deliverable no. (2001)-01, Computer Engineering Laboratory*, Faculty of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology, 2001.

23. **D. Crisu**, "An Architectural Survey and Modelling of Data Cache Memories", in *Technical Report Contract No. 5076/1999/B90*, Romanian National Agency of Science, Technolgy and Innovation, 1998.

Books

24. B. Mitu, and **D. Crisu**, "Internet şi World Wide Web", Editura Tehnică, Bucureşti, 2000, 188 pages + 1 CD-ROM, ISBN 97-331-1429-4.

Samenvatting

Dit proefschrift presenteert GRAAL (GRAphics AcceLerator) een ontwikkelmethodiek voor het ontwerpen van embedded tile-based rasterisatie hardware voor het versnellen van real-time, 3D grafische (OpenGL) applicaties voor mobiele applicaties. Het doel van GRAAL is om betaalbare en zuinige hardware te kunnen ontwerpen die hoge prestaties levert met een goede beeldkwaliteit. Het onderzoek in dit proefschrift concentreert zich op een aantal belangrijke problemen bij tile-based rasterisatie in hardware, zoals het doorlopen van de driehoeken, anti-aliasing en het sorteren van lijsten met primitieven. Een nieuwe hardware oplossing van een driehoeksalgoritme wordt gepresenteerd, die bestaat uit een systolisch scanconversie subsysteem en een verbeterd geheugen subsysteem. Het resulterende system is in staat om in een zeer voordelig ruimtelijk patroon 4 pixel posities per klokcyclus te produceren. Dit tegen gereduceerd vermogensgebruik en een verhoogde doorvoer van de pixel processing pipelines. Oppervlakte anti-aliasing sampling wordt verkregen door gebruik te maken van een algoritme voor pixel-dekkende maskergeneratie, dat de kosten reduceert door tijdens de berekening van de maskerdekking gebruik te maken van de kwadranten symmetrie eigenschap. Het resultaat is een ontwerp dat tegen aanzienlijk gereduceerde kosten een verdubbelde beeldkwaliteit oplevert in vergelijking met de huidige implementaties. Verder wordt een nieuw en efficient hardware lijst sorteringsalgoritme gepresenteerd, dat in de eerste fase van het rasterisatieproces de host processor in staat stelt om met gereduceerde inspanning de sorteerlijst van tiles te produceren en tevens het externe geheugenverkeer verminderd. Voor een implementatie van het Graal-Bench testprogramma werd bij een implementatieomvang die gelijk is aan een 8KB SRAM geheugen macro, het aantal benodigde instructies met een factor 4–$9\times$ verkleind en de geheugenkosten met een factor 3–$6\times$ verminderd. Naar schatting kan een GRAAL ontwerp, met een klokfrequentie van 200Mhz, een rendering en fill rate van 2.4 miljoen driehoeken/s en 460 miljoen pixels/s voor doorsnee 3D grafische scenes bereiken.

Curriculum Vitae

Dan CRIŞU was born in Bucharest, Romania, on August 3, 1971. He attended courses at the Faculty of Electronics and Telecommunications, University "Politehnica" of Bucharest, Romania. He obtained the *Electrical Engineer* BSc. degree in the Microelectronics specialization in 1998, followed by the MSc. degree in the same specialization in 1999.

During 1998–2000, he held the Teaching Assistant position at the Department of Devices, Circuits, and Electronic Instrumentation, Faculty of Electronics and Telecommunications, University "Politehnica" of Bucharest, Romania. His activity was focussed on teaching and offering seminars for introductory courses in Data Acquisition Systems and Programming Languages. He also performed consulting work for O2Micro, USA, mainly designing analog integrated circuits for power supply mangement in laptop computers.

In 2001, he joined the Electrical Engineering, Mathematics, and Computer Science Department, Delft University of Technology, Delft, The Netherlands, where he carried out a PhD stage with the Computer Engineering group under the supervision of dr. Sorin Coţofană. His research activity was supported by a doctoral grant from Nokia Research Centre, Finland. The outcome of this work is presented in this dissertation.

Since 2005, he has been with PowerVR, Imagination Technologies, UK, as a Senior Design Engineer. His role is the design and simulation of cutting-edge embedded 3-D graphics IP cores, many of which have found their way into iconic products.

His research interests include computer architecture, computer arithmetic, low power circuits, and 3-D graphics algorithms and hardware.