Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2010

MSc THESIS

Exploring Tile-Based Rasterization Alternatives for Mobile Devices

M.C.A. van der Weide

CE-MS-2010-19

Abstract

Tile-based rasterization was recently proposed as a solution for enabling three dimensional computer graphics on low-power mobile devices. In tile-based rasterization the screen is decomposed into independent parts, called tiles, that are processed sequentially by a simple embedded graphics accelerator. By keeping these tiles small, on-chip memory can be used to store intermediate rasterization values. This reduces a major source of power dissipation when compared to a conventional graphics pipeline, the external traffic between the rasterization hardware and the framebuffer.
In the GRAphics AcceLerator (GRAAL) project a tile-based rasterization solution utilizing a scenebuffer resident in external memory was proposed. The scenebuffer contains the rasterization instructions needed to process all the screen tiles and is reused for the rasterization of every tile. Several sorting algorithms were introduced to prevent unnecessary accesses to the graphics accelerator.
In this thesis we first argue that there is not enough available hardware budget to implement this sorting in hardware and that performing the sorting in software requires serious effort from the embedded microprocessor. Therefore, we propose two alternative tile-based rasterization solutions for mobile devices. First, direct rasterization, were the embedded accelerator is used directly to process a number of locally stored rasterization instructions for all tiles. Experimental results indicate that the external traffic is 80% when compared to non tile-based rasterization, but unfortunately scenebuffer rasterization results in only 25% of the external traffic of non tile-based rasterization. Second, hierarchical rasterization, a combination of scenebuffer and direct rasterization. Experimental results indicate that there is a wide range of system topologies that outperform scenebuffer rasterization from an external traffic point of view (up to 22% reduction over scenebuffer) and from a software workload point of view (up to 57.8% reduction over scenebuffer). More importantly, some topologies reduce both the external traffic and the software workload when compared to scenebuffer rasterization even some topologies assuming a smaller hardware budget.

Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Exploring Tile-Based Rasterization Alternatives for Mobile Devices

GRAAL revisited

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

M.C.A. van der Weide
born in Hellevoetsluis, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Exploring Tile-Based Rasterization Alternatives for Mobile Devices

by M.C.A. van der Weide

Abstract

Tile-based rasterization was recently proposed as a solution for enabling three dimensional computer graphics on low-power mobile devices. In tile-based rasterization the screen is decomposed into independent parts, called tiles, that are processed sequentially by a simple embedded graphics accelerator. By keeping these tiles small, on-chip memory can be used to store intermediate rasterization values. This reduces a major source of power dissipation when compared to a conventional graphics pipeline, the external traffic between the rasterization hardware and the framebuffer.

In the GRAphics AcceLerator (GRAAL) project a tile-based rasterization solution utilizing a scenebuffer resident in external memory was proposed. The scenebuffer contains the rasterization instructions needed to process all the screen tiles and is reused for the rasterization of every tile. Several sorting algorithms were introduced to prevent unnecessary accesses to the graphics accelerator.

In this thesis we first argue that there is not enough available hardware budget to implement this sorting in hardware and that performing the sorting in software requires serious effort from the embedded microprocessor. Therefore, we propose two alternative tile-based rasterization solutions for mobile devices. First, direct rasterization, were the embedded accelerator is used directly to process a number of locally stored rasterization instructions for all tiles. Experimental results indicate that the external traffic is 80% when compared to non tile-based rasterization, but unfortunately scenebuffer rasterization results in only 25% of the external traffic of non tile-based rasterization. Second, hierarchical rasterization, a combination of scenebuffer and direct rasterization. Experimental results indicate that there is a wide range of system topologies that outperform scenebuffer rasterization from an external traffic point of view (up to 22% reduction over scenebuffer) and from a software workload point of view (up to 57.8% reduction over scenebuffer). More importantly, some topologies reduce both the external traffic and the software workload when compared to scenebuffer rasterization even some topologies assuming a smaller hardware budget.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2010-19

Committee Members :

Advisor:	Sorin Cotofana, CE, TU Delft
Member:	R. van Leuken, CE, TU Delft
Member:	S. Wong, CE, TU Delft
Member:	S. Cotofana, CE, TU Delft
Member:	K. Bertels, CE, TU Delft

Contents

List of Figures

List of Tables

Acknowledgements

Great thanks goes out to my advisor for giving me the room, time, and space to determine my own path in this research. Although this path generally did not took me from point A to B in a straight line it thought me a great many things about myself and life in general. I am sure the path will keep winding and provide me with new challenges along every step.

Further thanks is due to my family and friends for always being there for support and provide me with a life with excitement and joy. Eventhough, this more then once distracted me from completing the work currently under viewers eye.

No thesis, or any other accomplishment for that matter, however small it may be, can ever be achieved without the aid of the most precious and loving people in the world,

Special thanks to my parents.

M.C.A. van der Weide
Delft, The Netherlands
July 2, 2010

Introduction 1

The field of computer graphics is a continuously growing and commercially attractive market. In [12] the results of a study in worldwide revenue in the entertainment sector were presented. This economic study indicates that the video gaming industry is 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 worldwide revenue in the video game industry in 2009 was estimated to be approximately $55 billion. More importantly, this study shows a considerable increase in the total revenue for wireless gaming on mobile terminals growing from $281 million in 2004 to $2.1 billion in 2009. This evaluated to a 49.3 % annual increase. By now these figures have been materialized by two major players that are involved in mobile devices; sony and nintendo, of which the former has sold 60 million as of march 2010 [35] devices (PSP) and the later has sold 128 million as of march 2010 [34] devices (Nintendo DS). These figures clearly show an increasing market for mobile gaming platforms.

This commercial interest in mobile gaming has triggered the introduction of a new field in research that provides for 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 and assembly cost are two important additional design criteria used to evaluate possible system solutions. One cannot simply suffice with a conventional approach to implementing computer graphics on a mobile device.

Driven by the economically-powerfull entertainment industry the research in mobile graphics proves to be attractive and implementing graphics acceleration on mobile devices is a field of particular interest. As a result a widespread industry standard for computer graphics, called OpenGL [32], has now a mobile counterpart in the form of OpenGL ES [24]. Moreover, ARM Ltd, a leading supplier of embedded Intellectual Property (IP) cores, has recently started offering a range of embedded solutions for computer graphics [9] alongside their embedded microprocessors.

To reduce power consumption on mobile devices one can often resort to System-on-Chip (SoC) design. In this design methodology several system components are combined onto a single chip to reduce the distance between components, resulting in increased performance and faster data communications. By combining several system functions on a single chip assembly costs are reduced and a final additional benefit for SoC design is reduced power consumption due to the external traffic elimination.

External traffic is a major source of power dissipation [21] mainly induced by the high capacitance of printed circuit board connections when compared to connections on a chip. A large factor in this higher capacitance is the length of the connections. By implementing several system modules on a single chip communication between these modules is done via a small on-chip connection, usually a bus, instead of a longer off-chip connection. Examples of a on-chip bus is the AMBA system bus [28] especially created for use in conjunction with ARM processing cores.

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, which is the desired image to display. This memory is read pixel by pixel 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 created from triangles.

During the last stage of the graphics process, the rasterization stage, the abovementioned 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 can ultimately be blended with a new color in order to generate a final color value. The key notion to observe in this process is the fact that the framebuffer is also frequently accessed to retrieve previously computed color (and especially depth) values during rasterization.

Unfortunately, even for small displays as widely used in mobile devices, a large amount of data is required to be stored in the framebuffer. The physical hardware required to store this amount of data can easily exceed the hardware budget available for mobile graphics on a SoC. For example, a 640x480 size display already requires a framebuffer of over 1 Megabyte. In DRAM memory one storage cell costs the equivalent of approximately one gate [22]. A framebuffer of 1 Megabyte implemented in DRAM on-chip would therefore require around 8 million gates. This can easily exceed the typical hardware budget for mobile graphics.

A DRAM memory cell is created with a single transistor and a trenched or stacked capacitor, in order to create such a capacitor on chip extra fabrication steps are required during the processing of the wafer. This means the production process to make a chip which contains DRAM memory is more costly that a normal fabrication process where only transistors are created. However, a normal fabrication process only allows the construction of SRAM memory cells, which require 6 gates per cell [22], resulting in an even larger memory area if the framebuffer was implemented on the SoC. For mobile devices, the framebuffer is therefore considered to be too large to be implemented on the SoC and is assumed to be implemented on an external memory chip.

Thus, the problem with implementing a conventional computer graphics solution on mobile devices is characterized by the fact that the framebuffer is too large to be implemented on-chip and that the access rate to the framebuffer during rasterization is high. This results in large external traffic between the graphics hardware and the framebuffer during rasterization, which in turn results in high power consumption for the total system. This is undesirable for mobile devices because these are powered with a limited power supply in the form of batteries. The key to mobile graphics is therefore to reduce the amount of external traffic during rasterization in order to result in low power dissipation.

1.1 Tile-Based Rasterization on mobile devices

Instead of using a traditional rasterization solution, where triangles are processed for the entire screen at a time, tile-based rasterization, where triangles are processed only for a part of the screen at a time, has been considered for implementation on mobile devices [16]. In tile-based rasterization, originally used in high performance parallel rasterization, the screen is divided into small sections, called tiles. All the rasterization insructions that compose a scene, mainly triangle instructions, are duplicated for all the tiles they belong to, which allows all the tiles to be processed independently.

In high performance rasterization tiles are processed in parallel by multiple rasterizers and the total screen is processed faster when compared to a fullscreen rasterization scenario. A single rasterizer used in tile-based rasterization can suffice with a small local framebuffer and the large global framebuffer is constructed by combining the final color values of the smaller local framebuffers. Researchers in the same field have proposed similar approaches under the name of bucket- [13] or chunk- [10] rendering.

The same concept is used for mobile graphics to decompose the screen, however instead of using several rasterizers in parallel all the tiles are processed by the same hardware accelerator serially. 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 microprocessor and 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 in order 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.

Previous research [16] proposed a framework for developing tile-based rasterization hardware for mobile devices. Within this framework a novel design for an embedded tile-based rasterizer called GRAphics AcceLerator (GRAAL) was proposed. GRAAL is an OpenGL compliant rasterizer for use in a tile-based rasterization scenario. Within this work design tradeoffs have been researched [19] and intelligent hardware solutions were proposed [17][14][15]. A brief overview of the rasterizers implementation is given in this thesis. However, our work is not limited to this exact rasterizer, it can be extended in general to any tile-based rasterizer.

Parallel to the work performed to find intelligent hardware solutions for the implementation of GRAAL, research was done on the sorting performed prior to rasterization of the triangles on the GRAAL hardware [2]. This research assumed that the sorting is performed per frame, a single image to be displayed on the screen. During this sorting all triangles in a frame are stored in a region of external memory called the scenebuffer. The scenebuffer allows the tiles to be processed serially, where each tile is only processed once. As a result GRAAL can discard any intermediate parameter values required during rasterization of a tile, namely depth and stencil values. Only the final color values necessary for visualization on a screen are stored into the external framebuffer. This approach reduces the traffic from GRAAL to the external framebuffer during the rasterization of presorted triangles.

Several algorithms were proposed to create the scenebuffer [8]. These algorithms can be implemented in software running on the microprocessor or can alternatively be implemented in hardware if there is enough hardware budget available. The main disadvantage of all algorithms is the increase of traffic to the external memory in order to create the scenebuffer. The amount of traffic generated during sorting is dependent on the number of tiles and thus on the tile size. It was found that an optimal tile size with respect to the traffic to the external memory is 32×32 pixels [7].

From here we will refer to this rasterization concept proposed in the previous work as scenebuffer (tile-based) rasterization.

1.2 Research Objectives & Contributions

In the previous research [2] an assumption was made with respect to the available hardware budget for graphics acceleration inside the SoC. The budget was based on estimations by ARM that additional hardware available for mobile graphics was limited to 200k to 500k gates. Spending a larger hardware budget on graphics hardware can make the resulting chip too expensive and the entire mobile devices would be too expensive.

We note in this thesis that a tile-based rasterizer of $32{\times}32$ pixels consumes a large part of the available hardware budget for the tilebuffer memory array alone. This leaves no hardware budget for the scenebuffer sorting steps. This means that the scenebuffer sorting is to be performed by software running on the microprocessor. This increase of software workload reduces the generation rate of triangles by the software and can diminish the overall system performance and is thus an undesirable effect of the previously proposed tile-based rasterization solution.

In this thesis we focus on a new rasterization concept that is aimed to reduce the sorting workload on the software side of the system running on a general purpose processor. Note that this microprocessor is the core of the chip and it is not solely reserved for graphical computations and should therefore not be strained by graphics computations. Our objectives for this research were the following:

- Explore a new tile-based rasterization concept were sorting is performed in hardware prior to rasterization. In this rasterization concept the notion is dropped that a frame must be fully sorted before rasterization can start. By doing so it takes a middle path between traditional rasterization where the process is performed for a single primitive and scenebuffer rasterization where the rasterization is essentially performed per frame. Intuitively speaking, when compared to scenebuffer rasterization the software workload is reduced, however we introduce additional external traffic when compared to scenebuffer rasterization.

- Determine the feasibility of this new tile-based rasterization solution. We aim to determine the hardware modifications required to the existing tile-based architecture in order to create a simple sorting scheme in hardware to reduce the software workload on the microprocessor.
- Develop a framework for determining the amount of external traffic during the rasterization in this new tile-based rasterization concept. We use the amount of external traffic as an indication of the amount of consumed power since external traffic is the major source of power dissipation.
- Within this framework we investigate certain important parameters, such as the tile size and the chosen sorting algorithm, and determine their effect on the total amount of external traffic. With the identification of the most important parameters the rasterization concept can be optimized.
- Finally, and most importantly, we compare the external traffic figures of the different rasterization concepts acquired by simulating benchmarks from a mobile benchmark suite [6]. By comparing the traffic figures of traditional, scenebuffer, and our new rasterization concept we can determine the merit of our new rasterization concept.

In order to reduce the workload on the microprocessor we wish to move the sorting from software to hardware. This software was required in order to utilize a tile-based rasterizer. However, storing all triangles in a frame in hardware cannot be done without exceeding the hardware budget. Therefore we explore a tile-based rasterization concept were the sorting is not performed on all the triangles in a frame, but on a limited amount of triangles that are stored on chip.

Note that by sorting only a number of triangles of a frame instead of all the triangles in a frame we can no longer discard intermediate rasterization values, like color and depth values, from the rasterization hardware before the end of the frame is encountered. Therefore when a tile is processed, based on the triangles currently available in the hardware buffer, all temporary rasterization values need to be stored into the framebuffer in the external memory. At the time of the next processing visit to the same tile these values have to be retrieved from the framebuffer. This, off course, results in increased external traffic when compared to scenebuffer rasterization. Clearly this increased external traffic is problematic and as such should be minimized as much as possible. We address this problem critically in this thesis.

Our contributions to the field of mobile graphics are the following:

- We present direct (tile-based) rasterization scenario as a new approach for low power rasterization on mobile devices. Given that the sorting that was previously done in software is now performed in hardware a limited number of rasterization primitives need to be stored on chip. The so called direct-sorting unit, a hardware component functioning as a interface between the microprocessor and the graphical processor, is providing this service and it is mainly composed of memory to store the triangle data.

We present simple hardware modifications to the memory arrays inside a tile-based rasterizer that prevent unnecessary reads and writes to the external memory. Instead of retrieving the entire tiles color, depth, and stencil values we only retrieve the necessary values during rasterization. By preventing these external memory transactions the external traffic, and in direct relation the power dissipation, is increased less dramatically.

- We observe that scenebuffer and direct rasterization are not mutually exclusive in a system solution. Therefore we can combine direct sorting in hardware and scenebuffer sorting in software into a single rasterization solution for mobile graphics, where the sorting workload can be easily balanced between hardware and software. We refer to this approach as hierarchical (tile-based) rasterization and it is ideal for reducing the complexity for both sorting concepts. The scenebuffer sorting algorithm can sort the incoming triangles to a smaller amount of bins, reducing the sorting workload in software, while the direct-sorting algorithm sorts a lower amount of tiles, reducing the required hardware needed to maintain triangle to tile overlap data and increasing the number of triangles stored on chip.

  Introducing hierarchical rasterization vastly increases the design space for a system engineer implementing mobile graphics. In embedded microprocessors were the microprocessor is heavily strained the system engineer can choose to restrict the scenebuffer sorting to 20 sections instead of 300 sections as in scenebuffer rasterization. The rest of the sorting required for tile-based rasterization is then performed in hardware in the direct-sorting unit, greatly reducing the strain on the microprocessor.

  The biggest strength of hierarchical rasterization is the introduced possibility to balance the workload between hardware and software. If performance of a certain system is low moving workload from the microprocessor to hardware can be necessary to improve the performance on the microprocessor. If in this case the hardware budget is not increased the price is paid with increased external traffic resulting in a higher power consumption. If a higher power consumption is unwanted the hardware budget should be increased. Likewise, performance can be sacrificed to reduce power consumption. Another option might be to maintain a relatively high software workload and reduce the hardware budget for the tile-base rasterizer which results in smaller and therefore cheaper more competitive chips due to higher yields (we present an example of this case in our result section). The possibilities are numerous and provide a welcome addition to tile-based rasterization.

In order to compare different rasterization concepts from a external traffic point of view we created a high-level simulator that calculates traffic figures, in total byte accesses to the frame- and scenebuffer in external memory. The simulator can calculate the external traffic for a number of frames in a given testbench suite in a matter of minutes.

When compared to the simulator provided by previous work, a register-level SystemC simulator, this is much faster because the provided simulator would take several hours just to rasterize one frame. In addition the simulator is highly configurable; therefore one can easily perform a exhaustive search of different system topologies in order to find an optimal solution.

We utilized the created simulator to explore the effects of several design considerations. We ran simulations on fullscreen, scenebuffer, direct, and hierarchical rasterization implementations. Each rasterization implementation was run several times while varying important design options. These simulations give us results that indicate the best choices for some important design considerations. Some important results we found are the following:

- Results indicate that scenebuffer sorting algorithm based on sorting the scenebuffer to bins as proposed in [8] results in the lowest amount of external traffic and is less dependent on the number of tiles as the other sorting algorithms from previous work. This does not deviate from the results found from previous work. The overall tendency in scenebuffer rasterization is that reducing the number of tiles, achieved by increasing the size of the tilebuffer inside the tile-based rasterizer, results in less external traffic.

- In contrast, results obtained while simulating our proposed direct tile-based rasterization scenario suggest that the total traffic to the external framebuffer is reduced when decreasing the tile size. By choosing a smaller tile size the hardware buffer inside the tile-based rasterizer, containing intermediate pixels values for the current tile, can be reduced and the hardware budget is lowered. Unfortunately reducing the tile size increases the number of tiles in the screen and the memory array of the direct-sorting unit increases cancelling the budget gain from the tilebuffers. This clearly indicates that various tradeoffs are possible.

- We present and compare simulation results for four sorting heuristics with varying degrees of complexity, which can easily be implemented in hadware inside the direct-sorting unit. The goal of these heuristics is to choose an appropriate tile to process based on the triangles currently stored inside the direct-sorting unit. Results indicate that a selection policy aimed at selecting tiles that hold small triangles is the best heuristic for minimizing the amount of traffic between the tile-based rasterizer and the external framebuffer. Intuitively speaking this can be justified as follows: (i) Making sure the smallest triangles, with a low amount of overlap to different tiles, can be discarded from the sorting unit quickly creates room for new triangles. (ii) Introducing new triangles as fast as possible reduces the number of tile switches and consequently reduces the external traffic.

- On average the total external traffic in the direct rasterization scenario is larger then in the scenebuffer rasterization scenario. Averaged over the set of chosen benchmarks the direct rasterization reduces the total external traffic to 80 % when compared with a fullscreen rasterization system. While the experimental results of a scenebuffer rasterization scenario suggests reduction of the total external traffic to 25 % when compared to a fullscreen rasterization. Nevertheless, the direct rasterization scenario can prove usefull for mobile graphics since it alleviates the workload on the software running on the microprocessor and works well for small tile sizes.

- The most promising results where obtained during the simulation of a hierarchical rasterization scenario. Our results suggest that hierarchical rasterization is able to reduce the external traffic when compared to scenebuffer rasterization, while reducing the software workload on the microprocessor related to sorting triangles to bins. Our results indicate that the external traffic is reduced to 90.8 % when compared to scenebuffer rasterization when sorting triangles to only 20 sections instead of 300 sections as in scenebuffer rasterization, additionally resulting in a 42.3 % software workload reduction. The external traffic in a hierarchical rasterization approach can even by reduced to approximately 78% when compared to scenebuffer rasterization when sorting to 60 or 80 sections.

- We also present external traffic figures that suggest that the external traffic in a hierarchical rasterization scenario can be less (93%) then the external traffic in a scenebuffer rasterization scenario even when less hardware (50%) when compared to scenebuffer rasterization is used to construct the direct-sorting unit and the tile-based rasterizer. Even in this case the amount of software workload is considerably reduced when compared to the workload in a scenebuffer rasterization scenario.

1.3 Overview

We start in Chapter 2 by presenting issues relevant for implementing computer graphics on mobile devices. We discuss the design criteria to be considered in mobile graphics and argue why a fullscreen rasterization scenario, as used in most comtemporary graphics devices, is not desirable for a mobile device. We provide some background on tile-based rasterization and present a small overview of the previous work.

Following, in Chapter 3 we start with an analysis of the previous work and then present the direct sorting scenario that moves the triangle sorting from the software side of the system to a unit implemented in hardware. We discuss the important aspects in direct rasterization, namely the tile selection policy, determining and representing the triangle to tile overlap, and the number of triangles stored in the hardware. We present hardware modifications to GRAAL that reduces external traffic. We conclude the chapter by presenting a hierarchical approach that uses both software sorting to a scenebuffer and direct hardware sorting, the resulting hierarchical system can be easily optimized for varying system requirements.

Next, Chapter 4 discusses the simulation platform in which we performed our experiments. We start by listing the benchmark elements used to evaluate various graphics pipeline configurations under realistic graphics workloads for mobile devices. Since our work is mainly targeted on the rasterization process we simply simulate the first stages of the graphic pipeline with a modified version of an existing graphics solution and work with an instruction stream as input for the rasterization stage. Our main focus lays on the traffic to the external memory. The computation of this traffic is split in two parts, traffic related to sorting and traffic related to storing and retrieving fragment values.

Chapter 5 presents the results obtained from our experiments. We first ran experiments on a single rasterization scenario while varying certain design tradeoffs in order to find the best solution for a system implementation. Afterwards we compare the varying rasterization scenarios on a traffic point of view. We conclude this chapter with an attempt to system level comparison between a hierarchical rasterization scenario and a pure scenebuffer rasterization scenario with interesting results.

In Chapter 6 we provide a short summary of this report, present our main contributions to the field of mobile graphics, and provide some additional research topics for future work. Finally, Appendix A is added to provide a background on computer graphics and the graphics pipeline for readers unfamiliar with the concepts.

Mobile Graphics 2

The computations used in traditional computer graphics are no different for mobile devices. For the readers unfamiliar with the computations in computer graphics we included a background on this topic in Appendix A. The design constraints relevant for mobile devices, which are described in Section 2.1, necessitate however the search for new hardware acceleration solutions. The reasons why the conventional approach of hardware acceleration for the rasterization stage, here referred to as fullscreen rasterization, are not optimal for mobile devices is discussed in Section 2.2. A promising rasterization alternative for mobile devices is tile-based rasterization, detailed in Section 2.3. Previous work on tile-based rasterization scenario is presented in Section 2.5, the special rasterization hardware it requires is discussed in Section 2.4.

2.1 Design Criteria

The field of mobile graphics is a special subset of traditional computer graphics with its own set of design considerations. The actual computations needed are equal to traditional computer graphics, but the system implementations are subject to different design constraints. In traditional computer graphics, systems were designed following a brute force approach with a high level of parallelism. The main design issues are induced by the high resolutions and the high performance required for an optimal user experience. As a result, contemporary graphics pipelines are optimized to accommodate high resolutions and high throughput and therefore exhibit massive energy consumption.

Mobile graphics, however, have to deal with issues related to the mobility of the resulting products as well as the user experience. This means the products are smaller, with lower resolution displays, are used closer to the eyes of the user and, most importantly, are powered by a limited power supply in the form of batteries.

The maximum resolution typical in contemporary mobile devices is around 640×480 pixels and the size of the screen is around 3 by 3 inches. But this is expected to substancially increase for future mobile devices. Due to these small size displays mobile devices are used at a short distance to the user eye. Because the user is very close to the screen the area of projection for a single pixel onto the users eye is actually larger then for a personal computer [1]. This means that every pixel must be rendered at a high quality and implies no hardware can be saved by reducing the color depth, i.e., the amount of bits used to represent color values.

Mobile devices are implemented with a special mobile microprocessor. These processors are optimized for low-power and only support a simple instruction set. Furthermore, these microprocessors often operate at much lower frequencies than desktop computers. Therefore implementing the graphical computations on a mobile microprocessor alone, without hardware acceleration, is not feasible while still providing adequate performance.

To implement low power computer graphics a lot of research in the field of mobile graphics was started in industry and at different universities. One of the possible steps to reduce power consumption in the graphics system is to implement parts of the graphics pipeline in a System-on-Chip (SoC) design. In SoC design the different hardware units that make up a system are localized onto a single chip to increase performance and to reduce communication overhead.

For example, the microprocessor and the graphics accelerator could be implemented on the same chip in a SoC design. This reduces the distance between these components and thus removes the need for power- and time-consuming communication between the components, which would have been required when both components were on separate chips. Therefore the overall power consumption is reduced and the system performance can be increased.

2.2 Problems with Conventional Rasterization

The straightforward approach to creating a computer graphics SoC is to move the hardware of a conventional graphics accelerator onto the same chip as the microprocessor. However, directly moving the rasterization process onto the microprocessor chip as it is usually implemented in conventional graphics systems does not result in a very effective system.

Because the conventional hardware rasterization scenario performs the rasterization of each and every single triangle for the entire screen at once, we call this rasterization approach a fullscreen rasterization scenario. If the conventional graphics pipeline, where every rasterization primitive (mostly triangles) are directly processed and its effects on the image are directly stored to the framebuffer, is ported to a SoC we obtain a system topology as depicted in Figure 2.1.

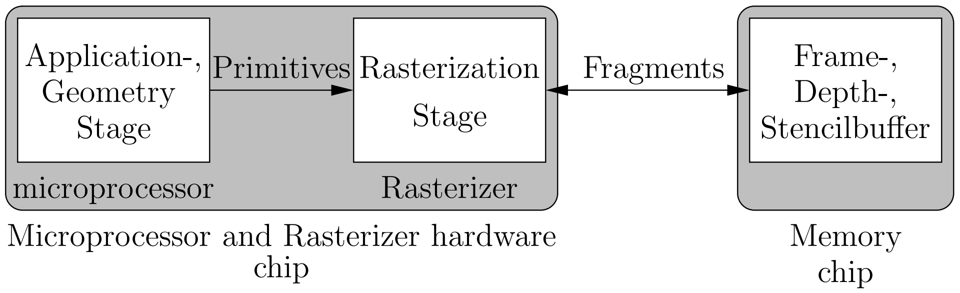

Application-, Geometry Stage
Primitives
Rasterization Stage
Fragments
Frame-, Depth-, Stencilbuffer
microprocessor
Rasterizer
Microprocessor and Rasterizer hardware chip
Memory chip

Figure 2.1: Conventional fullscreen rasterization organization.

The framebuffer, a memory region that holds the color information for every pixel in the screen is to large to be stored on chip and is therefore located on a separate memory chip. For simplicity we assume that the application stage and geometry stage (See Appendix A for details) are both performed in software running on an embedded microprocessor, since our main focus is the hardware acceleration of the computationally intensive rasterization stage.

During rasterization temporary pixel values, such as stencil, depth, and color values, are stored in memory for every pixel in the screen. The amount of data that needs to be stored during rasterization is significant. For example, a small 640 × 480 pixel image with 32 bits for color values in RGBA format, 24 bit fixed point depth values, and 8 bits for stencil values already requires a storage capacity of approximately 20 Megabit in order to contain the intermediate parameter values for all the pixels on the screen. Even when assuming that a costly production process is used that allows the creation of embedded DRAM memory, requiring approximately 1.5 gates per stored bit [27][26], the implementation of this memory on-chip exceeds the hardware budget for mobile graphics easily.

During the rasterization stage we move from computations on rasterization primitives to computations on individual fragments. A fragment is a triangles infuence on a single pixel. A triangle that spans several pixels results in an equal amount of generated fragments. For the correct rasterization of these fragments we need a considerable amount of pixel values stored inside the frame-, depth-, and stencilbuffer located on the separate memory chip. Because a triangle can be visible in any part of the screen each pixel value can be accessed at any time. Unfortunately, these off-chip memory accesses are one of the most power consuming operations in a system [21]. The overall total power consumption of a fullscreen rendering approach implemented on a SoC is therefore still very high. This means that a fullscreen rasterization scenario is not an appropriate solution for the rasterization stage in mobile devices.

2.3 Tile-Based Rasterization

The study in low-power solutions that enable computer graphics on mobile devices is a growing field. In this field tile-based rasterization was recently introduced as a promising solution for low-power embedded 3D graphics on mobile devices. Compared to the conventional rasterization solutions, tile-based rasterization is able to significantly reduce the fragment traffic between the rasterizer and the color, depth, and stencil framebuffers. We note in here that tile-based rasterization was originally developed for high-performance parallel rasterization and breaks down the rasterization of an entire screen into smaller sections, called tiles.

The basic working principle behind this rasterization approach is that there are no data dependencies between tiles during rasterization. A pixels color is essentially only dependent on its location and not on the data values of any of the surrounding pixels. Therefore the rasterization of each tile can be done individually and the final fullscreen result can be acquired by simply combining the rasterization result for all separate tiles into a single large image. Additionally, since there are no data dependencies between pixels the rasterization process is highly scalable and there are no limitations to the chosen tile size.

In high performance graphics tile-based rasterization is used to speed up the rasterization of high resolution applications by processing each tile on a separate rasterizer. For example, an application with a resolution of 8000 × 6000 can be broken down and rasterized by 100 simple hardware accelerators with a resolution of 800 × 600 pixels. The hardware in these individual rasterizers is less complex then a single 8000 × 6000 rasterizer and each rasterizer individually requires less memory to store the frame-, depth-, and stencilbuffers. Smaller memory is often faster to access and consumes less power per access.

In high performance parallel rendering, consider the case were the rasterization of an image is done by four separate rasterizers A, B, C, and D. Figure 2.2 illustrates the effort in time for each of the separate rasterizers during the tile-based rasterization of the image on the right. The triangle density in the four parts of the screen is different and hence the required effort for each rasterizer is different. However, because the workload is distributed among four rasterizers the total time needed for rasterization is reduced when compared to the time required by a single rasterizer. After a certain period of time denoted by T, the time required for the rasterization of the densest part of the screen, all rasterizers are done processing and the final image is complete.

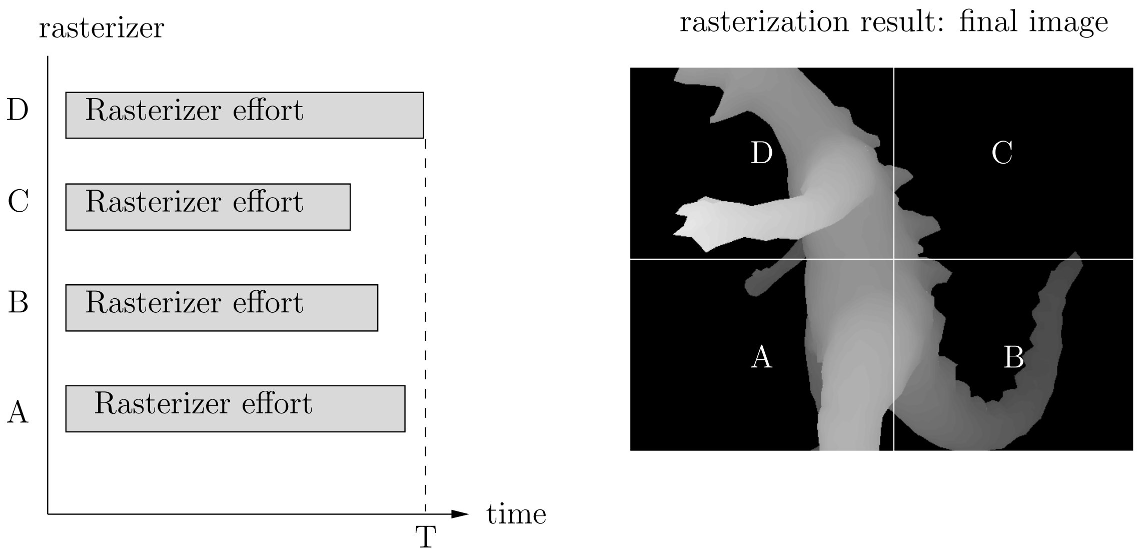


Figure 2.2: Effort versus time in Tile-based Rasterization with four Parallel Rasterizers.

In mobile devices tile-based rasterization can be used as a low-power solution for rasterization. The tile size is scaled down sufficiently in order to store the temporary stencil, depth, and color values needed to rasterize a tile on the same chip as the rasterization hardware. This means that all values required during the rasterization of a tile are stored locally and the performance of the rasterization hardware on chip is increased due to lower latencies while retrieving and storing fragment values. The rasterization of the entire screen is performed one tile at a time for each and every screen tile on the same rasterization hardware.

Figure 2.3 illustrates the rasterization effort in time for the tile-based rasterization of the same image as before, but now performed with only a single rasterizer. As expected because less hardware is used, the total time it takes to process the screen is longer. However, the main advantage of tile-based rasterization is that during rasterization the temporary fragment values are stored in a small on-chip memory instead of a large off-chip memory. The hardware rasterizer on the SoC features memory containing the

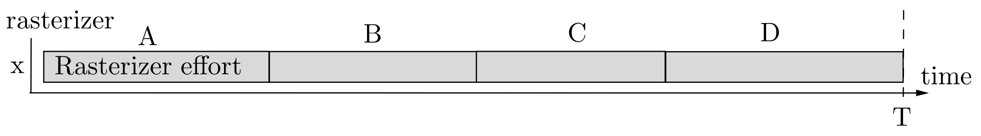


Figure 2.3: Effort versus time in Tile-based rasterization with a single Rasterizer.

stencil, depth, and color values of the pixels in the current tile. Each tile is processed once and during the rasterization no fragments values are required from the external depth-, stencil, or framebuffers. By rasterizing the tiles one by one the external memory accesses are reduced to the storing of final color values only. This reduces the power consumption related to fragment traffic significantly when compared to a fullscreen rasterization scenario.

Unfortunately, a single triangle instruction can influence any number of tiles, sometimes even the entire screen, and can therefore not be discarded until all the tiles influenced by it completely rasterized. This is the main disadvantage of tile-based rasterization for mobile devices. Due to the small available hardware budget tiles can only be processed one at a time. The instructions needed to correctly rasterize the screen have to be stored somewhere until the final image is completely rasterized.

The following sections present in more detail relevant tile-base rasterization aspects as follows: Section 2.4 provides background on previous work done in designing the hardware required for a tile-based rasterizer. Section 2.5 presents information on the previous work done on sorting the instructions of a frame.

2.4 Tile-Based Rasterization Hardware: GRAAL

The tile-based rasterization scenario requires a special tile-based rasterizer. Previous work [16] presented a framework for designing graphics hardware for mobile devices based on the tile-based rasterization concept. In this work hardware was designed that can be used to construct a tile-based rasterization unit on a mobile SoC. The framework is called GRAphics AcceLerator (GRAAL) and for simplicity we refer to the hardware unit designed in that framework with the same name.

Like a contemporary graphics system, as presented in Section A.2, is constructed with several pipeline units, GRAAL is similarly constructed with hardware units forming a pipeline and is fully compliant to the OpenGL Graphics API [32]. However, since GRAAL only performs the rasterization of a small tile, instead of the entire screen, the intermediate parameter values are stored locally in special memory buffers. This means no external traffic related to fragment values required during the rasterization of a tile and the associated power consumption with that traffic is removed from the system.

The pipeline structure within GRAAL is given in Figure 2.4. The first stage of the pipeline determines the overlap of the incoming triangles to the current tile. Pixels that are identified as inside a triangle are passed to the second stage where the fragment parameters are interpolated based on the pixel location and parameter values corresponding to the triangles vertices. After the second unit the fragments are subjected to a selec-

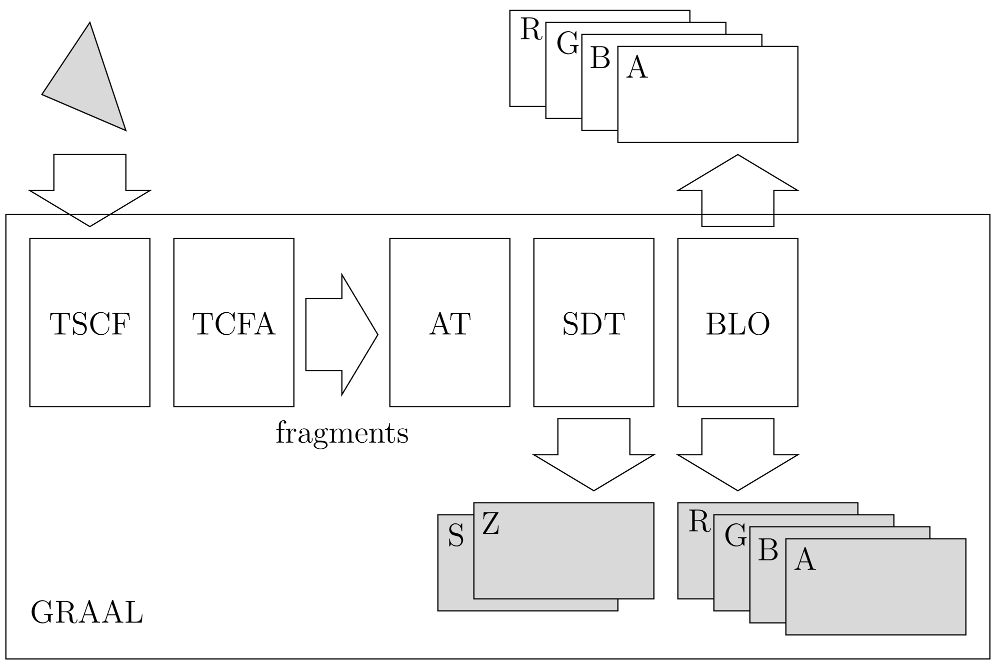


Figure 2.4: GRAAL pipeline with internal buffers for the stencil, depth, and color values.

tion of the OpenGL graphics tests that are important for mobile devices. Some of the OpenGL functionalities, like fog and dithering have not been implemented yet. Based on the outcome of the OpenGL tests updates are made to tile sized stencil,- depth-, and framebuffers located within the GRAAL hardware. The entire rasterization pipeline of GRAAL consists of five consecutive stages:

- Triangle Scan Convert to Fragments (TSCF) stage,
- Triangle Compute Fragments Attributes (TCFA) stage,
- Alpha Testing (AT) stage,
- Stencil and Depth Testing (SDT) stage,
- Blending and Logical Operation (BLO) stage.

In the first (TSCF) pipeline stage the incoming triangles are converted into fragments via the linear edge Equations (2.9), (2.10), and (2.11). For an 8×8 pixel window the edge function results are systolically computed in three clockcycles [14]. The result, the indication that pixels are outside or inside the current triangle, is stored in a special logic-enhanced memory [17] the size of a tile. After this process the logic-enhanced memory contains a stencil pattern of the triangle for the current tile, ones indicating that the corresponding pixel is inside the triangle and zeroes indicating that the corresponding pixel is outside the current triangle. The logic-enhanced memory simplifies passing pixel locations inside the triangle to the next pipeline stage.

The following (TCFA) stage queries the logic-enhanced memory in the first stage for the fragments inside the current triangle. Subsequently, this stage determines the parameter values for the fragment by linearly interpolating the depth (z) and the color values (r, g, b, and a). When texture mapping is enabled the parameters needed for perspective correct texture mapping (s/w, t/w, and $1/w$) are also interpolated as mentioned in Section A.1.5.

Interpolation for all the parameters is done via Equation (A.15), where p_{init}, dp_i/dx, and dp_i/dy for every parameter are calculated only once for an initial fragment within the triangle. All following fragments are computed with an iterative calculation based on the initial fragment from the triangle. The divisions in this stage are performed by a clever reciprocation hardware implementation [19] that trades precision for speed without visually reducing the image quality.

When texture mapping is enabled this unit is also responsible for texture mapping. The texture parameters s/w and t/w are divided by the $1/w$ parameter in order to acquire the hyperbolically interpolated values. The perspective correct parameters s and t are used to determine which texels to fetch from the texture memory in external memory. The fragments are generated by the logic-enhanced memory in a predetermined pattern that increases the hit ratio in a texture-pull architecture with a texture cache [14][25]. Research in [3] indicates that a small texture cache of 256 bits is already sufficient to achieve hitrates in the range of 90%.

After all the attributes for a fragment are computed it is passed to the next three stages of the pipeline. The TCFA stage then returns to query the logic-enhanced memory in the TSCF stage for the following fragments inside the current triangle. If the logic-enhanced memory signals that there are no more fragments in the current triangle, indicated by all zeros in the memory, the TSCF stage is ready to start the rasterization of the next triangle.

After the first two pipeline stages all operations are performed on fragments only. This is indicated by the horizontal arrow in Figure 2.4. The next two stages of the pipeline, alpha test (AT) and stencil depth test (SDT), perform configurable OpenGL tests as defined by the running 3D application. GRAAL supports the alpha, stencil, and depth test as specified by OpenGL. The SDT stage has internal memory for storing the required stencil and depth values for the current tile. This memory is referred to as the Tile Stencil Depth Buffer (TSDB). The storage of depth values and stencil values in this unit are combined in order to obtain a single 32 bit stencildepth value for easier transfer over a 32 bit bus.

The blending and logical operation (BLO) stage performs the blending of incoming fragment colors with the colors in the framebuffer or the logical operations on these color values. The framebuffer is again a small memory able to contain the color values for the pixel in the current tile. This memory is referred to as the Tile Color Buffer (TCB). The TCB is constructed from four separate 8 bit memories, one byte for each color component in RGBA.

2.5 Scenebuffer Tile-Based Rasterization

As mentioned in the previous section, the main disadvantage of tile-based rasterization on mobile devices is the need to store the instructions during the rasterization. A solution to this problem was presented and intensively studied in previous work [2].

2.5.1 Basic Concept

The basic concept behind scenebuffer tile-based rasterization is to simply store the incoming instructions, as generated by the geometry stage, in memory until all the instructions for a frame are generated. The end of a frame is indicated with a special instruction. In OpenGL this is the `glSwapBuffers` command. The rasterization hardware subsequently executes the stored instructions for every tile one after another. The memory were the instructions are stored until they are passed to the rasterizer is called the scenebuffer, therefore we refer to this rasterization approach as a scenebuffer tile-based rasterization scenario.

The instructions stored in the scenebuffer are mostly triangle instructions. Instructions that modify the rasterizer settings, called state instructions, are usually limited in number when compared to triangle instructions. Fortunately, most triangles in computer graphics are limited in area and therefore only visible in a limited number of tiles. Hence there is no need to process each triangle for every tile and a significant amount of workload on the rasterization hardware can be saved by sorting the triangles to tiles prior to rasterization.

Several sorting algorithms were proposed in previous work [8] and are discussed in further detail in Section 2.5.3. The sorting can be implemented before triangles are stored in the scenebuffer or after triangles are retrieved from the scenebuffer or a combination of both. We refer to sorting before or after the storing phase as pre-sorting and post-sorting, respectively.

The graphics pipeline in such a system is essentially a two phase process. Strictly speaking it is no longer a pipeline because the geometry stage has to be finished before the rasterization can start. The first two stages of the graphics pipeline, the application and the geometry stage, require no significant modifications if we want to adopt a tile-based rasterization approach. However due to the limited gate budget for embedded computer graphics we most likely have to implement the geometry stage in software.

Figure 2.5 depicts the first phase in the 3D rendering system implemented in a scenebuffer rasterization scenario. The 3D application is executed on the microprocessor. OpenGL instructions are generated via the application and geometry stages running on the microprocessor. The resulting data can be seen as a stream of instructions (triangles, clears and state modifications) to be issued to the rasterizer hardware in order to create an image of the frame.

If pre-sorting is used the sorting stage maintains an individual scenebuffer for every tile, called a bin. If no pre-sorting is used all instructions are simply stored to a single scenebuffer. In the second phase, depicted in Figure 2.6, the instructions stored in the scenebuffer are retrieved for every tile. In post-sorting a quick check is made if the

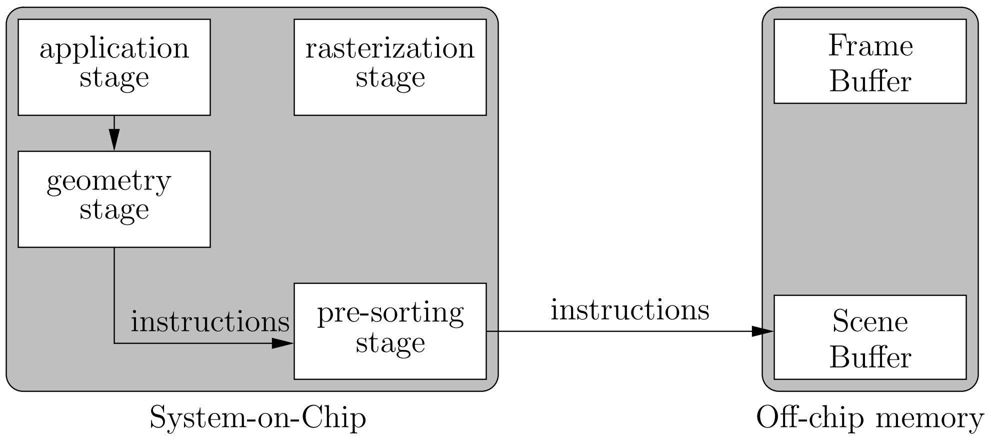


Figure 2.5: Phase one (application, geometry, and pre-sorting) in a scenebuffer rasterization scenario.

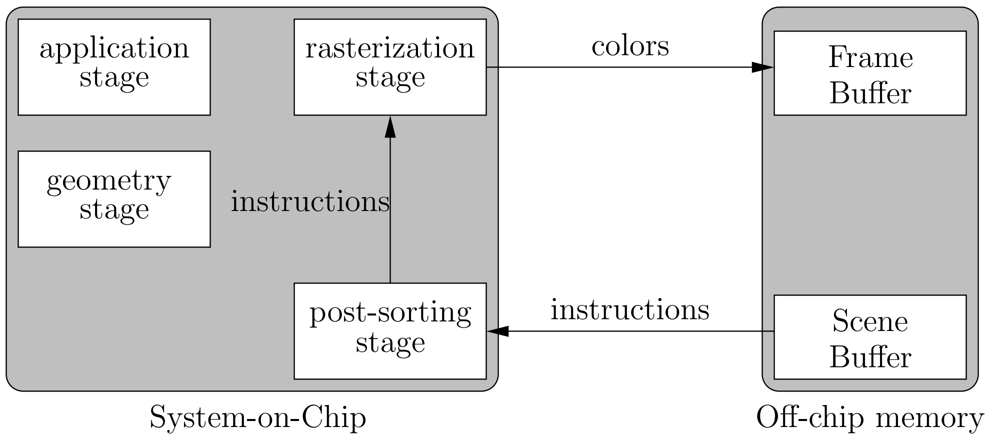


Figure 2.6: Phase two (post-sorting and rasterization) in a scenebuffer rasterization scenario.

triangles are present in the current tile. If not it is discarded without being send to the rasterizer, resulting in no hardware workload.

In this rasterization scenario all instructions for a frame are stored prior to rasterization. This means that during the rasterization of the tiles all instructions are available and every tile is only processed once. Because every tile is only rasterized once the intermediate depth and stencil values can be discarded after a tile is completed. This scenario leads to the maximal reduction of memory accesses to the external memory related to fragments when compared to a fullscreen rasterization scenario. Only the final color values are stored to the external framebuffer in this rasterization scenario.

There are unfortunately also several disadvantages that the scenebuffer rasterization scenario introduces to a graphics aystem. The first is the requirement of storing all the instructions for a frame to memory. The size of this scenebuffer is dynamic, highly dependent on the running application, and differs from frame to frame. Therefore the scenebuffer is most likely stored in external memory.

The second disadvantage is derived from the fast that the scenebuffer is stored in external memory. The creation and usage of the scenebuffer requires accesses to external memory before the actual rasterization can be started, increasing the external traffic related to triangles when compared to fullscreen rasterization. The increase of the amount of memory accesses can be limited by utilizing a sorting algorithm. Section 2.5.3 discusses different sorting algorithms as presented in previous work [8].

2.5.2 External Traffic

Because accesses to an external chip are a major source of power consumption [21] the critical aspect in the evaluation of various rasterization scenarios is the total amount of data traffic to the external memory. In the evaluation of rasterization scenarios this traffic is split in two separate groups: datafront and databack. The positioning of the two groups is illustrated in Figure 2.7.

In the datafront we group all the memory accesses related to the creation of the scenebuffer and the retrieving of instructions from the scenebuffer needed as input for the rasterization hardware. The databack groups all the memory accesses on the output side of the rasterizer hardware. All accesses between the rasterizer and the external memory are counted, this mainly includes the exchange of parameter values stored in the fullscreen depth-, color-, and stencilbuffer.

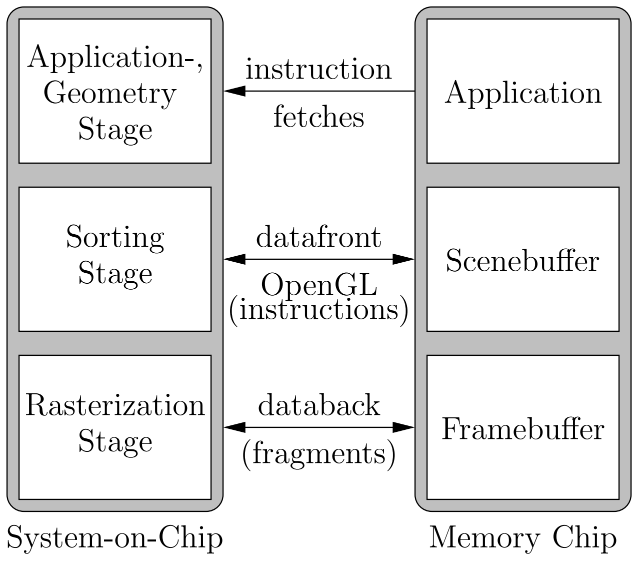


Figure 2.7: Definition of the datafront and databack traffic components.

The figure also illustrates the data traffic from the memory to the microprocessor SoC related to the instruction fetches of the application and geometry stages. It is assumed there are no significant differences in implementation between these stages in fullscreen and scenebuffer tile-based rasterization. This traffic is therefore not included in either the datafront and databack and it is omitted from all comparisons. However, differences in software implementations can result in different behavior of the instruction cache that can alter the number of instruction fetches from the external memory. This effect is thusfar not addressed in any work and assumed to be minor.

2.5.3 Sorting Algorithms

A sorting algorithm ensures that a triangle is only send to the tiles it overlaps. How triangle to tile overlap can be determined via either a bounding box test or a linear edge test is presented in Section 2.5.4. The sorting stage can either be implemented in hardware or in software depending on the gates left available in the hardware budget and performance requirements. We assume that the scenebuffer is too large to be stored on-chip and it is therefore stored in the external memory. Unfortunately, the size of the scenebuffer is highly dependent on the nature of the running application and differs from frame to frame. This requires dynamic memory allocation inside the off-chip memory. This can be easily done by the operating system (OS) running on the microprocessor, which makes it likely that the sorting stage is implemented in software.

Five different sorting algorithms were presented and compared in [8]. The first of these algorithms, the simplest called `direct`, was directly discarded from consideration due to poor performance when compared with the other four algorithms. The remaining four algorithms can be divided in two groups. Each group contains two algorithms that only differ in the way the overlap is computed.

The two algorithms in the first group, `two_step` and `two_step_let`, work with a single scenebuffer which is used for every tile. This single scenebuffer is completely processed for each and every tile continuously. Before triangles are stored to the scenebuffer, as part of the pre-sorting, a bounding box is computed that is stored together with the triangle. A post-sorting check is made whether the triangle has overlap with the current tile after a triangle is retrieved from the scenebuffer. The two algorithms in this group differ in the way they determine this overlap. Algorithm `two_step` only performs a fast bounding box test and algorithm `two_step_let` uses a slower, but more accurate linear edge test.

The other two algorithms, `sort` and `sort_let`, work with a more complex scenebuffer constructed from bins, which are essentially individual scenebuffers for all tiles in the screen. During the pre-sorting all the triangles are examined and only stored in the bins they overlap. Again the algorithms `sort` and `sort_let` differ in the way overlap is determined, either via a simple bounding box test or by a more complex linear edge test.

Figure 2.8 graphically represents the bounding box sorting in the `sort` algorithm for a 4 $\times$ 3 tiles rasterization scenario. The left image represents the positioning of the triangle within the tiles that compose the screen. The right image represents the construction of the scenebuffer in memory. The bounding box of the triangle is present in the six rightmost tiles of the screen. During pre-sorting the triangle is stored in the bins marked with an 'X', representing the right most tiles. Due to the inaccuracy of the

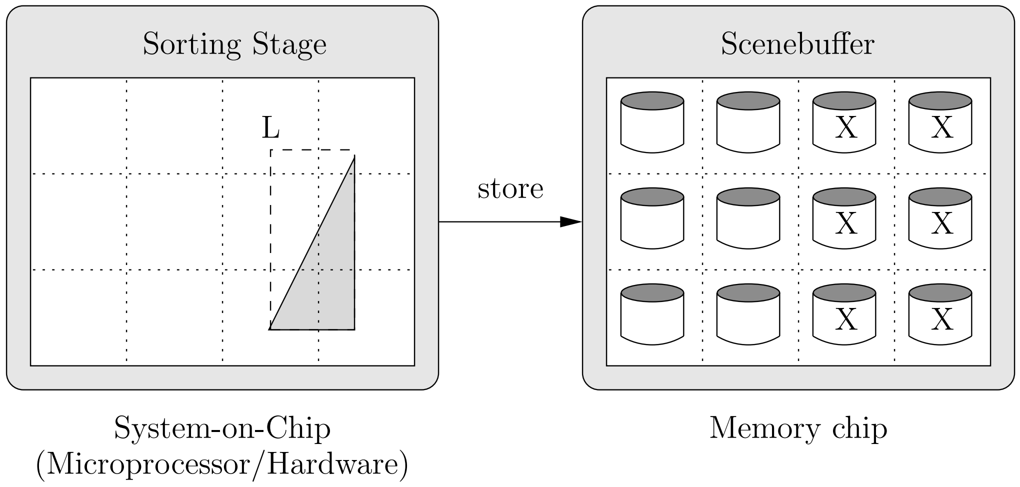


Figure 2.8: Sorting a triangle to bins in the scenebuffer.

bounding box test the triangle is incorrectly added to the uppermost left tile, marked with 'L' in the sorting stage. Using a linear edge test to improve accuracy eliminates this incorrect sorting at the expense of extra computations in software.

However, the instructions needed to rasterize a frame as provided by the geometry stage are not limited to triangles alone. Special instructions, called state instructions, that modify the state of the rasterization engine are also generated by the geometry stage. These instructions set the correct parameters for the fragment alpha testing, depth testing, blending, etc. The state of the rasterization engine determines how triangles must be rasterized and instructions to modify the state can be send to the rasterization engine anytime during the rendering of a frame. The effects of a state modification instruction influence all triangles that follow the instruction. Therefore state modifying instructions must be send to all bins.

Simulations oF the behavior of a tile-based rasterization solution using the sorting algorithms in [7] showed an increase of data traffic on the input side of the system. The largest part of the increase of input traffic was caused by triangle duplication. Triangles can be present in several tiles and therefore need to be retrieved for every tile. Moreover, a significant part of this traffic increase was due to state instructions that need to be stored to every tile. This is in contrast to the fullscreen rasterization scenario where the amount of data traffic due to state instructions is negligible when compared to triangle instructions.

In [5] a lazy state update routine was considered that reduces the amount of state modifying instructions send to the rasterizer. Although this routine significantly reduces instructions issued to the rasterization hardware it does not reduce the amount of state modification instructions retrieved from the scenebuffer significantly. State instructions still need to be stored and retrieved for every tile.

To limit the increase of data traffic in the datafront it is intuitive that we need to limit the number of tiles in the screen. In [4] an optimal tile size for a tile-based rasterizer was determined for the use in a scenebuffer tile-based rasterization scenario. It was concluded that a tile size of 32×32 is the optimal choice in order to reduce the amount of traffic in the datafront while still remaining inside the hardware budget of 500.000 gates.

2.5.4 Overlap Determination

The simplest way to determine the triangle to tile overlap is via a bounding box test. In this test a triangle is replaced by a bounding box, a four-tuple with the minimum and maximum x and y coordinates of a triangle. This is simply done via Equations (2.1), (2.2), (2.3), and (2.4).

$$x_{min} = min(x_A, x_B, x_C) \tag{2.1}$$

$$y_{min} = min(y_A, y_B, y_C) \tag{2.2}$$

$$x_{max} = max(x_A, x_B, x_C) \tag{2.3}$$

$$y_{max} = max(y_A, y_B, y_C) \tag{2.4}$$

This constructed bounding box is then used to check the overlap with the tiles. A triangle and a tile overlap when the following equations hold true:

$$x_{min} < x_{right} \tag{2.5}$$

$$y_{min} < y_{top} \tag{2.6}$$

$$x_{max} \geq x_{left} \tag{2.7}$$

$$y_{max} \geq y_{bottom} \tag{2.8}$$

Where x_{left}, x_{right}, y_{bottom}, and y_{top} are the boundaries of a tile. To determine the tile overlap for the all the tiles in the screen we need to evaluate Equations (2.7), (2.5), (2.8), and (2.6) for every tile. This process is illustrated in Figure (2.9) for a simple 5×4 tile screen. The darkened tiles are tiles that are considered part of the triangle overlap with the bounding box test. This figure also highlights a problem with the bounding box approach. Because the triangle is replaced by a square the overlap is not always determined correctly. As it is obvious from the figure the upper left tile in the bounding box is considered part of the triangle to tile overlap eventhough the triangle is not in it. We call the overlap in such a tile a ghost triangle.

Ghost triangles as introduced by the bounding box test can decrease the performance of the hardware rasterizer. Like normal triangles, ghost triangles are also send to the tile-based rasterizer. In other words some triangles are send to the rasterizer for tiles they are not present in. In the tile-based rasterizer these triangles are scan-converted and discarded when the rasterizer finds that none of the pixels was influenced. While the final image is still correct the rasterizer performs some unnecessary scan-conversions.

We can improve the precision of the overlap by performing an additional linear edge test as presented in [31]. When using linear edge testing a special function is introduced that

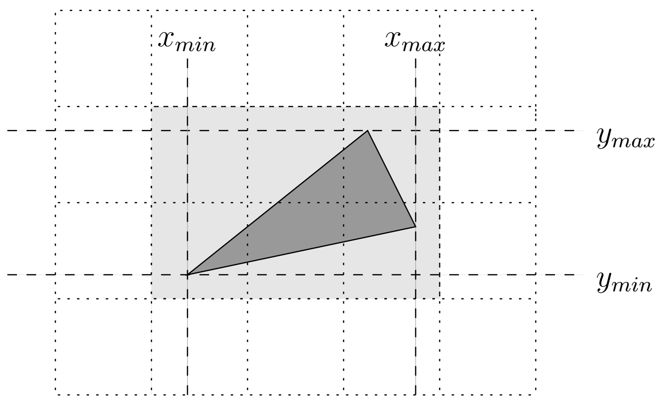


Figure 2.9: Determining the tile overlap via a bounding box test.

defines a relation between a point P and a triangles edge. When the pixel is to the right side of the edge the result of the function is positive, when the pixel is on the left side of the edge the result is negative, and when the pixel is exactly on the line the result is zero.

Consider a pixel position P and a triangle defined by the vertices A, B, and C. The edge functions for pixel P with respect to the edges of the triangle ABC are given in Equations (2.9), (2.10), and (2.11).

$$E_{AB}(x,y) = (x_P - x_A)\Delta y_{AB} - (y_P - y_A)\Delta x_{AB} \tag{2.9}$$

$$E_{BC}(x,y) = (x_P - x_B)\Delta y_{BC} - (y_P - y_B)\Delta x_{BC} \tag{2.10}$$

$$E_{CA}(x,y) = (x_P - x_C)\Delta y_{CA} - (y_P - y_C)\Delta x_{CA} \tag{2.11}$$

Where x and y are the coordinates of pixel P, Δx denotes the horizontal difference between the two vertices of an edge, and Δy denotes the vertical difference between the two vertices of an edge. Depending on the orientation of the triangle, either clockwise or counterclockwise, the pixel is inside the triangle if all evaluated edge functions are positive (clockwise) or negative (counterclockwise).

Figure 2.10 depicts the results of this process for a triangle ABC, where the vertices are defined clockwise. The edges of triangle ABC divide the plane, defined by the three vertices of ABC, in seven different regions. The sign of the edge functions E_{AB}, E_{BC}, and E_{CA}, in each region is represented by a tuple of $+$ and $-$ signs, which represent the signs of the results of Equations (2.9), (2.10), and (2.11), respectively. As shown only the region inside the triangle has the same sign for all three edge functions.

These equations can also be used to determine if a triangle intersects a square, or in our case a tile. In other words we need to compute if any of the pixels in a square are inside a given triangle. In [2] the following equations were presented to determine the intersection of a triangle, defined by vertices $A(x_A, y_A)$, $B(x_B, y_B)$, and $C(x_C, y_C)$, with a square, defined by a centerpoint (x_S, y_S) with a total width W:

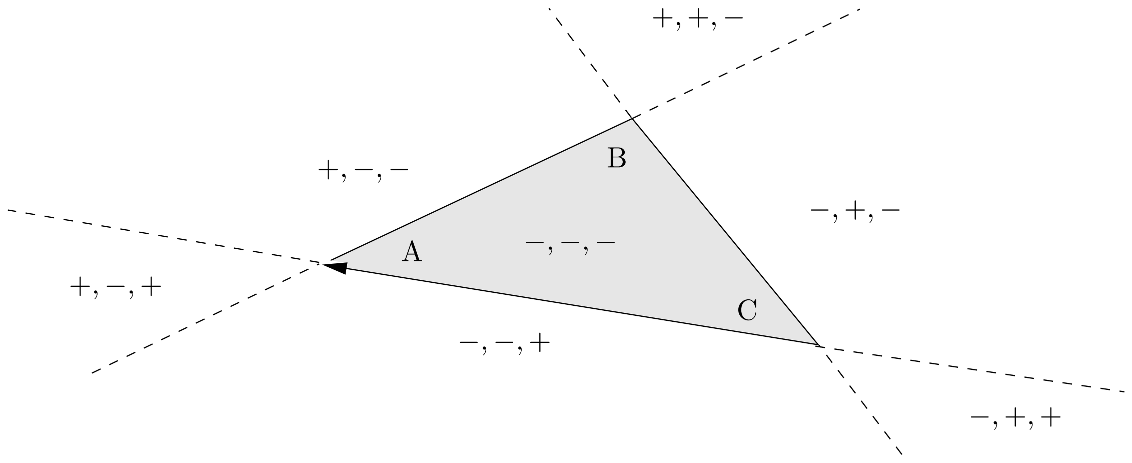


Figure 2.10: Determining the inside of a triangle based on linear edge functions.

$$E_{AB}(x_S, y_S) \leq \frac{1}{2} \cdot W \cdot (|x_B - x_A| + |y_B - y_A|) \tag{2.12}$$

$$E_{BC}(x_S, y_S) \leq \frac{1}{2} \cdot W \cdot (|x_C - x_B| + |y_C - y_B|) \tag{2.13}$$

$$E_{CA}(x_S, y_S) \leq \frac{1}{2} \cdot W \cdot (|x_A - x_C| + |y_A - y_C|) \tag{2.14}$$

Where E_{AB}, E_{BC}, and E_{CA} represent the linear edge functions for the lines AB, BC, and CA as calculated via Equations (2.9), (2.10), and (2.11), respectively. By noting that every tile can be regarded as a rectangle with width w and height h we can use a normalization of the coordinate space (division by (w,h)) to transform every tile to a square with width 1. The above Equations (2.12), (2.13), and (2.14) can then be used to refine the overlap testing.

In the following chapter we discuss some drawbacks to the proposed scenebuffer tile-based rasterization and propose an alternative tile-based rasterzation scenario to counter these drawbacks.

Direct Tile-Based Rasterization 3

This chapter begins with a critical analysis of the previous work in Section 3.1. Afterwards in Section 3.2 we present the basics of a new tile-based rasterization scenario, which we refer to as direct tile-based rasterization or simply direct rasterization. The implications of this rasterization scenario on the overal system are presented in Section 3.3. Given that the proposed direct rasterization and the scenebuffer rasterization are not mutually exclusive we present in Section 3.4 a combined solution which attempts to preserve the benefits of both approaches. Section 3.5 concludes the chapter with a summary of the different rasterization scenarios we aim to compare using simulations of realistic mobile applications.

3.1 Evaluation of Previous Work

In tile-based rasterization no stencil and depth values are stored into external memory. Only after a tile is completed are the color values stored to the external memory, while the stencil and depth values stored in the internal memory are simply discarded. This approach assumes that every frame is started with a `glClear(GL_DEPTH, GL_STENCIL)` instruction, reverting all depth and stencil values to a base value. Unfortunately, this assumption does not hold for all graphics applications.

Some applications use the stencil test to prohibit writing to pixels that do not need to be modified for each frame. For example, flying simulations first render the cockpit in the first frame with stenciling enabled, causing a set operation in the stencilbuffer on the locations that represent the cockpit. In the following frames, the surroundings are only drawn on pixels that are not part of the cockpit. In these applications the stencil values need to be stored in the external memory and retrieved for every frame causing an increase of databack traffic. As a side note it was mentioned in [6] that applications using the stencil buffer are uncommon for mobile devices.

In [4] the optimal tile size for the rasterizer in a scenebuffer-based rasterization scenario was determined to be 32×32 pixels. This was based on the resulting total traffic versus the size of the internal memory used to stored rasterization parameters. Increasing the tile size beyond 32×32 still reduces the datafront traffic but this reduction in traffic is outweighed by the increase of hardware required, in terms of hardware gates, in order to create the internal memory.

We observed that this proposed optimal tile size of 32×32 pixels requires the storage of the parameters of 1024 seperate pixels for a tile, resulting in a total tile size of 65.5k bits. This is based on [2] which assumes the four color components are each represented in 8 bit, the depth value is represented in 24 bit, and stencil value is represented in 8 bit. We assume that the on-chip memory is implemented in embedded SRAM, which requires approximately 6 gates to implement a memory cell for a single bit [27], since this can be created with a standard CMOS process. Embedded DRAM, which only requires approximately 1.5 gates per bit, requires a more expensive production process while SRAM memory can be created with the standard CMOS fabrication process. The resulting on-chip memory already requires 393k gates for the SRAM storage cells alone. This excludes gate overhead caused by address decoding logic, read/write logic and output multiplexing logic.

The available budget used for an embedded graphics pipeline according to ARM, a leading supplier for embedded processors, is between 200k and 500k gates [9]. This was assumed as a design constraint in the previous work as well [2]. We note that an 32×32 pixels tile size already consumes a large part of the available 200k to 500k hardware budget just for the tilebuffers alone.

With the previous fact in mind, the system envisioned in the previous research relies heavily on the microprocessor for correct 3D rasterization. Because the internal memory to store intermediate parameters during rasterization already consumes a large part of the hardware budget the application and geometry stage are assumed to be implemented in software running on the microprocessor.

Whether the sorting stage was implemented in hardware or software was left open by previous research. The size and construction of the scenebuffer is highly dependent on the running 3D application and furthermore, differs from frame to frame for most applications. Constructing the scenebuffer most likely requires dynamic allocation in the off-chip memory, which is the responsibility of the Operating System (OS) software on the microprocessor. Therefore, we can safely assume that the sorting stage in a scenebuffer tile-based rasterization is implemented in software running on the microprocessor.

We note that the microprocessor is the core of the mobile device and during rendering it is likely that the microprocessor is also required for other computational tasks within the system, related to the applications audio, user interface, and wireless communication. Furthermore, after a triangle is generated the microprocessor has to perform some sorting computations before it can continue the software computations needed for generating the next triangles. This is in contrast to fullscreen rasterization where the microprocessor does not need to spend time sorting. In scenebuffer rasterization this most likely reduces the triangle generation rate when compared with fullscreen rasterization.

3.2 Direct Tile-Based Rasterization

The biggest disadvantage of scenebuffer tile-based rasterization we encountered, as described in Section 2.5, is the amount of sorting workload before the actual rasterization is performed. In scenebuffer rasterization all rasterization instructions needed to render a single frame are stored to the scenebuffer in external memory before rasterization. Due to the dynamic nature of this scenebuffer, which can greatly differ between applications and even between consecutive frames in a single application, and because the hardware budget for computer graphics is quite small the sorting algorithm is assumed to be implemented in software. Hence, the workload of the software running on the microprocessor is increased when compared to fullscreen rasterization.

In this thesis we propose an alternative sorting algorithm for use in tile-based rasterization systems on mobile devices, which we refer to as direct tile-based rasterization. In direct rasterization we store a number of rasterization instructions in a local hardware unit, called the direct-sorting unit, and start the rasterization directly with only this limited number of instructions. In this way we remove the sorting workload on the microprocessor and remove the need to store the scenebuffer in external memory and thus remove external traffic prior to rasterization. The direct-sorting units effectively functions as an interface between the microprocessor and the hardware rasterizer.

In practice, the datafront is entirely removed by sending triangles directly to the direct-sorting unit. But during this research we assume the external memory is always needed to act as a buffer between the geometry stage of the graphics pipeline, that generates the rasterization instructions, and the rasterization stage of the graphics pipeline, that consumes the triangles. In other words we assume that the rate in which triangles are generated is not equal to the rate in which triangles are consumed. This is the worst-case scenario for the datafront traffic in direct tile-based rasterization, which can be solved easily in practice by performing non-graphical software workload when the direct sorting unit is not ready to receive new triangles. Therefore we also show results were this datafront is not present in direct rasterization.

3.2.1 Basic Concept

Consider a triangle stream T_{stream} consisting of all the triangles required to rasterize a frame. In contemporary 3D graphics applications this can range from hundreds of triangles to even millions of triangles for the most complex 3D applications. Considering we are dealing with mobile devices we can propably exclude complex CAD applications generating millions of triangles per frame. The stream of triangles is generated by the application and geometry stages of the graphics pipeline implemented in software running on the microprocessor. In scenebuffer rasterization this entire stream is stored in external memory before the rasterization stage starts.

However, in direct rasterization the rasterization process is started immediately after a few triangles are generated and stored in the direct-sorting unit. The triangles from T_{stream} are fed directly in to following algorithm implemented by the direct-sorting unit in hardware:

```
**Direct Sorting** ($T_{stream}$)
    Store $W$ triangles from $T_{stream}$ in $T_{buffer}$
    **while** $T_{buffer}$ holds triangles
        $tile$ = **Select Tile** ($T_{buffer}$)
        **for** all triangles in $T_{buffer}$
            **if** $triangle$ is present in $tile$
                send $triangle$ to rasterizer
                mark $triangle$ as done for $tile$
                **if** $triangle$ is done for all $tiles$
                    $triangle$ = next triangle from ($T_{stream}$)
```

Where T_{buffer} is a triangle buffer inside the SoC large enough to hold W triangles. The rasterization process of the tiles is started with the triangles stored in this buffer. The triangles in the T_{buffer} form a sort of window over the total triangle stream T_{stream} and hence we refer to W as the triangle window.

In this new sorting algorithm only a limited number of triangles of the entire frame are available for processing at any one time. In the rasterization stage we can therefore no longer simply process the tiles in a predetermined order as it is the case in scenebuffer rasterization. Therefore the first step in the algorithm is to select a suitable tile to process, indicated in the algorithm by the **Select Tile()** function, based on the triangles that are currently inside the triangle window. We introduce several heuristics that implement this function in Section 3.2.3.

After a tile is selected for processing the triangles inside the triangles window are examined for overlap with the selected tile. All triangles with overlap to this selected tile are send to the tile-based rasterizer for processing. Triangles that do not overlap with the selected tile are ignored until a new tile is selected for processing. When all triangles inside the triangle window are examined a new tile is selected by the **Select Tile()** function.

The concept of direct rasterization relies on spacial proximity of consecutive triangles in the triangle stream as generated by the geometry stage. Triangles are commonly defined close together in order to create larger complex objects. Therefore consecutive triangles have a large probability of being present in the same tiles. By focusing on the rasterization of a small number of consecutive triangles the complex scenebuffer sorting phase in software can be omitted and traffic to and from the external memory during the sorting phase is minimized or can in theory even by entirely removed.

The concept is visualized in Figure 3.1. After the geometry stage the input for the rasterization stage can be represented as an instruction stream, of which most of the instructions are triangles, ready to be send to the rasterization stage. In direct rasterization we store a limited number of consecutive triangles from this stream inside a hardware buffer on chip. This limited number of triangles forms a triangle window over the total triangle stream, indicated by the gray square over the stream in Figure 3.1.

When a triangle is completed for all tiles it is present in, the triangle is discarded and the triangle window is advanced by introducing a new triangle from the triangle stream. The triangles do not have to be discarded from the triangle window in order in which they are introduced since not all triangles have the same size and some of the newest triangles could actually be finished before the first triangles are finished. The processing of, discarding of, and the introduction of triangles continues until all the triangles in the triangle stream have been processed and the final 2D image is correctly stored in the framebuffer located on the external memory chip.

The biggest disadvantage our proposed solution introduces is the limited number of triangles that can be stored on-chip. The same limitations on the gate budget as presented in Section 2.1 prohibit us to store a large number of triangles on chip. And additionally a triangle requires quite a lot of storage as is discussed later in Section 4.3.1. This means

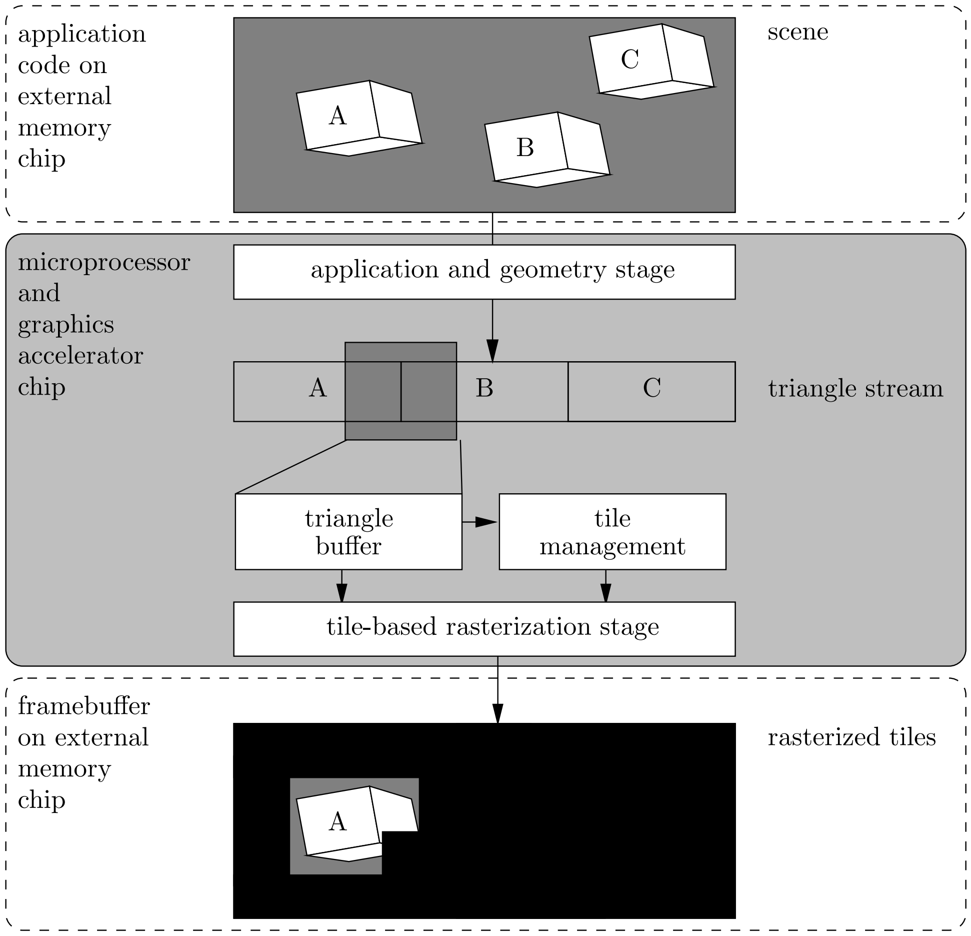

application code on external memory chip
scene
C
A
B
microprocessor and graphics accelerator chip
application and geometry stage
A
B
C
triangle stream
triangle buffer
tile management
tile-based rasterization stage
framebuffer on external memory chip
rasterized tiles
A

Figure 3.1: Direct tile-based rasterization concept in steps.

that we have to start the rasterization with the triangles available in the direct-sorting unit, which is only a small portion of the total triangles needed to correctly render an entire frame.

Based on the triangles in the current triangle window the most appropriate tile to process is determined and selected and all triangles with overlap to this tile are send to the rasterization hardware. Unfortunately, triangles outside the triangle window cannot yet be send to the rasterizer and future triangles might be present in the same tile as current triangles. Therefore some tiles must be selected and processed in multiple visits and intermediate values must be stored to the framebuffer in external memory between tile visits. This is in contrast to the scenebuffer tile-based rasterization scenario where each tile is only processed once and we only need to store the final color values.

When a rasterization pass of a tile is complete all the intermediate fragment values for the current tile need to be stored to off-chip memory. As described in Section 2.1 it is critical to minimize the number of accesses to the off-chip memory since they are power hungry. Since our proposal increases the number of rasterization passes to a tile we also increase the amount of databack traffic, the traffic related to accesses to the off-chip memory for storing and retrieving fragment values. On the other hand, we do not create the scenebuffer, and therefore the amount of datafront traffic in our proposal is equal to the datafront traffic in a fullscreen rasterization scenario.

3.2.2 Overlap Representation

In computer graphics triangles are defined via three vertices with the parameters given in Table A.1. Unfortunately, the screen coordinates of these vertices do not immediately indicate which tiles the triangle is present in. The same techniques as described in Section 2.5.4, to determine if pixels are inside a triangle, can be used to determine triangle to tile overlap. However, in order to prevent the re-computation of this overlap for a triangle for every processed tile, the triangle data stored in the direct-sorting unit are supplemented with a special data structure representing this triangle to tile overlap.

Inside the direct-sorting unit the overlap information is used to determine which tile to process, if a triangle is present in a tile, and if the triangle is completely processed for all tiles it is present in. The representation needs to satisfy the following criteria in order to enable correct functioning of the direct-sorting unit, while keeping in mind that this representation is to be implemented and used in hardware.

- It must indicate whether a triangle is present in a tile fast and simple,
- it must allow a triangle to be marked as completed for a tile in order to prevent rasterization of a triangle in the same tile twice,
- it must allow for easy determination of the fact that a triangle is completed for all tiles so that the triangle can be discarded from the triangle window.

Several options were considered for representing the triangle to tile overlap. The first is a tilemask representation, where we assign a bitvector to each triangle, where every bit represents a tile. A zero in the tilemask signifies that the triangle does not have to be rasterized for the corresponding tile. A one in the tilemask signals that the triangle still has to be rasterized for that tile. The tilemask representation allows triangles to be checked if they overlap to a certain tile by simply checking the corresponding bit. Similarly triangles are easily marked as processed for a tile by setting the corresponding bit to zero. Determining if a triangle is complete and can be discarded is done by checking if the tilemask contains all zeroes. Figure 3.2 illustrates the representation of a triangle via a tilemask.

Representing the tile overlap in a tilemask is easily implemented in hardware and working with the tilemask is easy. However, the size of the tilemask representation is quite large. For example, a 640×480 screen rasterized on a tile-based rasterizer with dimensions 32×32 already requires a 300 bit tilemask representation for every triangle. Thus, using a tilemask representations seems to be recommended for screens that are composed of a low number of tiles.

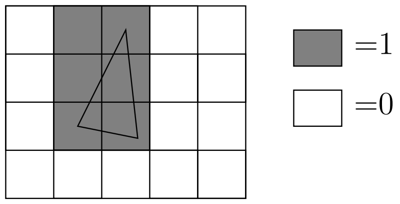


Figure 3.2: Representing the triangle inside the buffer with a tilemask

The second representation that can be implemented is a tilelist. The screen is essentially a 2D grid where every tile can be represented by a single index, or by a combined horizontal and vertical index. If we assume that the tilelist elements can be removed or invalidated this representation also satisfies the above mentioned criteria. The advantage of a tilelist is that it is easy to determine the first tile of a triangle simply by checking the first element of the tilelist and checking if a triangle is complete and can be discarded is done by checking if there are elements left in the list.

However, checking if a triangle is present in a certain tile, probably the most time critical responsibility of the direct-sorting unit, requires a number of comparison for each triangle. Each element of the tilelist needs to be compared to the current tile. Since the triangle to tile overlap is dynamic and can range from one to the entire screen the length of the tilemask is dynamic. This representation can therefore lead to complex hardware implementation.

In the following we assume that the triangle to tile overlap inside the direct-sorting unit is implemented using a tilemask. Although this representation is large to store in hardware its utilization is easily implemented in hardware without complex hardware computations.

3.2.3 Tile Selection

In the scenebuffer-based rasterization approach introduced in Section 2.5 all tiles in the screen are processed sequentially. The tile rasterization order in scenebuffer rasterization is arbitrary because every tile is only processed once. However, in our newly proposed direct rasterization scenario the tile rasterization order is dependent on the triangles present in the triangle window and thus on the running graphical application. Furtermore tiles can be processed in several visits instead of only a single visit. Every time a tile is processed beyond the first visit requires retrieving intermediate values from the framebuffer in external memory. This means that the tile rasterization order is no longer arbitrary, but it must be carefully choosen in order to minimize accesses to external memory.

Because only a limited amount of triangles are available in the sorting unit some sort of tile selection mechanism is needed that ensures a proper tile is selected for processing. However, there is no clear way to determine which tile should be selected based on the triangles currently inside the triangles window, because these triangles only represent a small fraction of the entire frame.

In the end the main goal of a tile selection mechanism is to reduce the amount of external traffic to the framebuffer related to reading and writing of temporary pixel values. In our new proposed rasterization scenario reading and writing of pixel values is done when tiles changes. Therefore the tile selection policy should try to aim to reduce the number of tile visits.

In this research we compare the total external traffic generated by several heuristics that can be used for selecting a tile to process. These heuristics are the following:

- `first_triangle`,
- `skip_large`,
- `smallest_triangle`,
- `densest_tile`.

The listed selection mechanisms are ordered in their increasing complexity meaning that `first_triangle` is the simplest solution to implement and `densest_tile` is the most complex to implement inside the direct-sorting unit implemented in hardware.

The `first_triangle` policy simply selects a tile to process based on the first triangle inserted in the triangle window. In this selection policy we assumed tiles are selected from the bottom left to the top right, first selecting all tiles in a row before advancing to the next row. The `first_triangle` selection policy simply selects the first tile of the first triangle. This selection policy does not require complex hardware to determine the best tile to process as it only needs to find a tile that overlaps the first triangle. This policy is basically the implementation of a first in, first out (FIFO) policy that aims to process the first triangle as fast as possible in order to advance the triangle window.

The `skip_large` policy aims to address one drawback of the `first_triangle`. Large triangles that are present in a large number of tiles tend to reduce the effectivity of the `first_triangle` policy by introducing a lot of tile visits before the large triangle is completed and can be discarded. Commonly, large objects are used to set the background of a scene while afterwards smaller, more detailed, objects are used to contruct the foreground of a scene. In the first selection policy the triangles are processed in their generation order and the background triangles are first rasterized and stored to the framebuffer in external memory only to be overwritten by a following tile visit.

The `skip_large` policy solves this problem by ignoring large triangles in the tile selection process and thus prevents tiles with background objects to be selected for processing. As a sidenote we mention that not all large triangles are used for background purposes and vice versa not all background objects are large. A better solution would be to check the triangles z-coordinate, but not every application uses the same depth values for background objects. Therefore we have no prior knowledge to base a classification of objects as background accurately. The easiest solution might be to provide background information in the software application. But this requires modifications in the OpenGL standard and is therefore not considered here.

The `smallest_triangle` policy aims to empty the triangle buffer in the direct-sorting unit as quickly as possible in order to advance the triangle window. By quickly advancing the triangle window this policy aims to introduce the triangles to the triangle window as fast as possible. Like in the scenebuffer rasterization scenario, we no longer need to store intermediate stencil and depth values when the final triangle of a frame is present in the direct-sorting unit and thus we can significantly reduce the databack traffic when we know the frame is nearly completed.

This policy assumes that rasterizing the smallest triangle is the quickest way to ensure the triangle window is advanced. This policy requires information about the size of a triangle. The area of triangles is calculated in the software side of the graphics system in order to determine the triangles orientation (clockwise, counterclockwise) needed to perform culling and could therefore be directly provided as a parameter to the sorting unit. Another alternative is to express the size of a triangle in the tile overlap, the tile overlap needs to be determined before we can actually employ direct rasterization and is therefore quite simple to calculate.

The `densest_tile` selection policy takes the tile that currently has the highest amount of triangle overlap. This selection policy ensures tiles that result in the highest amount of hardware workload are selected and tries to minimize the number of tile switches. In order to find the densest tile we need to check the tile overlap for all triangles for all tiles and perform additions whenever a triangle overlaps a tile.

3.2.4 Databack Reduction

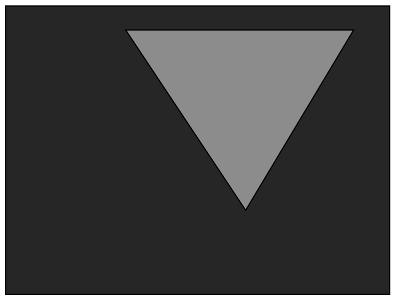
In a direct rasterization scenario tiles could be processed in several visits. This requires the storage and retrieval of intermediate fragment values to and from the framebuffer in external memory. As mentioned these accesses to external memory need to be minimized since they are a major source of power consumption. The main goal of tile-based rasterization is to minimize accesses to the external memory in order to reduce the power consumption.

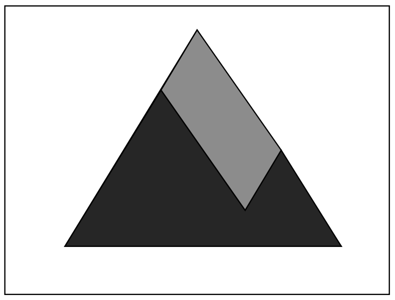
Fortunately, it is not necessary to retrieve and store the fragment values for every pixel in a tile during a rasterization visit of a tile. Consider the following scenario, where after the geometry stage the stream of rasterization instructions is as in the following example of code:

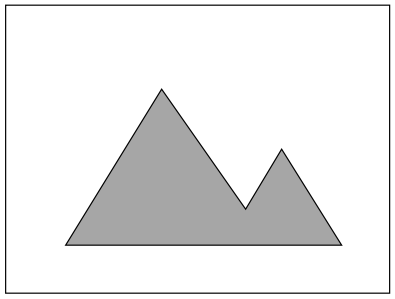
```
glClear();
Triangle(1); // present in tile (i,j)
//
Triangle(2); // NOT present in tile (i,j)
Triangle(3); // NOT present in tile (i,j)
...
//
Triangle(N); // present in tile (i,j)
```

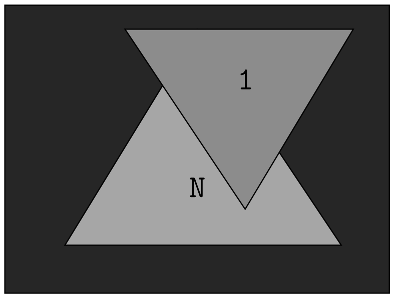
Where `N` is large enough to ensure that triangles `1` and `N` are not rasterized in the same rasterization visit for tile (i, j), while all triangles between triangles `1` and `N` do not influence this tile. This ensures that tile (i, j) is selected for processing twice. We assume the case that triangle `1` is in front of triangle `N` and the triangles are positioned so that they result in a certain amount of overlap. For example, the results of the image after all the rasterization is complete could be like the image depicted in Figure 3.3(d).

As stated above, tile (i, j) is rasterized in two passes. In the first visit the `glClear()` and `Triangle(1)` instructions are send to the rasterization hardware. Let us assume that


(a) Rasterization output: first pass


(b) Rasterization input: second pass


(c) Rasterization output: second pass



(d) Rasterization result: final

Figure 3.3: Rasterization progress for tile (i, j).

all the fragment values are modified by the `glClear()` instruction, which is generally the case with a `glClear()` instruction. After the first visit all temporary pixel values of the tile need to be stored to the off-chip memory. The result of this first rasterization visit is depicted in Figure 3.3(a).

After this first visit of tile (i,j) the tile-based rasterizer processes the other triangles that influences other tiles and eventually a second visit to tile (i, j) is required. In this second visit the `Triangle(N)` operation is executed. Because triangle `N` does not influence the entire tile, we do not need to retrieve all the fragment values of the entire tile. Only the values of the fragments that are inside triangle `N` are required and need to be retrieved from the framebuffer as depicted by the colored section of the frame in Figure 3.3(b). The uncolored fragments are not retrieved from the external memory and therefore do not result in external traffic.

Additionally, not all values inside triangle `N` are modified during the second rasterization visit of tile (i, j) since triangle `1` is in front of triangle `N`. The values that are inside triangle `1` are still correctly stored in the framebuffer and are therefore not required to be stored again. Figure 3.3(c) represents the values that eventually need to be stored to the external memory after the second rasterization visit.

Therefore, to reduce traffic to and from the external memory, the rasterization hardware (GRAAL) needs to be modified in order to record if a certain fragment value stored in the tilebuffers in on-chip memory is valid. This requires three bits per pixel, two for the stencil and depth values and one for the color values. If a fragment value is not valid it needs to be retrieved from the framebuffer in external memory. Furthermore, the rasterization hardware needs to record if a certain fragment value has been modified during the current rasterization visit. This requires another three bits per pixel. If a fragment value is not modified during the current rasterization visit it is not stored to the framebuffer in external memory.

3.3 Software and Hardware Modifications

The proposed direct rasterization scenario using a direct-sorting unit in hardware affects the system in three mayor areas: the software workload running on the microprocessor, the resulting traffic to and from the external memory, and the required hardware for the direct-sorting unit and the hardware required for the graphics accelerator. The system topology changes to the system topology depicted in Figure 3.4.

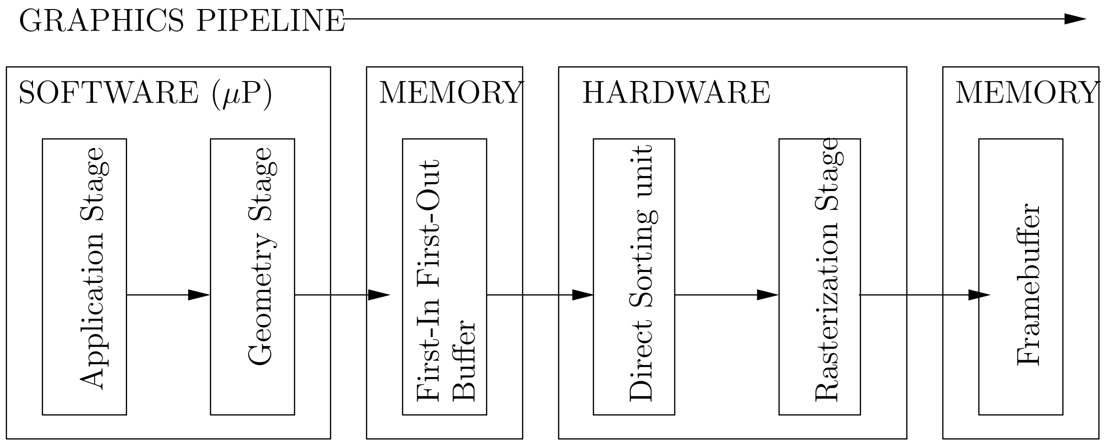


Figure 3.4: Direct (Tile-Based) rasterization system topology.

As it can be deduced from the figure the direct rasterization using a direct sorting unit in hardware again resembles a software-hardware pipeline on a per primitive level. Unlike the scenebuffer rasterization scenario where the pipeline was basically on a per frame level. As mentioned in Section 3.2 the utilization of the main memory to temporary store instructions is a worst-case scenario.

3.3.1 Software

The implications on the software side of the system are simple since the microprocessor no longer performs tile-based sorting inside the software. We assume that the remaining graphic pipeline stages (application and geometry stages) are implemented in software exactly identical as is the case in a conventional graphics pipeline. In contrast to scenebuffer rasterization where the software is responsible of creating a scenebuffer in external memory the software now only has to store generated triangles in a First-in First-Out (FIFO) buffer in external memory (worst-case since it results in external traffic) or directly to the direct sorting unit inside the SoC (realistic and preferable case since it results in internal traffic).

Without this sorting complexity the microprocessor can allocate more resources on the application and geometry stage in our direct rasterization scenario. Consequently, we dissipate less power on the microprocessor per triangle and as an additional result, the triangle generation rate is increased which could improve the overall system performance. However, both of these these implications are not evaluated during this research since They are highly dependent on the chosen microprocessor and their analysis requires very detailed software models. Instead we focus on the amount of external traffic as an indication of power consumptions.

3.3.2 External Traffic

When compared to the scenebuffer rasterization the usage of the external memory is changed in the direct rasterization scenario. First of all, removing the sorting reduces the memory imprint of the system software, which in turn results in less instruction fetches from the main memory. However, because the microprocessor is assumed to be supplemented with a instruction cache and a data cache this effect is not considered very significant. Additionally, this effect is hard to evaluate without an exact specification of the microprocessor, cache architecture, and final system software and this effect is therefore not further considered in this research.

Second, we remove the creation of the scenebuffer in external memory. In the ideal situation this completely removes the datafront traffic related to storing rasterization instructions, triangles and state instructions, to the scenebuffer in external memory because triangles can be send to the direct-sorting unit directly. If the rate in which triangles are generated by the geometry stage on the microprocessor is equal to the triangle consumption rate of the GRAAL hardware no buffering of rasterization instructions is required in external memory. If the triangle generation rate on the microprocessor surpasses the triangle consumption rate of the direct-sorting unit, a handshaking mechanism can signal the microprocessor to stall triangle generation and temporarily perform some non-graphical software instructions untill the direct-sorting unit is ready to receive another triangle.

Third, and the most prominent difference to scenebuffer rasterization is the increase of databack traffic because tiles are now processed in multiple visits. Since tiles are not processed in a single visit as in scenebuffer rasterization intermediate color, depth, and stencil values need to stored to and retrieved from the external memory. This results in an increase of the amount of databack traffic when compared to the scenebuffer rasterization.

3.3.3 Hardware

The required hardware changes when compared to the scenebuffer rasterization scenario. In essence there are not many hardware changes in direct rasterization to the actual rasterizer. But, if we take the basic tile-based rasterizer from a scenebuffer rasterization scenario as a referrence we need to add the following hardware to this rasterizer:

- Hardware additions to the tile-based rasterizer allowing it to store and retrieve intermediate depth and stencil values to and from the external framebuffer. In the tile-based rasterizer used in a scenebuffer scenario storing and retrieving the depth and stencil values would only be needed in special cases and might not be implemented. For example, when the created scenebuffer in external memory exceeds memory limits one needs to start processing the sorted triangles before the end of frame is encountered. In this case, like in direct rasterization, the intermediate depth and stencil values cannot be discarded and need to be stored and retrieved from the external framebuffer.

- Hardware additions to the tilebuffers in the tile-based rasterizer that record whether the value for a pixel is valid and it is modified as discussed in Section 3.2.4. This adds two bits to the memory per pixel per stored parameter value (color, depth and stencil value).

The hardware rasterizer was slightly modified to reduce databack traffic as described in Section 3.2.4. In addition to a pixels stencil, depth, and color values GRAAL now also records if these values are valid and if they are modified. This increases the number of bits stored per pixel by 6. We assume the data sizes as used in the original GRAAL, with depth values of 24 bits, stencil values of 8 bit, and RGBA color values of 32 bits, are not changed. This results in a total storage requirement in on-chip memory of 70 bits per pixel in the tile.

$$gates = 6 \cdot size_{tile} \cdot 70 \tag{3.1}$$

The total hardware requirement for the memory elements inside GRAAL, called tilebuffers, is given by Equation (3.1). Where $size_{tile}$ is the number of pixels in a tile.

Next to the tile-based rasterization hardware we need to add a direct-sorting unit. The amount of gates that are required to implement the direct-sorting unit in hardware is difficult to predict without an actual design. However, we can use an indication of the required amount of gates based on the total amount of memory used in the hardware accelerator. We did this previously by estimating the hardware budget for a scenebuffer rasterization accelerator using a 32 × 32 on-chip memory implemented in embedded SRAM cells in Section 3.1. To apply the same estimation on the new graphics accelerator we need to indentify the storage requirements of the direct-sorting unit. Figure 3.5 identifies the major components of the direct sorting unit.

Every triangle introduced to the direct sorting unit is first presented to the overlap calculation (unit). In this unit comparisons are performed in order to construct the tilemask as discussed in Section 3.2.2. In order to determine overlap to a single tile we need to perform four comparisons. These are the comparisons given in Equations (2.7), (2.5), (2.8), and (2.6). Given a system with 300 tiles this is potentially a large unit, however when tile boundaries are chosen on discreet binary values (8, 16, 32, etc, pixels) the comparisons can be simplified by discarding the lower bits. Additionally, triangles will not be introduced to the direct sorting on unit every clock cycle (sending a triangle over the bus alone takes a number of clockcycles) and it is therefere not necessary to

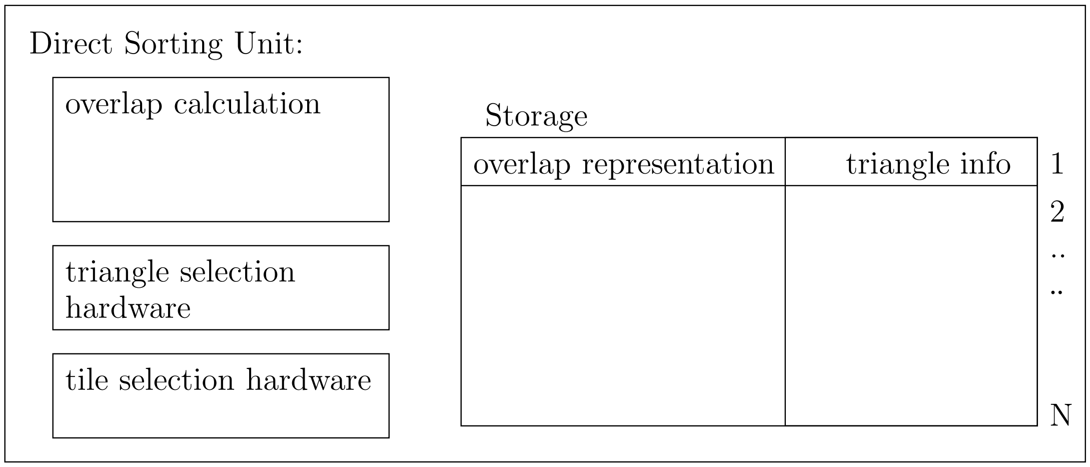


Figure 3.5: Direct Sorting unit major hardware components.

compute the entire tilemask at once. This means that comparators can be reused and the hardware required for generating the tilemask representation can be reduced.

The triangle selection unit selects triangles that can be send to the current tile. This is the simplest unit in the direct-sorting unit because it only needs to check one bit in the overlap representation in order to determine if the triangle should be send to the tile-based raterizer. The hardware required for this unit is therefore marginal.

The tile selection unit plays an important part in the direct sorting unit, but unfortunately its exact hardware proportion can not be determined without the actual design. The design is off course dependent on the choosen selection heuristic. We do note however that the tile selection policy is not a time critical operation. The processing of a triangle to a tile takes multiple clockcycles and multiple triangles can be send to the rasterizer during the processing of a tile. During all this time we can perform the computations required for the tile selection policy. Since it is not a time critical process we can use slow low-cost solutions that reuse hardware to minimize the strain on the hardware budget.

The largest and easiest to predict unit in the direct sorting unit is the storage unit. For every triangle we need to store its relevant parameters that need to be send to the rasterizer for processing. In this thesis we assume that the amount of data to store per triangle is fixed. By assuming this we ignore the fact that certain triangles might not always need all of the available parameters. If, for example, texture mapping is disabled the rasterization hardware disregards the texture parameters s and t and there is no need to store them in the direct-sorting unit.

Next to the standard triangle information the direct-sorting unit also needs to store the triangle to tile overlap representation as discussed in Section 3.2.2. Because we find the tilemask representation better suited to be implemented in hardware we supplement each triangle with a tilemask that is utilized in the sorting unit for the necessary triangle management. The amount of data required to store the tilemask is dependent on the number of tiles used to construct the screen.

$$gates = 6 \cdot \#_{triangles} \cdot (\#_{tiles} + size_{triangle}) \tag{3.2}$$

An estimate for the hardware requirement for the direct-sorting unit based on the required memory is given in Equation (3.2). Where $\#_{triangles}$ is the number of triangles that can be stored in the direct-sorting unit, in other words the size of the triangle window. The size of the tilemask is dependent on the number of tiles in the screen indicated by $\#_{tiles}$. The storage requirement for the standard triangle data is represented by $size_{triangle}$.

Combining Equations (3.1) and (3.2) results in an estimation of the required hardware for an on-chip hardware accelerator in a direct rasterization scenario. The total required hardware for memory elements in both the direct-sorting unit and the tilebuffers inside GRAAL can then be calculated via Equation (3.3).

$$gates = 6 \cdot (size_{tile} \cdot 70 + \#_{triangles} \cdot (\#_{tiles} + size_{triangle})) \tag{3.3}$$

Where $size_{tile}$ is the size of the tile-based rasterizer used for the actual rasterization and $\#_{tiles}$ is the number of tiles in the screen that the direct-sorting unit needs to process during direct the rasterization stage. In this computation $size_{triangle}$ is the storage size of a triangle and $\#_{triangles}$ is the total number of triangles we can store in the sorting unit at any one time.

3.4 Hierarchical Rasterization

The direct rasterization scenario and the scenebuffer rasterization scenario represent two ends of a spectrum from a software workload point of view. On one side of the spectrum scenebuffer rasterization utilizes the microprocessor for sorting and on the other side of the spectrum direct rasterization utilizes hardware for this sorting. Both approaches aim to minimize the databack traffic by sorting the triangles in a frame to the tiles they overlap.

Instead of looking at these two distinct ends of the spectrum, that either fully utilize the microprocessor for sorting or fully utilize hardware for this sorting we can look at solutions that balance the workload between hardware and software. Instead of performing the rasterization via either scenebuffer or direct rasterization we can use a hierarchical tile-based rasterization scenario that utilizes both software sorting as in the scenebuffer scenario and hardware sorting as in the direct scenario.

3.4.1 Basic Concept

The basic concept behind hierarchical scenebuffer and direct rasterization is to balance the workload between hardware and software. As mentioned in Section 3.1 a tile-based rasterization scenario for a 640×480 pixel screen resolution using 32×32 pixel tile-based rasterization hardware results in a system with 300 tiles. In scenebuffer rasterization this results in a high software workload related to sorting and in direct rasterization this results in a large tilemask, effectively reducing the number of triangles that can be stored locally. A smaller triangle window results in increased databack traffic because more rasterization passes are required in order to correctly render all the tiles in the image.

In a hierarchical rasterization scenario the scenebuffer sorting on the microprocessor divides the screen into sections, which are still to large to rasterize directly on a tile-based rasterizer like GRAAL. The triangles in these sections are then rasterized with direct rasterization using a direct-sorting unit.

Consider, for example, the case depicted in Figure 3.6 were a 640 $\times$ 480 pixel screen is divided in twelve 160 $\times$ 160 pixel sections using a scenebuffer sorting algorithm. These 160 $\times$ 160 pixel sections are then divided in twentyfive 32 $\times$ 32 pixel tiles using a direct-sorting algorithm. This reduces the software sorting from 300 to 12 tiles and reduces the tilemasks needed in hardware from 300 bits to 25 bits achieving a reduction of complexity in both software and hardware.

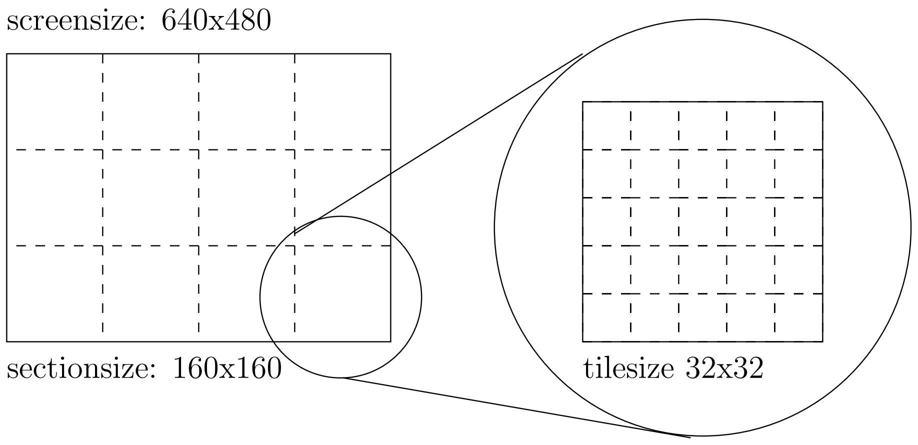


Figure 3.6: Screen division into sections and recursive section division into tiles.

For a system using a hierarchical rasterization scenario this means that the workload of the software in the system is reduced. Triangles generated by the geometry stage only need to be sorted to 12 tiles instead of the 300 tiles in a standalone scenebuffer rasterization scenario. The direct-sorting unit is also simplified, since it only renders a 160 $\times$ 160 pixel section at a time instead of the entire screen.

Figure 3.7 presents the block diagram for this hierarchical approach. From the software point of view, the size of a section can be made larger then the current maximum of 32 $\times$ 32 pixels dictated by the maximum tile-based rasterizer implementable within the 500k gate budget. By using a larger section size the complexity of the sorting on the microprocessor is reduced because the entire screen is constructed from less sections then in a scenebuffer rasterization scenario. Thus we require less computations on the microprocessor to compute the overlap via the equations presented in Section 2.5.4. Additionally, because the direct-sorting unit in hardware computes the tilemask in hardware the software side of the system can suffice with a simple bounding box test instead of a linear edge test. Finally, the datafront traffic decreases when the section size increases.

Similarly, from a hardware point of view the complexity of the direct-sorting unit is reduced. Instead of dividing the entire screen in tiles we only need to divide the

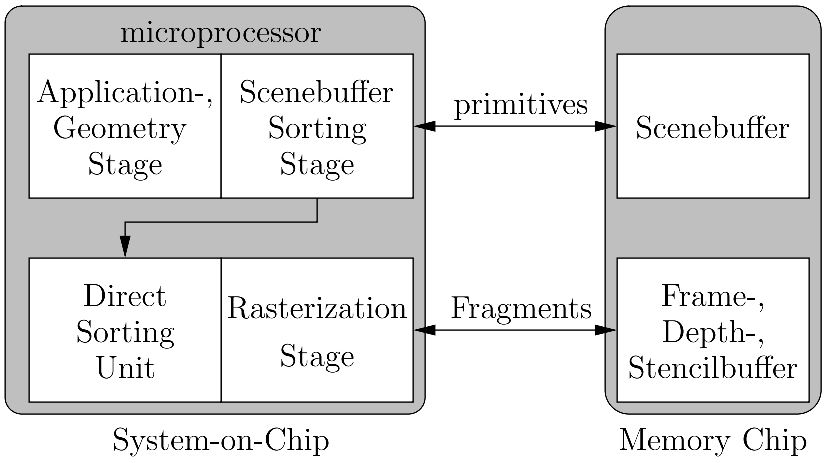


Figure 3.7: System diagram for hierarchical tile-based rasterization.

currently selected section in tiles. This reduced amount of tiles in the hardware leads to smaller tilemasks and reduces the memory required for storing a single triangle in hardware. This means that more triangles can be stored with the same hardware budget increasing the triangle window. Additionally, by sorting the incoming triangle stream from the geometry stage to sections the number of triangles per section is only a part of the entire triangle stream. Combining these two consequences, a larger triangle window and less total triangles per section, increases the effectiveness of the direct-sorting unit. This results in less tile switching and leads to reduced databack traffic when compared to the direct-rasterization scenario alone.

3.5 Summary

In this chapter we proposed direct rasterization, a rasterization scenario that moves the sorting in a tile-based rasterization scenario to hardware and processes triangles directly without a scenebuffer. The resulting hardware unit, called the direct-sorting unit, stores a limited number of triangles on-chip and supplies the rasterizer hardware with commands directly without prior sorting of all the instruction in a frame.

The direct-sorting unit first determines a suitable tile to process and signals the rasterization hardware which tile to process. During the rasterization of a tile all triangles present in the direct sorting unit that share overlap with the selected tile are send to the rasterization hardware for processing. When all of the triangles in the sorting unit that share overlap are processed a next tile is selected. During this process finished triangles are discarded and new triangles are introduced until all the triangles in the frame are processed.

Because the direct-sorting unit can only hold a limited number of triangles tiles cannot be considered finished until the final triangles of a frame are present in the hardware unit. This means intermediate values computated during rasterization need to be stored to external memory after a tile visitation. Any future visitations of the same tile requires these values to be retrieved from external memory. This results in an increase of databack traffic when compared to scenebuffer rasterization.

Together with conventional rasterization and the previously proposed scenebuffer tile-based rasterization we now have a number of graphics acceleration options which we aim to compare in effectivity from an external traffic point of view. In the following we refer to these options as:

- **Fullscreen**: Not explicitly mentioned but nevertheless the first option, is the conventional fullscreen rasterization scenario that utilizes an on-chip rasterizer and operates on an external fullscreen framebuffer. This system is characterized by a high traffic between the rasterizer and the external memory during rasterization because every computed fragment results in external traffic.
- **Scenebuffer**: From previous work we have the scenebuffer tile-based rasterization scenario. Before rasterization of a frame starts the software stores all instructions, either with or without pre-sorting, to a scenebuffer in external memory resulting in datafront traffic. However, the rasterization is performed with a minimum of databack traffic. On overal this results in a reduction of external traffic when compared to fullscreen rasterization at the cost of microprocessor workload.
- **Direct**: In direct rasterization we use a slightly modified version of the tile-based rasterizer used in scenebuffer rasterization. However, instead of using the microprocessor to sort the instructions required for a frame we propose a direct-sorting unit in hardware that supplies the rasterizer with triangles. This approach reduces the microprocessor workload when compared to scenebuffer rasterization and removes datafront traffic due to sorting. However, this approach requires revisiting some tiles, resulting in increased databack traffic when compared to scenebuffer rasterization.
- **Hierarchical**: In hierarchical rasterization we use both scenebuffer sorting on the microprocessor and a direct-sorting unit in hardware. This scenario allows a balance between software and hardware sorting solutions. The software sorting can be reduced by dividing the screen in sections larger then the previously proposed 32 $\times$ 32 pixels as used in a standalone scenebuffer rasterization scenario. This results in lower sorting workload and corresponding datafront traffic. Additionally, instead of processing every triangle, the triangles that must be processed for a section are filtered before being presented to the direct-sorting unit improving the effectivity and reducing tile visitations and associated databack traffic.

In the following chapter we describe the framework for the experiments we carried out in order to compare these different system implementations based on the total amount of external traffic generated by the rasterization stage of the graphics pipeline.

Experimental Platform 4

This chapter describes the simulator we built in order to compare the performance of several rasterization scenarios as seen from the external traffic point of view. In this traffic we mainly focus on the interaction with the external memory directly. Section 4.1 describes the benchmarks we utilized to emulate realistic mobile graphics applications running on a mobile platform. The application and geometry stages are simulated with an adaptation of existing software resources as described in Section 4.2. The external traffic generated during the rasterization stage is evaluated via a custom simulator written in standard C code. The implementation of this simulator is discussed in Section 4.3.

4.1 Benchmarks

In order to evaluate candidate rasterization scenarios for mobile devices they need to be applied for graphics applications that resemble the graphic workload of actual mobile platforms. In [6] a testbench suite was presented that provides seven OpenGL instruction traces from graphics applications that characterize typical workload for future mobile graphics devices.

From this benchmark suite we used only six benchmarks, since the seventh benchmark (Q3L) can be omitted since it is a low-resolution version of another benchmark (Q3H) in the suite. All benchmarks have a 640×480 resolution, which is higher then QCIF (176×144) and QVGA (320×240) resolution displays utilized in most mobile devices available today. However, future displays are likely to feature an increased resolution and furthermore, future mobile devices can be equipped with TV-OUT support enabling the use of an external screen for visualization. This means that in practice there is no real physical limit to the image resolution.

The components we selected from the testbench as input for our experiments are:

- Q3H: a trace generated from a popular OpenGL 3D first person shooter.
- TUX: a trace generated from a OpenGL 3D racing game for Linux.
- AWADvs: a benchmark suite taken from the SPECviewperf 6.1.4 benchmark suite.
- ANL: a trace from a VRML scene of a walk through the Austrian National Library.
- GRAZ: a trace from a VRML scene of a flyover of the area around Graz.
- DINO: a trace from a VRML scene representing a dinosaur.

Detailed information about the workload of the benchmarks is provided in [6] and is therefore omitted here. We only note that the Q3H benchmark is considered the most complex application of the benchmark suite since it uses a multitude of textures and enables color blending. Without color blending the color value of the generated fragments simply overwrites the color previously stored in the framebuffer. The previously stored color values are thus never read from the framebuffer. Since our proposed direct rasterization scenario sometimes requires reads from the external memory to retrieve previous color values the performance for this benchmark is critical.

The average triangle count per frame for every benchmark is above 1000 triangles. Because we want to quickly evaluate a large number of different system configurations, we limit the number of frames we process per benchmark to 10. Unless otherwise stated we run 10 frames for every benchmark, where we skip 25 frames between every frame in order to generate varying workload for the graphics system.

Because some of the benchmarks start with introductional frames that do not represent a realistic workload we use an offset for every benchmark. This means that we skip the first number of frames until we reach the actual core of the graphics application. The introduction menu in the TUX benchmark is the longest so we skip the first 800 frames forn this benchmark. For the Q3H benchmark we use an offset of 500 frames and for the other benchmarks we use a default offset of 200 frames. Since the ANL benchmark featured some frames with a low triangle count around frame 220 we used an offset of 300 frames for this benchmark.

4.2 Application and Geometry Stages

We aim to compare the four rasterization scenarios that were summarized in Section 3.5 from the point of view of the total amount of external traffic they generate. This cannot be achieved without realistic input to the rasterization stage, normally this input is generated by the application and geometry stage of the graphics pipeline. We assume that the implementation of the application and geometry stage is the same for all four rasterization scenarios. The exact implementation of the application and geometry stage is therefore not relevant for this research. However, we do require the execution of these two stages in order to acquire the input for the rasterization stage.

This is achieved by preprocessing the application and geometry stages for the benchmarks introduced in the previous section. From previous work [2] we have a modified library, based on the freely available Mesa library [29], that performs the computations of the application and geometry stages and produces a trace file with OpenGL commands ready to be send to a rasterizer. The actual input of the rasterization simulation then becomes an instruction stream corresponding to the benchmark applications. In this instruction stream we have all instructions needed to rasterize all application frames, like state modification instructions, clear instructions, swapbuffer instructions, and off course triangle instructions.

The modified Mesa library also performs a graphics technique called culling, a process which removes triangles that are orientated backwards to the observer. In some graphics applications the backside of a triangle is never visible, for instance the backside of triangles that are used to construct a sphere. Triangles orientated with the backside towards the observer, which are to the inside of the sphere, are then discarded by the culling process reducing the workload in the rasterization stage.

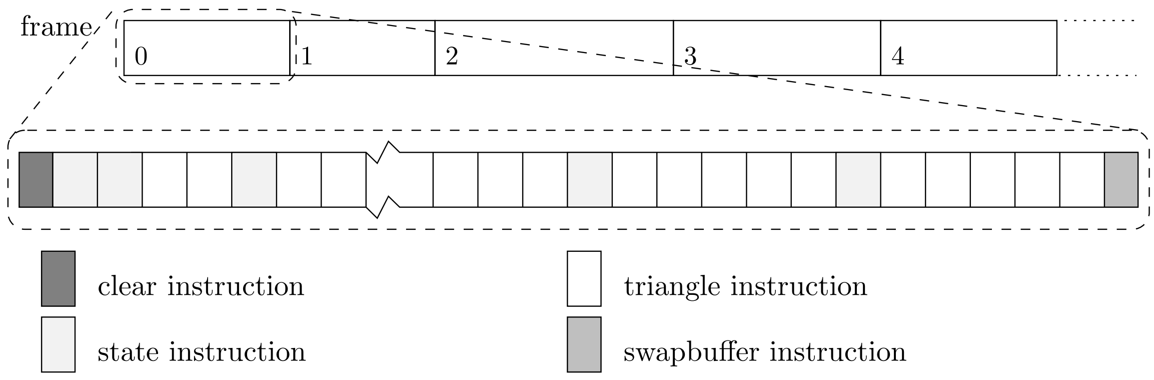


Figure 4.1: Instruction stream generated by the modified mesa3d library.

Figure 4.1 provides a visual representation of the input stream as generated by the modified mesa3d library in the form of a trace file. All instructions of all the frames are concatenated into a single stream for the running application. All applications in the benchmark suite start their frames with a clear instruction and end them with a swapbuffer instruction. The instruction stream also includes state instructions, which modify the rasterization settings in the hardware, or in other words the hardware state. A state instruction influences all triangles after the state instruction and in general every modification of the hardware state is followed by several triangles that are all rasterized with the same state. It is important to note that the hardware state remains valid for the next frames and must be therefore always actively maintained.

4.3 Rasterization Stage Simulation

A rasterization simulator written in SystemC was previously developed in the GRAAL project to evaluate design considerations inside tile-based rasterization hardware. This simulator computed the exact transfer of data on a register level inside graphics acceleration hardware on a mobile device. As it is a low-level simulator computing the data transfers inside the hardware is a time consuming process. Because the simulator uses data formats that cannot be easily represented on a regular CPU the total simulation time for a single system configuration can exceed several hours even for only a single (!) frame on a standard x86 personal computer.

Furthermore, the previous simulator is not easily reconfigurable for the simulation of a tile-based rasterizer with any chosen tile size. If one wants to simulate a system with a 8×8 pixel tilebuffer one needs to completely rewrite the currently written 32×16 pixel tilebuffer module and modify any modules communicating to the tilebuffer.

In our work we aim to compare the external traffic in the rasterization scenarios summarized in Section 3.5 with varying tile sizes, different scenebuffer-management algorithms (scenebuffer, hierarchical rasterization), and different tile-selection policies (direct, hierarchical rasterization). To this end we require a quick simulation of a large range of different system configurations. The rasterization simulator utilized in the previous work is simply not suitable for this task. Therefore, we decided the create a new tile-based rasterization simulator that is able to provide external traffic figures quickly.

This section describes the new simulator suite created to quickly evaluate 3D graphics system designs that are based on a mobile microprocessor with an on-chip tile-based rasterizer. The goal of this simulator suite is to quickly produce external traffic figures for the rasterization phase fo a number of different tile-based rasterization scenarios. This allows for the fast evaluation of the effects of changing certain system parameters from an external traffic point of view. Because external traffic is a major source of power dissipation the external traffic figures are used as an indication for the total power consumption of the graphics system.

The simulator is created with the following system topologies in mind:

- The application and geometry stages of the graphics pipeline are implemented in software running on the microprocessor. Due to the limited hardware budget for mobile graphics on the microprocessor chip all available hardware is utilized for hardware acceleration of the rasterization stage of the graphics pipeline only. The application and geometry stages are highly dependent on the type of application and require high-level computations like matrix multiplications and are thus best performed in software making this a likely assumption.

- A pixels color is represented with four components; red, green, blue, and alpha, where each separate component requires 8 bits storage. A depth value requires 24 bits storage and finally a stencil value requires 8 bits storage. Therefore to store all parameters for a single pixel in the framebuffer we require 64 bits of storage.

- The fullscreen framebuffer containing the image depth, stencil, and color values is to large to be implemented on the microprocessor chip and it is therefore stored on a separate chip. This was discussed in Section 2.2. The access to this external chip consumes a lot of power thus, the main goal of tile-based rasterization is to limit the number accesses to this fullscreen framebuffer during the rasterization stage.

- For tile-based rasterization scenarios hardware acceleration is achieved via a tile-based rasterizer with a limited-size on-chip memory for storing intermediate stencil, depth, and color values for a single tile. For a fullscreen rasterization scenario a conventional rasterizer is assumed that requires one access to the external memory for every interaction (read or write) with the framebuffer.

- Similar to the fullscreen framebuffer, the scenebuffer, a memory buffer that contains all instructions needed to rasterize a single frame, is to large to be implemented on the microprocessor chip and it is therefore stored on a separate memory chip. Since a single frame can consist of over one thousand triangles, each requiring approximately 55 bytes as explained in Section 4.3.1, the size of the scenebuffer can quickly exceed the gate budget for on-chip storage.

Based on these assumptions a simulator suite was written in normal C code with the main goal of measuring the memory references to off-chip memory during rasterization. The input to this suite is a stream of OpenGL instructions as generated by the application and geometry stage library as discussed in Section 4.2. Where possible the rasterization is simplified by omitting stages and computations that do not directly influence the amount of external traffic generated by the system. The simulator suite itself is split in two independent parts because accesses to the external memory can be split in two categories: (i) datafront traffic, the traffic related to sending rasterization instructions to the graphics accelerator; and (ii) databack traffic, the traffic to and from the external framebuffer during the rasterization computations.

4.3.1 Measuring Datafront Traffic

The datafront traffic simulator computes the number of memory accesses required during the creation of the scenebuffer of a selected graphics application. Additionally, the simulator computes the number of memory accesses required to retrieve instructions from the scenebuffer during the rasterization of the benchmark application. It is important to note that the actual rasterization is not required in order to compute the datafront traffic during the rasterization and this simulator can therefore be run standalone.

The simulator can be used to calculate the datafront traffic in a fullscreen -, scenebuffer -, direct -, and a hierarchical - rasterization scenario. If scenebuffer sorting is utilized in the system the simulator can be set to simulate either of the four scenebuffer sorting algorithms which were presented from previous work [8] and briefly discussed in Section 2.5.3, namely **two_step**, **two_step_let**, **sort**, and **sort_let**.

The purpose of the datafront simulator is to quickly evaluate the number of external memory references required to create and use the scenebuffer based on the following input parameters:

- The fullscreen size of the application. When not indicated otherwise all simulations are run with a 640×480 screen resolution, which is a likely screen resolution for mobile devices and the standard resolution of the benchmarks in the chosen benchmark suite.

- The screen division into tiles (scenebuffer rasterization) or into sections (hierarchical rasterization). If the screen is divided into larger parts, resulting in a lower number of tiles or sections, less data is duplicated resulting in a lower amount of datafront traffic.

- The chosen scenebuffer sorting algorithm as presented in Section 2.5.3. For direct rasterization and fullscreen rasterization no scenebuffer sorting is performed and every instruction is only stored and retrieved from external memory once.

The most important aspects about the implementation of the datafront traffic simulator are discussed briefly in the following subsections.

4.3.1.1 Scenebuffer Management Algorithms

The datafront traffic calculation is based on the scenebuffer management algorithms as proposed in [8]. The choice of scenebuffer management algorithm determines how often data are send to -or retrieved from- the scenebuffer in external memory.

In the **two_step** variants of these algorithms every instruction is stored in a single scenebuffer only once, which is consequently read for every tile. This means that every instruction results in a number of writes to the external memory that can be computed via Equation (4.1).

$$writes = size_{opc} + size_{parameters} \tag{4.1}$$

where $size_{opc}$ is the size of the opcode, in bytes, used to distinguish all different rasterization instructions and $size_{parameters}$ is the size of each individual instructions parameters, in bytes, which is different for every type of rasterization instruction. Basically the number of writes in a **two_step** algorithm equals the total size of the instruction stream since every instruction is only written to the scenebuffer once. For triangles however the triangles bounding box is stored in the scenebuffer adding a number of bits to the $size_{parameters}$ variable. The size of the bounding box is dependent on the number of tiles in the screen as explained in one of the following sections.

Because the single scenebuffer is used for every tile all instructions have to be read for every tile. Fortunately, triangle instructions, which make up the majority of the instructions, are only partly read for every tile. The complete triangle data are only read from the external memory when the bounding box indicates that a triangle overlaps with the current tile. The number of reads per non-triangle instruction is computed via Equation (4.2) while the reads per triangle are computed via Equation (4.3).

$$reads_{nontriangle} = \#_{tiles} \cdot (size_{opc} + size_{parameters}) \tag{4.2}$$

$$\begin{aligned} reads_{triangle} = \#_{tiles} \cdot (size_{opc} + size_{bounding-box}) + \\ \#_{overlap} \cdot size_{parameters} \end{aligned} \tag{4.3}$$

Where $\#_{tiles}$ is the number of tiles (scenebuffer rasterization) or sections (hierarchical rasterization) the screen is divided in by the software sorting. The coefficient $\#_{overlap}$ indicates the total tiles the triangle overlaps computed via the bounding box test. Reducing the datafront traffic with a more accurate linear edge test is not possible in this case, because this test requires the exact screen coordinates of the triangles, which are stored in the scenebuffer. Retrieving these to do a more accurate overlap test results in external traffic and hence cannot reduce the external traffic.

The datafront traffic in a scenebuffer rasterization scenario using a **sort** algorithm is fundamentally different to the datafront in a **two_step** algorithm. When a sorting algorithm is used all the instructions are send to individual bins in external memory, representing a single tile. An instruction is send to a bin for each tile it influences. This means that instructions that influence the entire screen, like state instructions, need to be send to every bin in external memory. For non-triangle instructions the number of reads is equal to the number of writes to the external memory and can be computed via Equation (4.4)

$$reads_{nontriangle} = writes_{nontriangle} = \#_{tiles} \cdot (size_{opc} + size_{parameters}) \tag{4.4}$$

Triangle instructions are stored only in the bins that represent tiles the triangle overlaps. Fortunately, it is not necessary to duplicate the entire triangle data for every bin. Instead, the complete triangle data are stored to external memory once and a memory reference to this location is stored in each bin the triangle is present in. Equation (4.5) is used to compute the number of writes to external memory in order to store a triangle in the scenebuffer, while Equation (4.6) is used to compute the number of reads from the external memory, for retrieving a single triangle instruction from the scenebuffer.

$$writes_{triangle} = size_{parameters} + \#_{overlap} \cdot (size_{opc} + size_{memoryreference}) \tag{4.5}$$

$$reads_{triangle} = \#_{overlap} \cdot (size_{opc} + size_{memoryreference} + size_{parameters}) \tag{4.6}$$

In contrast to the $\#_{overlap}$ in the **two_step** algorithm the triangle to tile overlap in the **sort** algorithm can be computed via either a bounding box test or a linear edge test because the sorting is done before the instructions are stored to the external memory. Using a more exact overlap test can therefore influence the amount of datafront traffic in a rasterization scenario using a **sort** algorithm.

4.3.1.2 Instruction Size

In order to count the number of accesses to the external memory it is important to know how large each rasterization instruction is in bytes. We assume that each instruction can be disinguished by an one byte opcode. The different instructions are then supplemented with additional parameters if required. When determining the size of an instruction, either state or triangle instruction, the sizes of the parameters are assumed as given in Table 4.1.

For example, the state instruction `glClearColor` is constructed with a 1 byte opcode and 4 parameter bytes representing the RGBA color that is used when the screen is cleared. Similarly, a standard triangle instruction consists of an opcode and three vertices, each in turn consisting of two screen coordinates x and y, depth coordinate z, homogeneous coordinate w, four color values R, G, B, alpha value A, and two texture coordinates s and t. In total a standard complete triangle instruction consists of 55 bytes.

4.3.1.3 Bounding Box Size

As mentioned before the **two_step** algorithm stores a bounding box in the scenebuffer for each triangles instruction. A bounding box is created from four values: MIN_x,

parameter type	size (byte)
screen coordinate	2
depth coordinate	3
homogeneous coordinate	3
color value	1
alpha value	1
stencil value	1
texture coordinates	2
bounding box	variable
memory reference	4

Table 4.1: Assumed parameter storage sizes.

MAX_x, MIN_y, and MAX_y and its size is dependent on the number of tiles in the screen. If a lower number of tiles are used to construct the screen the bounding box can be represented with less bits. The size of the bounding box in bits is calculated via Equation (4.7).

$$size_{box} = 2 \cdot log_2(tiles_x) + 2 \cdot log_2(tiles_y) \tag{4.7}$$

Where $tiles_x$ and $tiles_y$ are the number of tiles in the horizontal and vertical direction, respectively. We assume that the external memory is byte addressable and therefore the bounding box size only increases at discrete byte boundaries. Table 4.2 lists the bounding box sizes for a number of screen divisions. These sizes are used during the simulations.

section/tile size (pixels)	tiles/sections	bounding box size (byte)
8x8	80x60	4
16x8	40x60	3
32x32	20x15	3
40x30	16x16	2
160x120	4x4	1

Table 4.2: Bounding box size for 640×480 applications for different screen divisions.

4.3.1.4 Triangle Size

Unfortunately the storage size of a triangle is not fixed for every triangle, but it is determined by the current state settings. A triangle is always defined by three vertices, where each vertex is always defined by two screen coordinates (x and y). When OpenGL depth testing is disabled no depth coordinates (z) are required for any of the vertices. Similarly, certain OpenGL settings do not require the triangles to be supplemented with color values (R, G, B, and A) or texture coordinates (s and t). The actual size of a triangle is therefore not always 55 bytes as computed in the previous section, but it is determined at runtime based on the current settings of the OpenGL state.

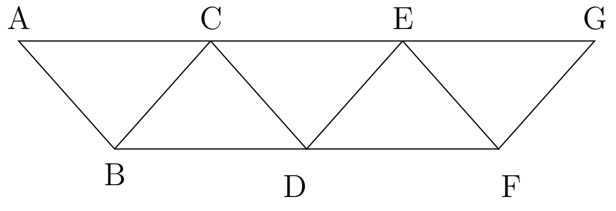


Figure 4.2: Triangle Mesh.

Additionally, since triangles are used to construct complex polygons like triangle meshes a lot of the vertices inside the polygon are shared among consecutive triangles. Consider the triangle mesh depicted in Figure 4.2. The first triangle ABC is defined by vertices A, B, and C. In order to define the next triangle BCD only one new vertex D needs to be introduced. Triangles ABC and BCD therefore share two of their vertices. This is true for most polygons in computer graphics. By storing shared vertices in external memory for several triangles instead of storing all vertices separate for every triangle the datafront traffic can be reduced. Vertex sharing was not yet considered in [7], but it can significantly reduce triangle data written to the external memory.

Vertex sharing is implemented in the datafront simulator be maintaining a short list of ten vertices during simulation. If a triangle uses one of the vertices in the list a memory reference is added to the datafront traffic computation instead of a full vertex write. The list is maintained using a FIFO algorithm, where every new vertex replaces the vertex that was introduced first.

4.3.1.5 Simulator Implementation

The implementation of the datafront traffic simulator is rather straightforward. The input, the instruction stream as generated by the modified mesa3d library, is read instruction per instruction. A counter is incremented everytime a `glSwapBuffer` instruction is processed indicating the current frame has finished. Because the simulator for the databack traffic, discussed in Section 4.3.2 hereafter, performs the actual rasterization for only a few number of frames for each benchmark the datafront simulator can similarly be executed to process only a select number of frames.

The number of external memory references in the datafront is dependent on the type of instruction. For all non-triangle instructions, mostly state instruction that influence all tiles in the screen, the number of reads and writes to and from the scenebuffer in external memory is always fixed. Therefore the number of memory references for these instruction can be precomputed using Equations (4.1) through (4.6) before simulation. When a non-triangle instruction is processed during the running of the benchmark this precomputed memory reference size is added to the running total of memory references. This running total indicates the total amount of datafront traffic.

The number of accesses to external memory and the resulting datafront traffic related to triangle instructions is variable for each triangle instruction. Therefore, when a triangle instruction is processed during the running of a benchmark the exact number of accesses to the external memory are computed by performing the following steps:

- The computation of the triangle to tile overlap. Based on the selected scenebuffer management algorithm, this is implemented via either a bounding box test or a linear edge test as detailed in Section 2.5.4.
- The computation of the size of a vertex. This is done by examining the current OpenGL settings in order to determine which parameters are used by the rasterization hardware. Parameters that are not currently used, do not need to be stored along with the triangle instruction in the scenebuffer. The assumed sizes for all parameters are listed in Table 4.1.
- The determination of shared vertices. The vertices of the current triangle instruction are compared to a list of ten previously introduced vertices. If one of the vertices of the current triangle is inside this list the size of a memory reference is added to the running total of datafront traffic. If a new vertex is introduced the size of the entire vertex is added to this running total.[1]

Based on the results of the executed steps and the chosen scenebuffer management algorithm the total number of memory accesses related to the triangle instructions are computed and added to the running total of memory accesses.

4.3.2 Measuring Databack Traffic

The databack simulator computes the number of external memory accesses to the framebuffer required during the rasterization of a chosen benchmark. The databack simulator is more complex then the datafront simulator since this simulator needs to count the actual memory references during rasterization. This cannot be done without performing the rasterization computations. We refer readers unfamiliar with these computations to Appendix A.

Fortunately, no databack traffic simulation is required for the scenebuffer sorting scenario. In this scenario each pixel is only stored to the external framebuffer once, resulting in the minimal databack traffic where only the final color values are stored and all stencil and depth values are discarded. Therefore, the external memory references for a scenebuffer scenario are always fixed depending only on the fullscreen resolution. Since all our benchmarks have a resolution of 640×480 the databack traffic in this scenario is calculated via Equation (4.8). We assume that every color is represented with 32 bits, 1 byte for each component in RGBA. The total databack traffic for the scenebuffer tile-based rasterization scenario is 1.23 MB per frame. This is unrelated to the type of application and thus equal for every benchmark.

[1]Maintaining a list of ten vertices was empirically determined. Experimenting with larger vertex lists did not result in a significant decrease of the databack traffic.

$$640 \cdot 480 \cdot bits_{color} \tag{4.8}$$

For the other rasterization scenarios the databack simulator performs the actual rasterization computations. In fullscreen rasterization every rasterized triangle results in a number of fragments. For each fragment we count a access to the external memory. For direct rasterization and hierarchical rasterization each generated fragment results in an update to the tilebuffers that are inside the rasterization hardware. At a tile switch the accesses to the external memory are computed. The simulator mimics the system topology and computes the number of memory references to the external memory. The following parameters can be set in order to simulate different rasterization scenarios and different hardware implementations:

- The fullscreen size of the application. When not indicated otherwise all simulations are run with a 640 $\times$ 480 screen resolution.
- The screen division into sections. The division of the fullscreen resolution into sections determines the amount of workload for the software running on the microprocessor. If the screen is not divided, or in other words divided into a single section, the simulator calculates the databack traffic for a direct rasterization scenario. In this case no sorting is performed on the microprocessor.
- The size of a tile. Using a smaller tile size means that less temporary pixel values are stored on-chip, reducing the required hardware for storing these data values on-chip. On the other side it increases the amount tiles needed to create the screen (or section in a hierarchical rasterization scenario).
- The number of triangles stored on chip. This parameter directly influences the triangle window used by the direct-sorting unit. However, the amount of triangles that can be stored is limited depending on the available hardware budget.
- The tile-selection policy. The direct-sorting unit needs to implement a heuristic for choosing the next tile. All four selection policies as discussed in Section 3.2.3 are implemented and can be selected before simulation.
- Whether bounding box testing or linear edge testing is used to construct the triangle-to-tile overlap representation inside the direct-sorting unit for direct and hierarchical rasterization.

4.3.2.1 Fragment Sizes

The databack traffic in a direct rasterization scenario and a hierarchical rasterization scenario consits of storing temporary fragment values to the external memory and retrieving temporary fragment values from this external memory. After a tile is visited all the intermediate color, stencil, and depth values that were modified during the tile visitation are stored to the external memory. Table 4.3 lists the size of the fragment values in the databack traffic. Every pixel is represented by three color values, one alpha value, one stencil value, and one depth value.

parameter type	size (byte)
depth coordinate	3
color value	1
alpha value	1
stencil value	1

Table 4.3: Fragment value sizes in databack traffic in bytes.

4.3.2.2 Simulator Implementation

The databack simulator is build in two stages. The first stage implements the scenebuffer sorting algorithm, implemented via the simplest sorting algorithm, **direct**, because it is the simplest to implement. No references to the external memory are registered during this stage since the datafront is calculated with the separate simulator described in Section 4.3.1. For simulating a hierarchical rasterization scenario this stage divides the fullscreen resolution into smaller sections. Each section is then sequentially processed by the direct-sorting unit simulator.

The second stage of the simulator implements the functional behavior of the direct-sorting unit and the rasterization unit. The databack traffic simulator is actually a configurable software model of a hierarchical rasterization scenario. However, by varying parameters also the databack traffic in a direct rasterization scenario can be calculated. The databack traffic for a fullscreen rasterization scenario is maintained in parallel during every simulation by counting the number of generated fragments.

Figure 4.3 illustrates the construction of the databack traffic simulator. The first stage, implementing the scenebuffer sorting, filters the single scenebuffer for the section currently being processed. All instructions relevant to the current section are send to the software model of the direct-sorting unit. Instructions that do not influence the current section, mainly triangle instructions that reside in other sections, are not send to the direct-sorting unit software model for the current section.

The second stage, processes the instructions for every sections one by one, where every section is divided into a number of tiles. The direct-sorting unit processes the current section by issuing rasterization commands to the rasterization unit in a tile-based fashion. The software model of the direct-sorting unit is configured before simulation by choosing one of the tile selection policies discussed in Section 3.2.3 and a maximum number of stored triangles. The rasterization unit is configured to the chosen tile size and the rasterization computations are implemented similarly to the GRAAL pipeline as described in Section 2.4.

4.3.2.3 Direct-Sorting Unit Model

The model for the direct-sorting unit again consists of two parts. The first part is a triangle buffer that can be set to contain an arbitrary number of triangles prior to simulation. The second part is a management unit that uses the data contained in the triangle buffer to determine the best tile to process and selects triangles to process.

When a triangle is introduced in the direct-sorting unit the tilemask is created using

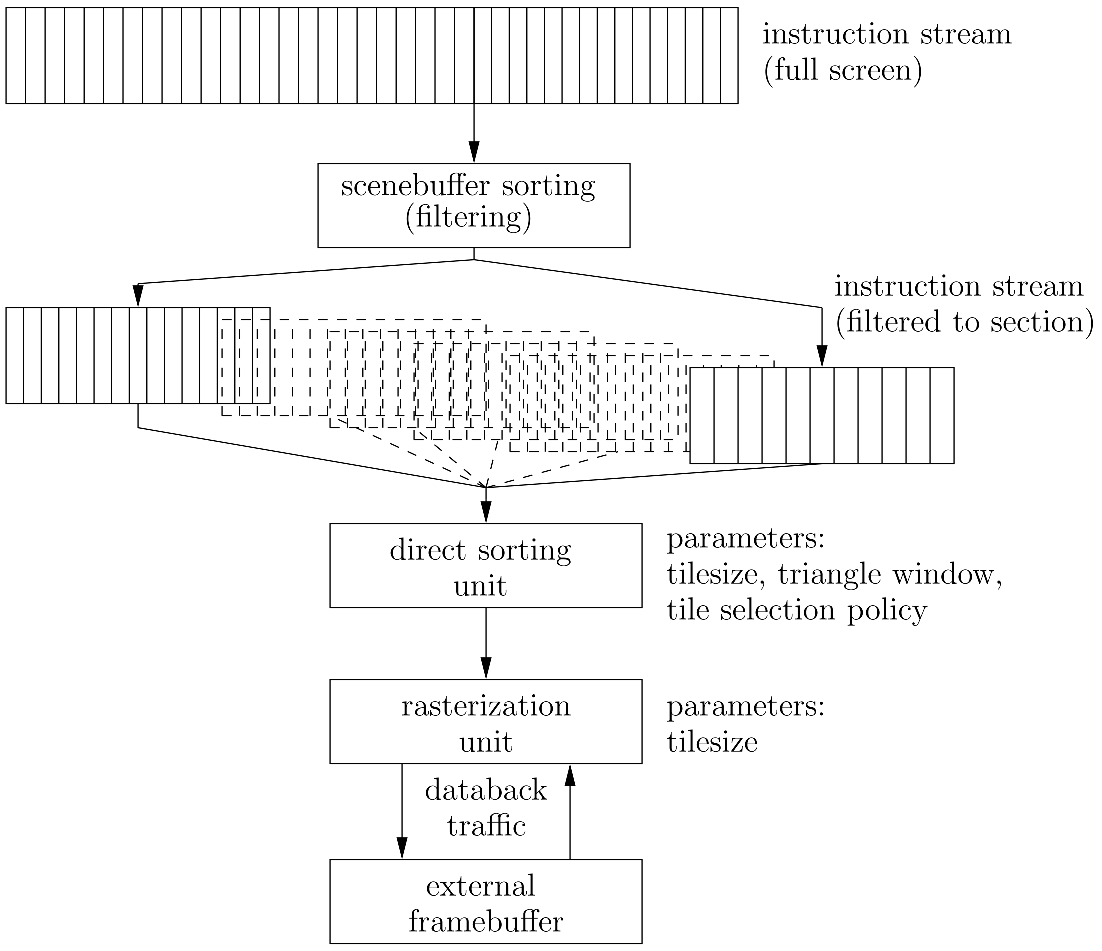


Figure 4.3: Construction of the databack traffic simulator.

either a bounding box test or a linear edge test, depending on the selected parameter prior to the simulation. This tilemask is stored together with the triangle data inside the triangle buffer and used as a basis for tile and triangle selection. The management unit is able to access the triangle tilemasks and uses the information in the tilemasks to select a tile to process via one of the proposed tile selection heuristics in Section 3.2.3. After this triangle, clear, and state instructions are send to the rasterization unit for the current tile. After all instructions for the current tile are depleted a new tile is selected for processing.

The tilemask is an important data structure in the direct-sorting unit model. It serves as a basis for the tile selection policy and it is used to maintain the rasterization status of triangles. Each time a triangle is send to the rasterization unit its tilemask is updated by setting the bit representing the current tile to zero. When a triangle tilemask contains only zeroes, indicating the triangle was processed for all the tiles it was present in, the triangle can be discarded and a next triangle from the instruction stream is retrieved.

One important aspect of the software model of the direct-sorting unit is that for all tiles the triangles are always processed in the order in which they were introduced in the direct-sorting unit. This means that all computations to calculate the final color for a single pixel are exactly identical and computed in the same order. This is also the case in fullscreen rasterization and scenebuffer rasterization. Consequently, the number of texture fetches per pixel does not differ between rasterization scenarios. This is a fundamental observation that allows the texture computations to be dropped from the databack simulation.

For this reason the calculation of the texture fetches is omitted from the simulator entirely. The software implementation of the rasterization unit model is therefore simplified by omitting texture calculations. This does not affect the correct computations of the databack traffic resulting from data transfers of fragments between the rasterization unit and the external memory. Unfortunately, it does mean that the calculated colors are no longer correct and only the depth values can be used for visual inspection of the rasterization result.

4.3.2.4 Rasterization Unit Model

The software model of the rastization unit is implemented in a simple manner. The most complex computations, rasterizing the triangle via the edge functions Equations (2.9) through (2.11), are directly performed when a triangle is processed by the software model. The results of these computations, the exact pixel locations that are influenced by the triangle are stored in a two dimensional array representing the tile. This array models the logic-enhanced memory as introduced in [17].

```
Rasterize (Triangle T)
    perform_edge_functions(T)
    while (Fragment F = fragments_left())
    {
        compute_fragment_attributes(T, F)
        if (not alpha_test_passed(F)) break;
        if (not stencil_depth_test_passed(F)) break;
        update_framebuffer(F)
    {
```

The code depicted above is a simplistic representation of the rasterizer software model. Once a triangle is considered for rasterization the software model immediately computes the pixels inside the triangle with the *perform_edge_functions*() function. This function computes all pixels in a tile that are inside the introduced triangle via edge functions [31]. The results are stored in the array mentioned above. After the pixels inside the triangle are identified the array is searched for pixels inside the current triangle in the function *fragments_left*(). This function sets the location of a fragment in its X and Y coordinate or it signals that there are no pixels left inside the triangle for the current tile.

The fragment parameters are then calculated by function *compute_fragment_attributes*() via Equation (A.15), which is discussed in Appendix A. In short each pixels parameter value is linearly interpolated based on the parameter values of the triangles vertices. As mentioned in the previous section no texture mapping is performed by the software model of the rasterization unit. Every fragment that would be colored by a texture mapping computation is now set to plain white.

Similar to the GRAAL tile-based rasterization hardware designed in previous work [18], the rasterization simulator performs three OpenGL functions in order. These are the alpha test, stencil and depth test, and blending and logical operations. The first two functions determine if the fragment is visible and consequently influences the framebuffer and the last function determines the exact type of update to the framebuffer. These three tests are the basic tests that are always implemented in OpenGL compliant hardware. For more detail about these functions the reader is referred to Appendix A.

4.3.2.5 Counting External Memory Accesses

No exact model for the on-chip storage is implemented in the simulator and therefore the stencil, depth, and color values are simply stored in a section sized buffer for depth, stencil, and color values, respectively. These temporary data values can be read and written directly. This is off course not as in a real system, but the purpose of the simulator is to track the number of required accesses (read and write) to the external memory and the exact values can then easily be stored in a larger buffer.

To record wheter external memory accesses are required we created tile sized buffers containing the valid and modified bits as mentioned in Section 3.2.4 in software. These buffers are used to determine the databack traffic in a hierarchical and direct rasterization scenario. Whenever a data value is read the valid buffer is examined whether or not an access to the external memory is required or if the currently stored value in the buffer is already available. If the value is already valid, either by a previous write or read, no databack traffic is necessary.

In our proposed hierarchical and direct rasterization scenarios no data values (depth, stencil and color values) are written in the external memory until a tile visit is complete. A tile-sized modified buffer is used to determine the exact amount of databack traffic during the tile switch. Only the modified pixels need to be send to the framebuffer in external memory, while all pixels that remain unchanged do not need to be send and therefore do not result in databack traffic. After all triangles for a tile are processed the content of this modified buffer is examined in order to determine the number of writes to the external framebuffer for this tile visit.

In the fullscreen rasterization scenario no data values are stored on-chip and therefore all reads of previously stored stencil, depth, and color values result in fetches from the external memory and are thus counted as databack traffic. Similarly, all fragment values that pass the stencil and depth test result in databack traffic to the external memory and are counted during rasterization.

4.3.2.6 Visual Inspection

The final stencil, depth, and color values for a section can be written to an image file for visual conformation of the correct rasterization of the frame. Because the texture mapping is omitted the color of the primitives is plain white for textured triangles. This means the color image can sometimes not be used to verify the correct rasterization. Fortunately, the depth values always allow for visual conformation of the rasterization stage. Figure 4.4 presents a typical depth image generated during the simulation of the dino benchmark. A lighter shade of white indicates that the pixel is closer to the observer, while black indicates background pixels.

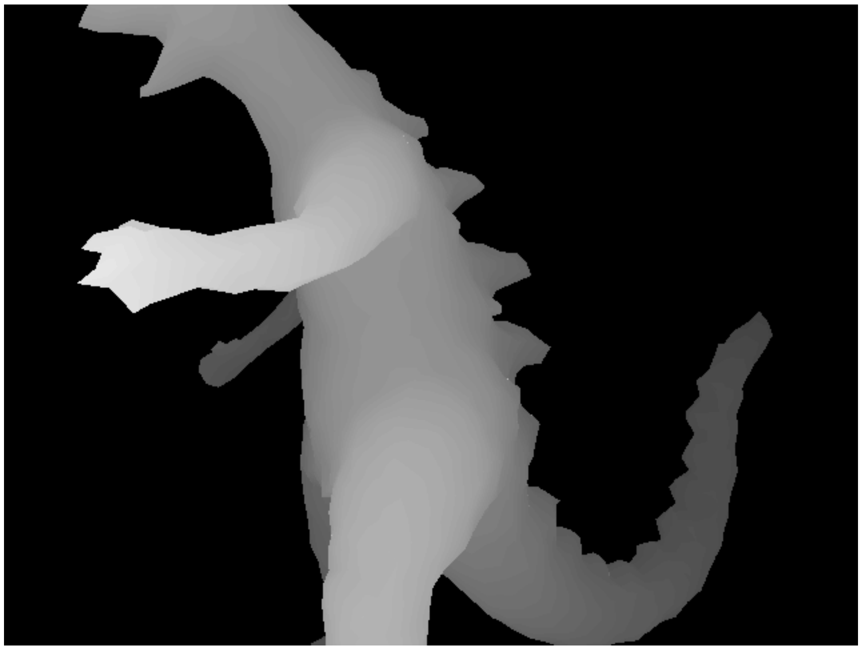


Figure 4.4: Depth image exported during rasterization simulation of the dino benchmark.

In the following chapter we present the results of our experiments. We start by simulation results of scenebuffer rasterization, followed by direct rasterization and after comparing both tile-based rasterization scenarios we present simulation results on a hierarchical rasterization approach that benefits from advantages of both scenebuffer and direct rasterization.

Experimental Results 5

In Chapter 4 we presented a framework for determining the amount of external traffic in different tile-based rasterization scenarios. In this chapter we aim to compare the external traffic generated by fullscreen -, scenebuffer -, direct -, and hierarchical rasterization. We present our results acquired by performing rasterization simulations for the benchmark suite presented in previous work [6].

In Section 5.1 we first present simulation results that illustrate the behavior of the datafront portion of the external traffic. This traffic is a critical component of the total traffic in a scenebuffer rasterization implementation. Similarly, Section 5.2 presents simulation results that capture the behavior of the databack portion of the external traffic. This part of the traffic is critical in the evaluation of the fullscreen and direct rasterization implementations. Section 5.3 thereafter combines the traffic figures from the first two sections to compare the previously introduced scenebuffer rasterization scenario with our newly proposed direct rasterization scenario. Finally, Section 5.4 presents simulation results for a hierarchical rasterization scenario, a scenario that utilizes the best properties of both scenebuffer and direct rasterization.

5.1 Evaluating Datafront Traffic

Previous work [8] proposed several scenebuffer sorting algorithms to be used in conjunction with the scenebuffer tile-based rasterization scenario. These algorithms, described in Section 2.5.3, determine the amount of external traffic in the datafront generated during rasterization. The datafront is affected by the chosen scenebuffer sorting algorithm and the amount of tiles used to construct the entire screen. Basically, in a solution with a low number of tiles (larger tiles), less rasterization primitives are duplicated for different tiles and consequently the datafront is less when compared to solutions that construct the screen from more tiles (smaller tiles). In short, when the triangle to tile overlap is reduced the amount of external traffic in the datafront is reduced.

This section presents traffic simulation results for the six benchmarks presented in Section 4.1. The computation of the datafront is done via the approach presented in Section 4.3.1. For each benchmark we process 10 frames, using an offset large enough to skip introduction frames and skipping 25 frames after processing each frame in order to provide an average datafront value for the benchmark in a short amount of time.

The amount of external traffic in the datafront for the processed frames in a fullscreen rasterization scenario, which is equal to the datafront in a direct rasterization scenario, is listed in Table 5.1. This is the datafront generated when no sorting is performed in software and each produced primitive is simple send to the graphics accelerator. It is measured by calculating the memory references required when each rasterization instruction is only stored to the external memory once. In short a single write to the external memory and a single read from the external memory for every instruction.

benchmark	ANL	DINO	GRAZ	AWAD	TUX	Q3H
datafront (kB)	161.9	294.7	91.4	376.1	85.1	193.2

Table 5.1: Avarage external traffic in the datafront of a fullscreen rasterization scenario for the selected frames.(Equal to the datafront of a direct rasterization scenario.)

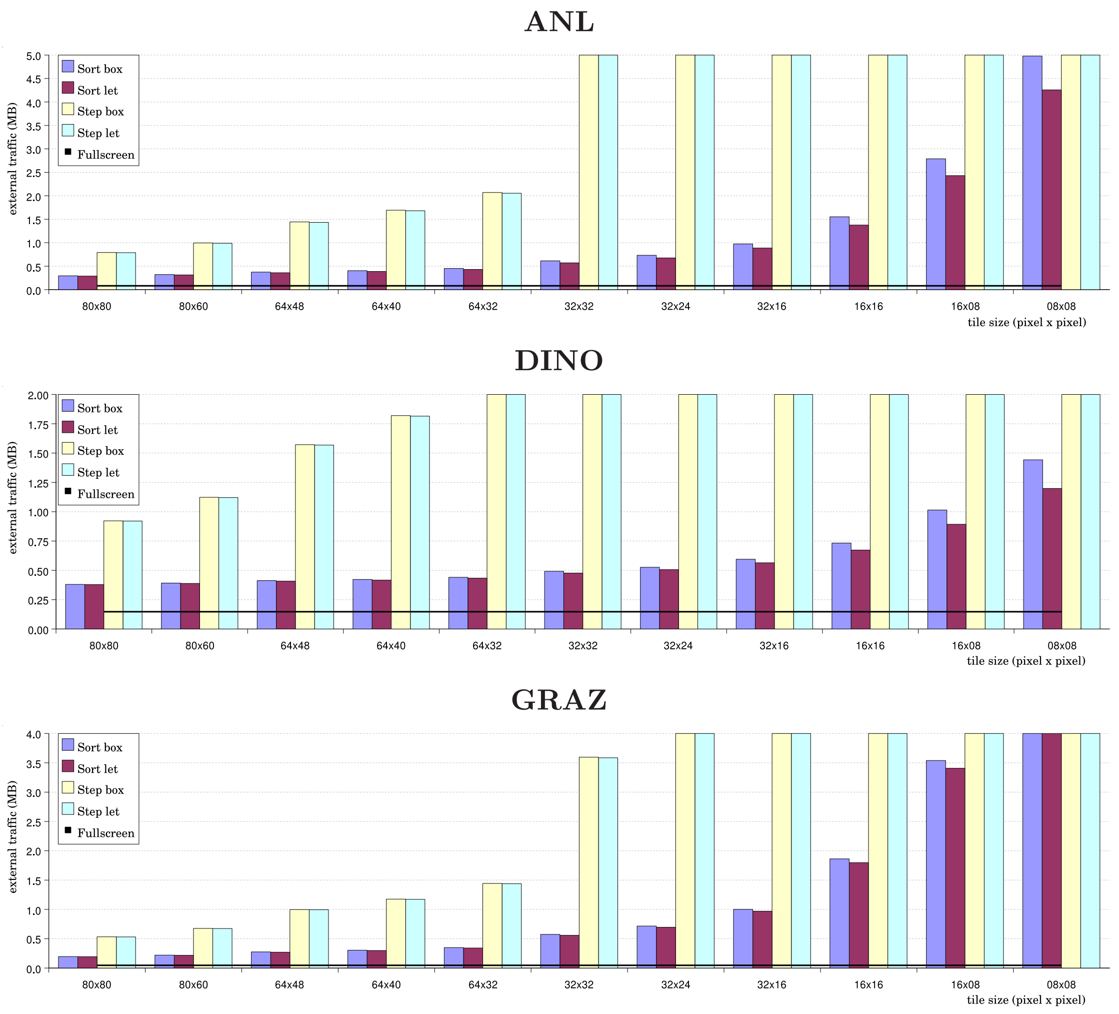


Figure 5.1: Average datafront per frame during the rasterization of the ANL, DINO, and GRAZ benchmarks utilizing four different sorting algorithms in a scenebuffer tile-based rasterization scenario with varying tile sizes.

The external traffic in the datafront measured during the simulation of different scenebuffer rasterization implementations is presented in Figures 5.1 and 5.2. The figures indicate external traffic in the datafront only. The first figure presents the datafront traffic for the three VRML benchmarks ANL, DINO, and GRAZ. The second figure presents the same information for the AWADvs, TUX, and Q3H benchmarks. The results depict the average amount of traffic per frame in megabytes during the processing of 10 frames. Please note that some values in these figures have been clipped (the values for smaller tile sizes) in order to improve the visibility of realistic external traffic values.

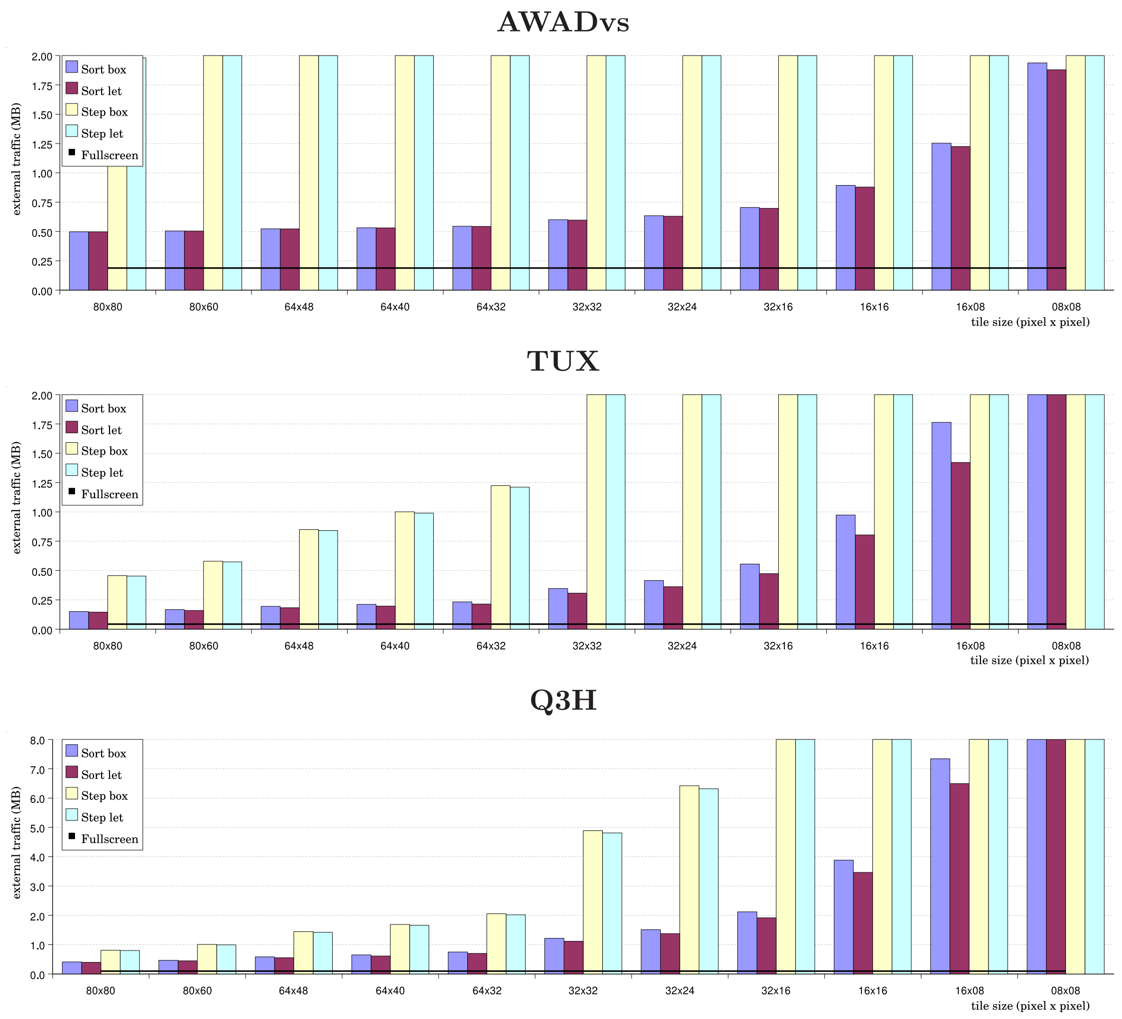


Figure 5.2: Average data_front per frame during the rasterization of the AWADvs, TUX, and Q3H benchmarks utilizing four different sorting algorithms in a scenebuffer tile-based rasterization scenario with varying tile sizes.

From the results in these figures we see that both **step** algorithms result in a large amount of external traffic in the datafront. In these algorithms the number of writes to the external memory required to create the scenebuffer is proportional with the number of writes to the external memory in the datafront of a fullscreen approach. Every instruction is only written to the scenebuffer in external memory once. In the step algorithms the number of writes to external memory are slightly increased by also storing the triangles bounding boxes to the scenebuffer.

However, in the **step** algorithms, the number of reads from the external memory in the datafront is much higher when compared to fullscreen rasterization. Because the scenebuffer is read for every tile, the amount of reads in the datafront is proportional to the number of tiles. Eventhough the algorithm tries to minimize the reads from external memory by first testing a triangles bounding box before fetching the entire triangle data. Our results show that fetching the bounding box data for every triangle for every tile still amounts to a large amount of external traffic.

The results show that the **step** algorithms are only suited for scenebuffer rasterization scenarios with a low number of tiles. Increasing the number of tiles not only increases the duplication of the triangles, but also increases the amount of bits required to store a triangles bounding box. We remark that the **step** algorithms are only competitive in the simulations with tile sizes that are too large to be implemented on-chip.

We observe that the **sort** algorithms are clearly better suited to reduce the amount of external traffic in the datafront during the sorting phase in a scenebuffer rasterization scenario. Like concluded in [7] we see that implementations with small tiles (for example 8×8 pixels) result in high datafront traffic and this traffic can easily be reduced by increasing the tilesize. However, increasing the tile size beyond 32×32 pixels does not result in considerable extra reduction of the datafront traffic. Beyond this tile size a large increase of on-chip memory does not result in a proportional decrease of datafront traffic. We also remark that a 32×32 on-chip memory is the about the largest tile size that can be implemented with a SoC hardware budget for mobile graphics of 500k gates.

We conclude that the external traffic in the datafront of a scenebuffer rasterization scenario is best minimized when using one of the sorting algorithms introduced in previous work: **sortbox** or **sortlet**. Both the **step** algorithms perform poorly for tile sizes that can be realistically implemented on chip. Whether the overlap should be tested via a bounding box test or via a linear edge test is inconclusive. Although the linear edge test results in less traffic in the datafront, the difference between the options is small and only relevant for smaller tiles. The choice for the implementation of the test should be based on the amount of effort required in the microprocessor for a linear edge test versus the amount of effort needed in the tilebased rasterizer to discard ghost triangles from tiles.

Based on these results we can conclude that the most appropriate tile size for a scenebuffer rasterization implementations is either 32×16 pixel or 32×32 pixels, which reconfirms the result from previous work [8]. This is based on the available hardware budget, limiting the tile size to 32×32 and based on the fact that the reduction of the external traffic in the datafront is considerable when compared to implementations with smaller tile sizes.

5.2 Evaluating Databack Traffic

The previous section presented the external traffic figures for the datafront for different scenebuffer rasterization scenarios. This datafront traffic, the external traffic related to sending instructions to the graphics accelerator, is negligible for a direct rasterization scenario because it does not construct a scenebuffer in external memory, but sends instructions to the graphics accelerator hardware on chip directly. Therefore the critical component to evaluate in a direct rasterization scenario is the databack traffic. This section presents simulation results with respect to this databack traffic for several implementations of a direct rasterization scenario. The key aspects that influence the amount of databack are the tile size, the triangle window, and the tile selection policy.

We presented several tile selection policies in Section 3.2.3. These policies determine in which order the tiles of the screen are processed using only the triangles currently available inside the triangle window. The size of the triangle window is dependent on the number of triangles that can be stored on chip. The order in which tiles are chosen directly influences the consumption of triangles from the triangle window and therefore affects the introduction of new triangles. In this way the tile selection policy influences the total amount of tile switches needed in order to render the entire frame. During every tile switch data are exchanged with the external memory, which adds to the databack traffic.

As mentioned in Section 4.3.2 the databack traffic in a scenebuffer rasterization scenario is fixed. Every pixel is only send to the external framebuffer once. For a 640×480 screen this results in 1.23 megabytes of external traffic per frame. For a fullscreen rasterization scenario the databack traffic is the dominant part of the external traffic since every generated fragment needs to be stored in the external framebuffer and therefore generates databack traffic.

This section presents traffic simulation results for the six benchmarks discussed in Section 4.1. The computation of the databack traffic is done with the simulator we developed and presented in Section 4.3.2. To quickly generate avarage traffic figures for each benchmark and several system topologies we process 10 frames for each benchmark, using an offset large enough to skip introduction frames and skipping 25 frames after processing a frame.

The external traffic in the databack measured during the simulation of different direct rasterization implementations is presented in Figures 5.3 and 5.4. The first figure presents the databack traffic in megabytes calculated during the rasterization of the three VRML benchmarks ANL, DINO, and GRAZ in a direct rasterization scenario. The second figure presents the same information for simulations running the AWADvs, TUX, and Q3H benchmarks.

During all these simulations we assumed a fixed triangle window of 32 triangles in order to evaluate the influence of the tile selection policy alone. All triangle tilemasks used in the direct-sorting unit are constructed via a simple bounding box test.

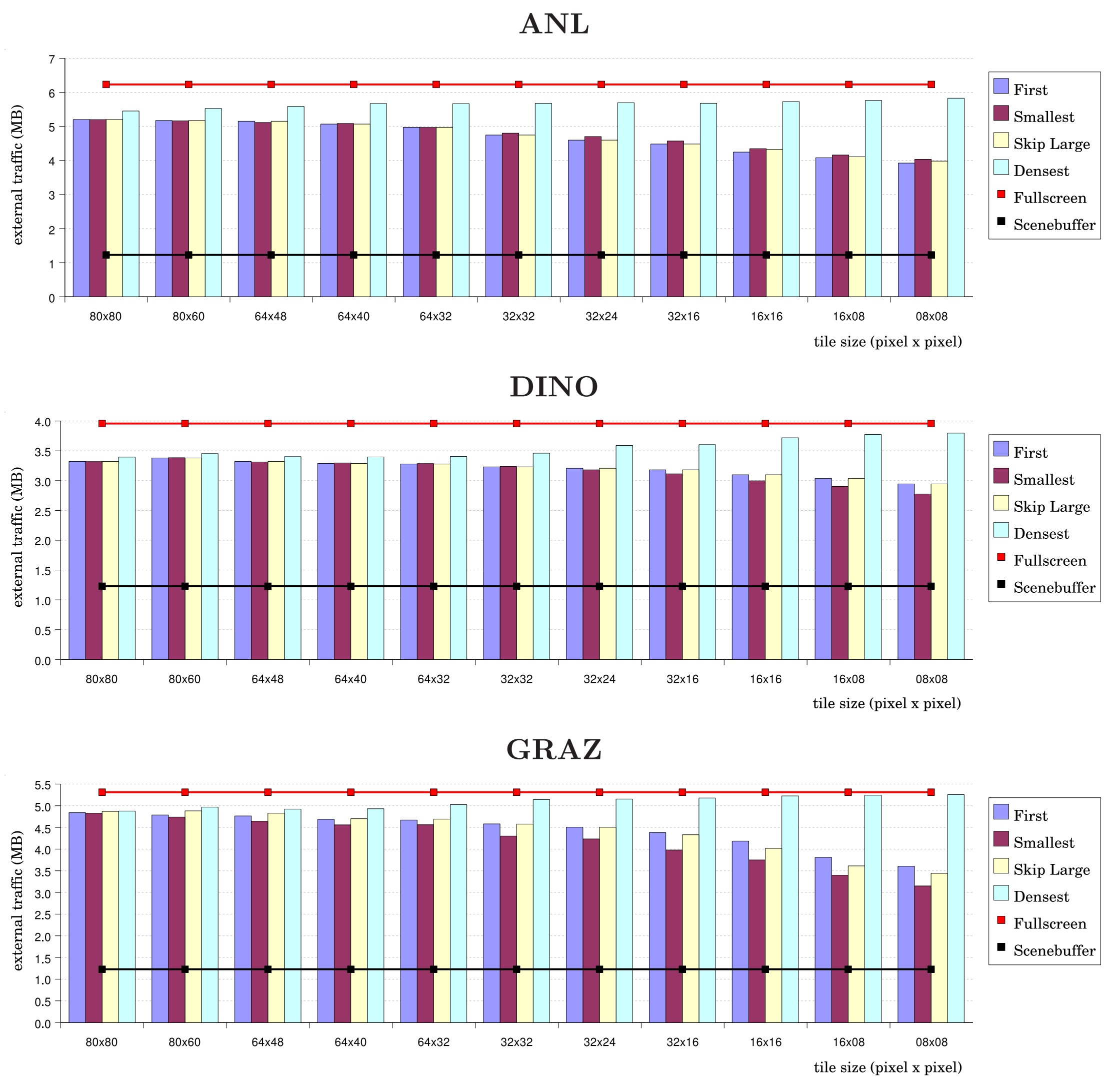


Figure 5.3: Average external traffic in the databack per frame during the rasterization of the ANL, DINO, and GRAZ benchmarks utilizing four different tile selection mechanisms in a direct tile-based rasterization scenario with varying tile sizes.

These figures present the avarage amount of external traffic in the databack per frame during the rasterization of the selected 10 frames while varying the tile selection policy and the tile sizes. The horizontal red line indicates the external traffic in the databack of a fullscreen rasterization scenario, while the horizontal black line indicates the external traffic in the databack of a scenebuffer rasterization scenario. The databack traffic in a direct rasterization scenario must be less then the databack traffic in the fullscreen rasterization scenario in order to be considered as a viable alternative for implementing computer graphics hardware on mobile devices.

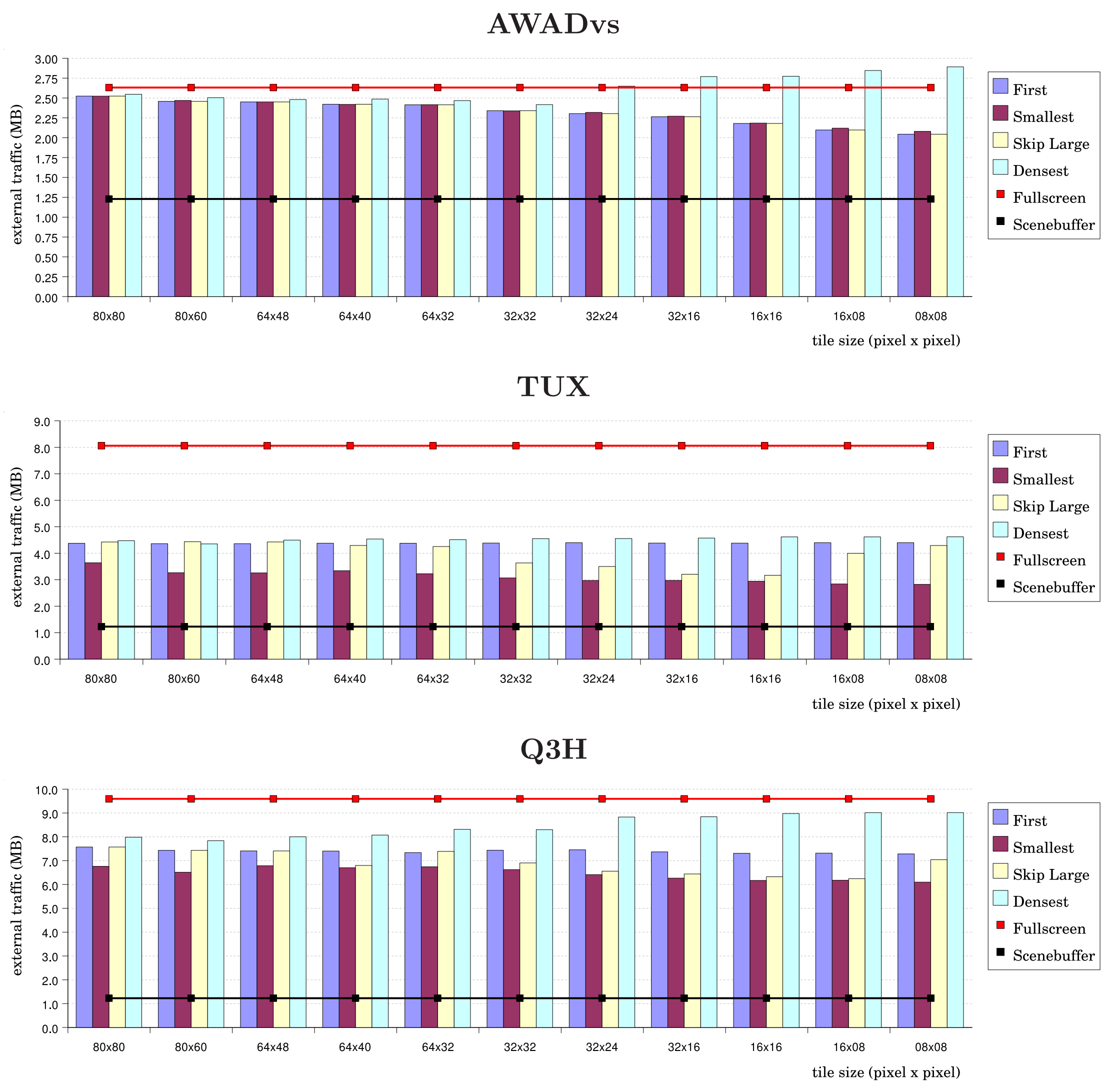


Figure 5.4: Average external traffic in the databack during the rasterization of the AWADvs, TUX, and Q3H benchmarks utilizing four different tile selection mechanisms in a direct tile-based rasterization scenario with varying tile sizes.

From the amount of traffic depicted in these figures we can observe three facts. The first fact is that the tile selection policy that choses the densest tile to process is a poor implementation of the tile selection policy. It is clearly outperformed by the other policies and considering that it is the most complex of the selection policies it is discarded from consideration completely. For the AWADvs benchmark it even generates more databack then the fullscreen approach.

This can be explained by observing the following scenario. During the generation of triangles in the geometry stage most triangles that are defined within one complex shape will be visible in the same region on the screen and therefore also visible in the same tiles. During the generation of triangles of two complex shapes, the geometry stage will first generate all the triangles that construct shape A followed by all the triangles that construct shape B. Consider the case where shapes A and B are placed in the screen as in Figure 5.5.

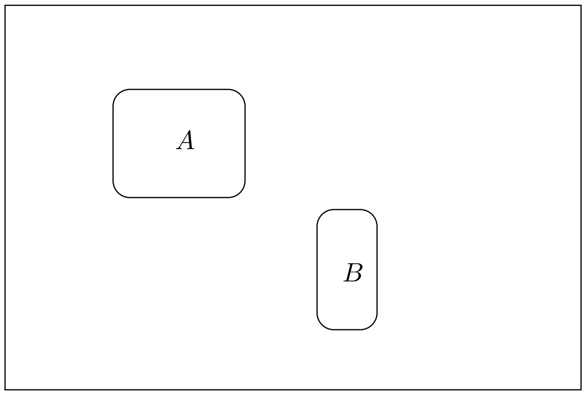


Figure 5.5: Positioning of two non-overlapping Shapes A and B in a screen.

During the rasterization stage the direct rasterization scenario using the densest tile policy first rasterizes the triangles that related to shape A because the triangle window features only triangles related to shape A. The triangle window in this case is illustrated in the top half of Figure 5.6. During this phase triangles from shape A are sent to the rasterizer and finished triangles are discarded from the triangle window in order to introduce new triangles from the triangle stream. The new triangles are related to shape B that resides in another region of the screen.

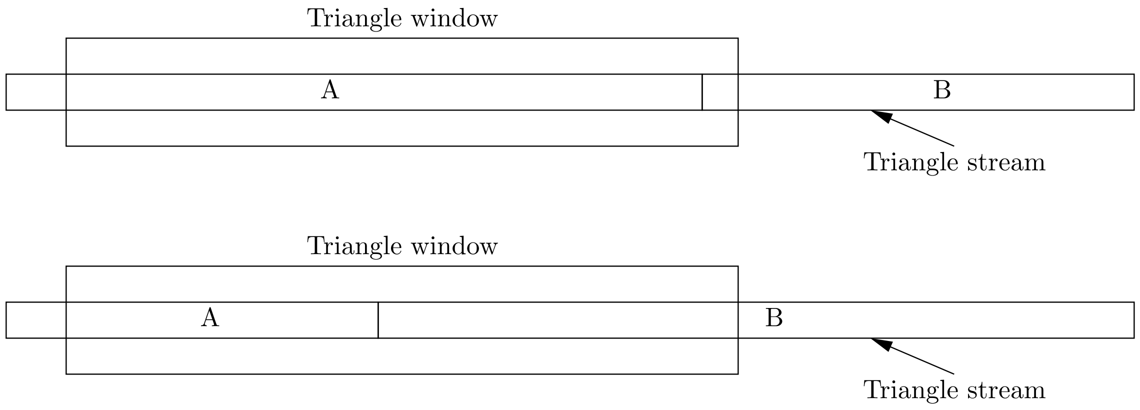


Figure 5.6: Advance of the triangle window in a direct rasterization scenario.

As more and more triangles from shape A are rasterized more and more triangles from shape B are introduced and eventually, before all triangles of shape A are rasterized, the triangle window contains more triangles from shape B than triangles from shape A. The triangle window for this case is illustrated in the bottom half of Figure 5.6. The densest tile policy now selects tiles containing triangles from shape B for rasterization leaving the remaining triangles of shape A inside the triangle buffer. This effect is cummulative when a third and fourth shape are introduced. The rasterization of the new shapes is performed ineffectively because the triangle window is cluthered with old triangles. This problem does not exist in the other three tile selection policies.

The second fact that we observe, when we omit the densest tile selection policy, is that the three remaining selection policies are all able to reduce the databack traffic when compared to the fullscreen rasterization approach for all benchmarks. This reduction of the external traffic is the largest for the TUX benchmark. On overall the selection policy that uses the smallest triangle in the triangle window to select a tile to process results in the largest databack traffic reduction and this is especially visible in the TUX and Q3H benchmarks. Unfortunately, none of the selection policies results in a databack traffic reduction quite as significant as the scenebuffer rasterization scenario. But we have to keep in mind that direct rasterization requires no prior sorting computations in the software running on the microprocessor. Additionally the scenebuffer rasterization scenario produces larger external traffic figures for the datafront traffic.

The third, and probably most interesting fact we observe is that the three remaining tile selection policies result in less databack traffic when working with smaller tile sizes. The maximum traffic reduction for all benchmarks is acquired when rasterizing with the smallest possible rasterizer. (i.e., tile size)

Tables 5.2 through 5.7 list the databack reductions in percentages when compared to the traffic in a fullscreen rasterization scenario for different tile sizes for all the benchmarks. We only listed tile sizes of tiles that could be created from a certain unit size pixel window. The chosen unit size is an 8×8 pixels rasterizer because the scan conversion in the GRAphics AcceLerator is done on an 8×8 pixel window. Also we did not list tile sizes over 32×32 pixels because implementing these systems is likely to exceed the hardware budget for mobile graphics just for the internal memory required to store temporary pixel values.

rasterizer size	`first_triangle`	`smallest_triangle`	`skip_large`	scenebuffer
32x32	23.8	22.9	23.8	74.8
32x24	26.2	24.5	26.2	74.8
32x16	28.0	26.6	28.1	74.8
16x16	31.9	30.3	30.6	74.8
16x08	34.5	33.2	34.0	74.8
08x08	37.0	35.3	36.1	74.8

Table 5.2: Reduction of databack traffic (in %) when compared to a fullscreen approach during rasterization of the ANL benchmark utilizing the proposed tile selection policies.

We observe that a decreasing the tile size from 32×32 pixels to 8×8 pixels results in an increased traffic reduction when compared to a fullscreen rasterization scenario for most of the benchmarks. Only the simulation of the TUX and Q3H benchmarks using the

rasterizer size	`first_triangle`	`smallest_triangle`	`skip_large`	scenebuffer
32x32	18.4	18.2	18.4	81.5
32x24	19.0	19.6	19.0	81.5
32x16	19.6	21.3	19.6	81.5
16x16	21.7	24.3	21.7	81.5
16x08	23.3	26.7	23.3	81.5
08x08	25.6	29.9	25.6	81.5

Table 5.3: Reduction of databack traffic (in %) when compared to a fullscreen approach during rasterization of the DINO benchmark utilizing the proposed tile selection policies.

`skip_large` tile selection policy deviates from this observation. The increased reduction of databack traffic when using smaller tiles is most evident for the ANL, DINO, GRAZ, and AWADvs benchmarks. The increased reduction of databack traffic for decreasing tile sizes is in direct contrast to the scenebuffer rasterization approach that benefits from reduced datafront traffic when increasing the tile size.

This can be explained by observing that tiles are cleared in the first processing visit. A clear instruction is unconditional and sets all pixels in the framebuffer to a default color value and a default depth value. Therefore the first processing visit does not require reads from the external memory. When a large tile size is used this first clear instruction influences all pixels in that tile, and the default values are stored to the external memory after the first visit of that tile. During the second visit of that tile these default values stored in the first visit need to be retrieved from the external memory, which results in databack traffic. In contrast, when a small tile size is used the initial clear instruction is performed for a smaller area, and the unconditional write to the rest of the tiles is preserved. Preserving these unconditional writes seems essential to reduce the databack traffic in a direct rasterization scenario.

Unfortunately, we note that most of the implementations used in the above simulations exceed the hardware gate budget for mobile graphics mentioned in Section 2.1. An indication for the required gates is calculated via Equation (3.3). This indication of the required hardware is based on the total amount of gates required for storage. This includes the storage of triangles parameters and the tilemasks for 32 triangles in the direct-sorting unit and the on-chip storage of temporary fragment values (color, depth and stencilvalues) in the hardware accelerator.

We noticed that reducing the tile size reduces the databack traffic in a direct rasterization scenario. Unfortunately, reducing the tile size also increases the number of tiles on the screen. As mentioned in Section 3.2.2 the direct-sorting unit uses a tilemask to represent the triangle to tile overlap inside the hardware. When a screen is divided into more tiles the size of the tilemask in hardware is increased, which in turn increases the storage requirement per triangle. Table 5.8 lists the hardware requirements for the direct rasterization scenario for the used tile sizes assuming 32 triangles are stored on-chip. The gate requirement for the rasterization unit is calculated via Equation (3.1) and the

rasterizer size	`first_triangle`	`smallest_triangle`	`skip_large`	scenebuffer
32x32	13.8	19.1	13.9	71.7
32x24	15.2	20.3	15.2	71.7
32x16	17.5	25.1	18.5	71.7
16x16	21.2	29.4	24.4	71.7
16x08	28.3	36.1	32.0	71.7
08x08	32.2	40.7	35.2	71.7

Table 5.4: Reduction of databack traffic (in %) when compared to a fullscreen approach during direct rasterization of the GRAZ benchmark utilizing the proposed tile selection policies.

rasterizer size	`first_triangle`	`smallest_triangle`	`skip_large`	scenebuffer
32x32	11.1	11.2	11.1	47.9
32x24	12.5	11.9	12.5	47.9
32x16	14.0	13.7	14.0	47.9
16x16	17.2	17.0	17.2	47.9
16x08	20.3	19.4	20.3	47.9
08x08	22.3	21.0	22.3	47.9

Table 5.5: Reduction of databack traffic (in %) when compared to a fullscreen approach during direct rasterization of the AWADvs benchmark utilizing the proposed tile selection policies.

rasterizer size	`first_triangle`	`smallest_triangle`	`skip_large`	scenebuffer
32x32	45.6	61.9	54.9	81.5
32x24	45.4	63.1	56.6	81.5
32x16	45.6	63.1	60.2	81.5
16x16	45.6	63.5	60.7	81.5
16x08	45.4	64.8	50.4	81.5
08x08	45.4	64.9	46.7	81.5

Table 5.6: Reduction of databack traffic (in %) when compared to a fullscreen approach during direct rasterization of the TUX benchmark utilizing the proposed tile selection policies.

rasterizer size	`first_triangle`	`smallest_triangle`	`skip_large`	scenebuffer
32x32	22.5	30.9	28.0	77.0
32x24	22.3	33.1	31.6	77.0
32x16	23.2	34.7	32.9	77.0
16x16	23.8	35.7	34.1	77.0
16x08	23.7	35.6	34.9	77.0
08x08	24.1	36.4	26.6	77.0

Table 5.7: Reduction of databack traffic (in %) when compared to a fullscreen approach during direct rasterization of the Q3H benchmark utilizing the proposed tile selection policies.

rasterizer size	tiles	rasterization unit memory (gates)	direct-sorting unit memory (gates)	total (gates)
8 × 8	4800	27k	1005k	1032k
16 × 8	2400	54k	546k	600k
16 × 16	1200	108k	314k	422k
32 × 16	600	215k	199k	414k
32 × 24	400	322k	161k	483k
32 × 32	300	430k	142k	572k

Table 5.8: Calculated hardware budget for memory elements in a direct rasterization scenario based on a triangle window of 32 with varying tile sizes.

direct-sorting unit gate requirement is calculated via Equation (3.2). The total size of all triangle parameters is assumed to be 55 bytes total.

The hardware requirements in Table 5.8 suggest that a direct rasterization scenario with a small tile size requires a lot of memory to store the tilemasks inside the direct-sorting unit. This means that a balance needs to be found between hardware budget and databack traffic. The table further indicates that a direct rasterization scenario with a 16 × 16 or a 32 × 16 tile size can be achieved with a reasonable hardware budget. Reducing the triangle window reduces the total amount of storage required, but also reduces the effectiveness of the direct-sorting algorithm.

5.3 Comparing Scenebuffer versus Direct Rasterization

We have already seen some external data traffic figures in the previous two section that we can use in order to compare a scenebuffer rasterization and a direct rasterization scenario. During the investigation of datafront traffic in Section 5.1 we demonstrated the increase of datafront for decreasing tile sizes in a scenebuffer rasterization scenario. In contrast, we proved the decrease of databack traffic for decreasing tile sizes in the direct rasterization scenario during the evaluation of the databack traffic in a direct rasterization scenario in Section 5.2.

In this section we combine the results acquired from datafront and databack simulations in order to determine which implementations of a tile-based rasterization scenario is the best as seen from the memory traffic point of view. For the scenebuffer rasterization scenario we assume the scenebuffer rasterization system uses the **sortlet** sorting algorithm, which results in the minimum amount of datafront traffic. We compare the external traffic figures obtained while simulating the datafront traffic with the **sortlet** algorithm with the external traffic figures obtained by simulating a direct rasterization scenario implementing the `smallest_triangle` selection policy.

The graphs in Figures 5.7, 5.8, and 5.9 present the total external traffic generated by fullscreen, scenebuffer, and direct rasterization for the ANL, DINO, and GRAZ benchmarks, respectively. We observe that the direct rasterization scenario results in more total external traffic when compared to the scenebuffer rasterization scenario for tile sizes

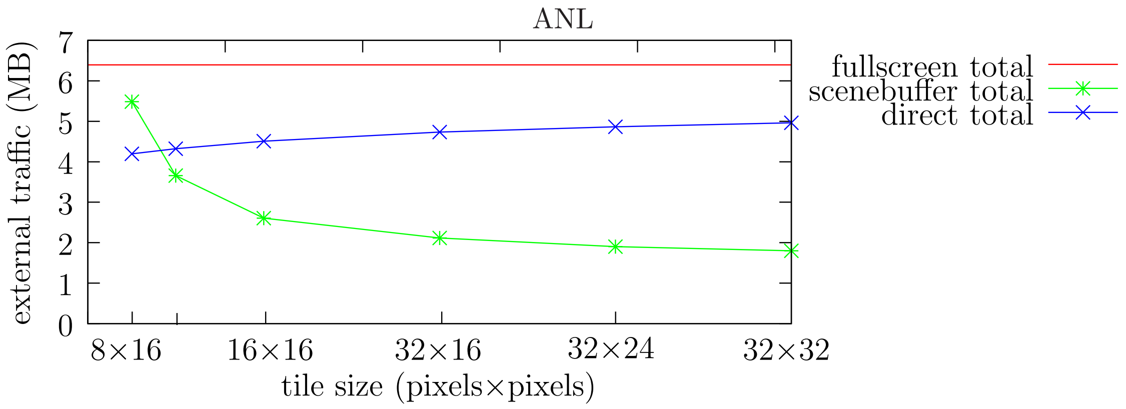


Figure 5.7: Comparing the external traffic between a fullscreen, scenebuffer, and direct rasterization scenario during the ANL benchmark.

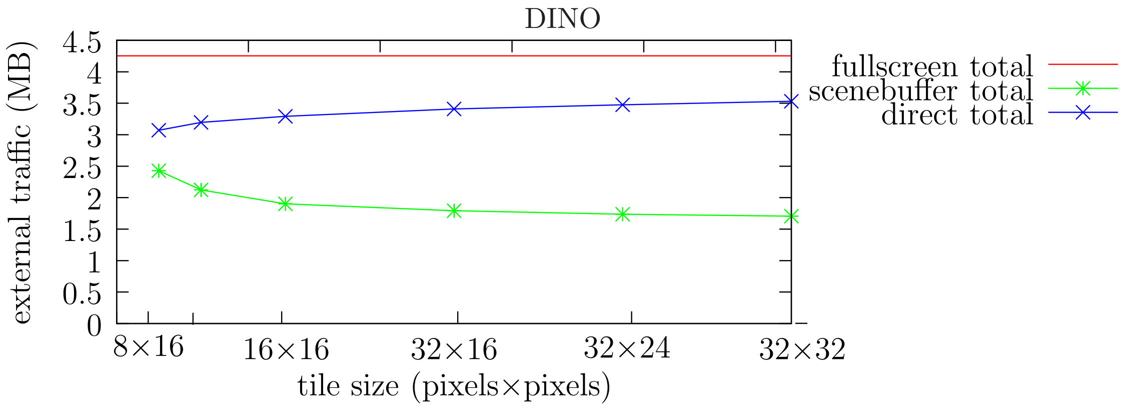


Figure 5.8: Comparing the external traffic between a fullscreen, scenebuffer, and direct rasterization scenario during the DINO benchmark.

larger then 16 × 16 pixels. For tile sizes of 16 ×16 pixels the direct rasterization scenario is still outperformed by the scenebuffer rasterization scenario, from the external traffic point of view, but the difference between the traffic figures is smaller. For even smaller tile sizes the direct rasterization scenario outperforms the scenebuffer rasterization scenario, from a traffic point of view, for most of the benchmarks. In all benchmatks but the Dino benchmark, direct rasterization is able to realize a traffic reduction similar or better than scenebuffer rasterization scenario for small tile sizes of, for example, 8×8 pixels.

Similar results are obtained for the benchmarks AWADvs, TUX, and DINO, which are depicted in Figures 5.10, 5.11, and 5.12, respectively. The highest reduction of external traffic when compared to fullscreen rasterization is realized for the TUX benchmark where on overall the external traffic is reduced by 60 % when compared to the external traffic in a fullscreen rasterization scenario.

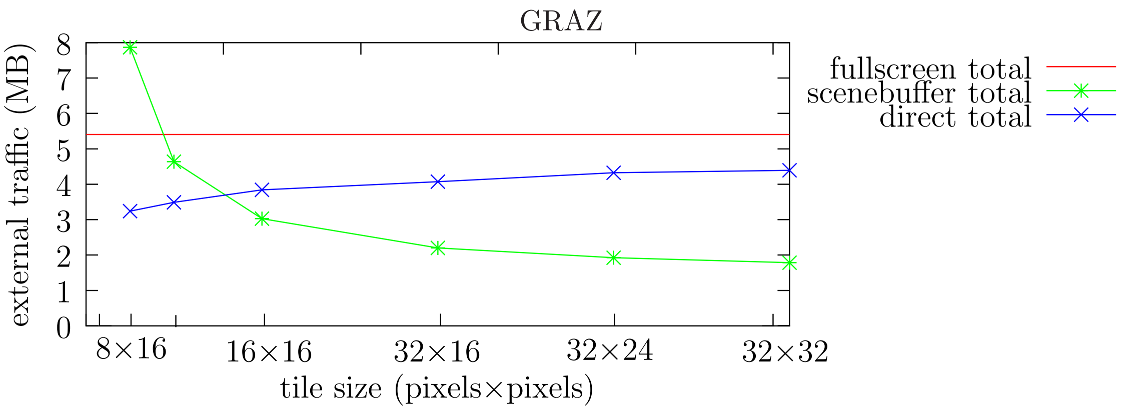


Figure 5.9: Comparing the external traffic between a fullscreen, scenebuffer, and direct rasterization scenario during the GRAZ benchmark.

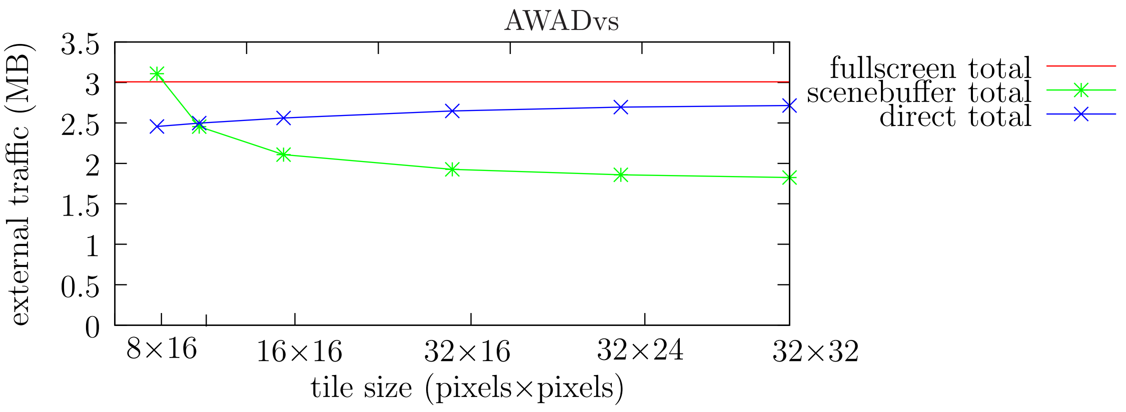


Figure 5.10: Comparing the external traffic between a fullscreen, scenebuffer, and direct rasterization scenario during the AWADvs benchmark for smaller tile sizes.

The fullscreen rasterization scenario performs well for the AWADvs and DINO benchmarks, because these benchmarks do not feature a high amount of overdraw. Therefore the amount of traffic in these benchmasks is already quite low for the fullscreen rasterization scenario and the traffic reduction of the direct rasterization scenario is therefore low, but even for these benchmarks the total amount of external traffic is still reduced when compared to fullscreen rasterization.

Unfortunately, as mentioned in the previous section, the hardware requirements for the direct rasterization scenario increases fast when using decreasing tile sizes. A smaller tile size results in more tiles and therefore a larger tilemask. Since we need to store a tilemask for every triangle on-chip the hardware budget in these system topologies is rapidly exceeded. Table 5.8 presented the hardware requirements for the rasterization scenarios as used for the simulations when we assume that the tile overlap is represented with a tilemask.

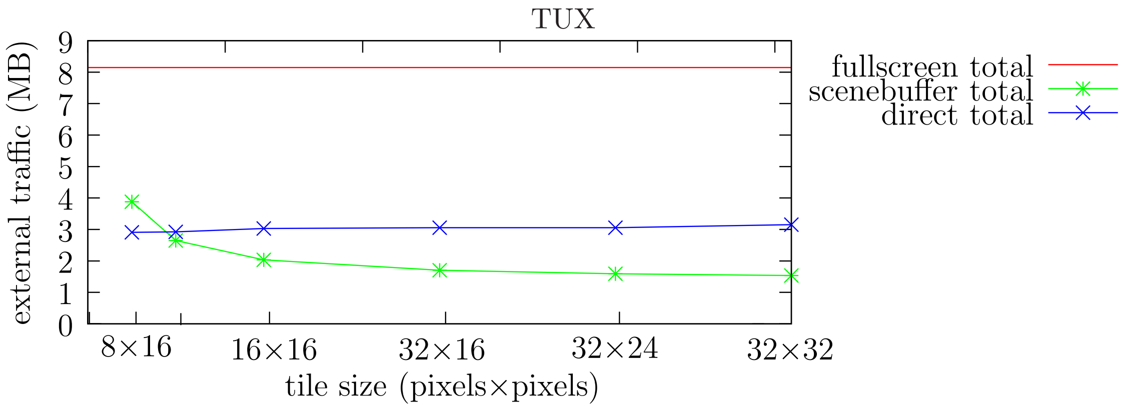


Figure 5.11: Comparing the external traffic between a fullscreen, scenebuffer, and direct rasterization scenario during the TUX benchmark for smaller tile sizes.

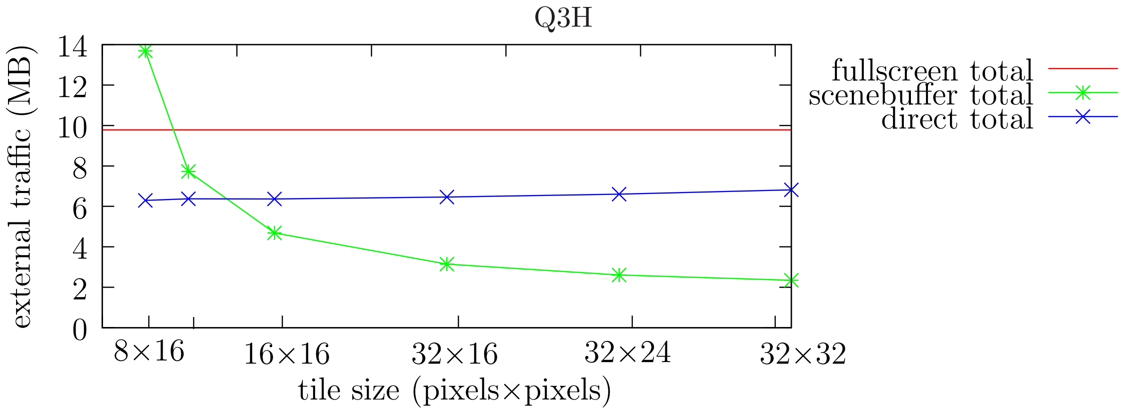


Figure 5.12: Comparing the external traffic between a fullscreen, scenebuffer, and direct rasterization scenario during the Q3H benchmark for smaller tile sizes.

From the results presented in this section we conclude that a direct tile-based rasterization scenario is not able to compete with the scenebuffer tile-based rasterization scenario with the same hardware budget. We therefore conclude that a standalone direct-sorting unit is not feasible for a system implementation with the assumed small hardware budget under 500k gates. The required storage on-chip to store the triangle data (parameters and tilemask) is simply too large to provide the direct-sorting algorithm with a reasonable triangle window. Not to mention that the hardware required for the creation of the tilemask, tile selection, and triangle selection are not yet included in the hardware requirements calculated via Equation (3.3).

Although the performance related to the external traffic of a direct rasterization graphics system increases with decreasing tile sizes the resulting increase in the size of the tilemasks increases the hardware requirements and rapidly exceeds the hardware budget for mobile graphics. However, since the direct rasterization scenario proved usefull for smaller tiles it seems ideal to reduce the hardware requirements in a hierarchical scenebuffer and direct rasterization scenario as proposed in Section 3.4. The following section shows results of data traffic simulations of a hierarchical rasterization with both scenebuffer and direct sorting.

5.4 Combined Rasterization

In Section 3.4 we proposed a hierarchical solution that uses both scenebuffer and direct sorting. By utilizing a hierarchical approach we can create a better balance between workload in hardware and the workload in software. The same simulators as used in Sections 5.1 and 5.2 are used to determine the external traffic for a hierarchical rasterization scenario. In a hierarchical rasterization scenario we utilize scenebuffer sorting to divide the entire screen into several sections and each section is then processed by the direct-sorting algorithm, which further divides each section into tiles.

In the simulations of the hierarchical rasterization scenario we use the **sortbox** algorithm for the implementation of the scenebuffer sorting in software. To determine the datafront traffic for this hierarchical rasterization scenario we simply need to replace $\#_{tiles}$ with $\#_{sections}$ in the computation of the datafront traffic. To determine the databack traffic we first divide the screen into sections and then rasterize all sections sequentially with a simulation of the direct sorting algorithm. Our rasterization simulator that computes the databack traffic, described in Section 4.3.2 is able to divide the screen in an arbitrary number of sections in sections.

In this section we present a novel solution for system designers to balance the workload between hardware and software in a hierarchical scenebuffer and direct sorting rasterization scenario. The effectiveness of the hierarchical rasterization approach is dependent on the system partitioning. We aim to find a suitable partitioning scheme that divides the screen in sections with sorting in software and that uses tile-based rasterizer hardware to process each section individually. An example partitioning of the screen is given in Figure 5.13.

In Section 5.2 we concluded that the direct-sorting unit produces less databack traffic in configurations with smaller tile sizes. The best results, from an external traffic point of view, were obtained during simulations where an 8×8 pixel rasterizer was used. Therefore, in order to reduce the search space while searching for a suitable system partitioning, we assume that the tile based rasterization hardware performs rasterization computations on 8×8 pixel tiles. This has an additional benefit that the amount of stencil, depth, and color values stored on chip is minimal keeping the associated memory array small. This leaves a large part of the hardware budget for storing triangles on chip, which results in a large triangle window used in the direct-sorting algorithm.

On the other hand, we concluded in Section 5.1 that the scenebuffer sorting benefits from large tiles, or sections as we call them in hierarchical rasterization, in order to reduce the external traffic in the datafront traffic. In hierarchical rasterization, from the perspective of the scenebuffer sorting algorithm in software, the tiles for the sorting

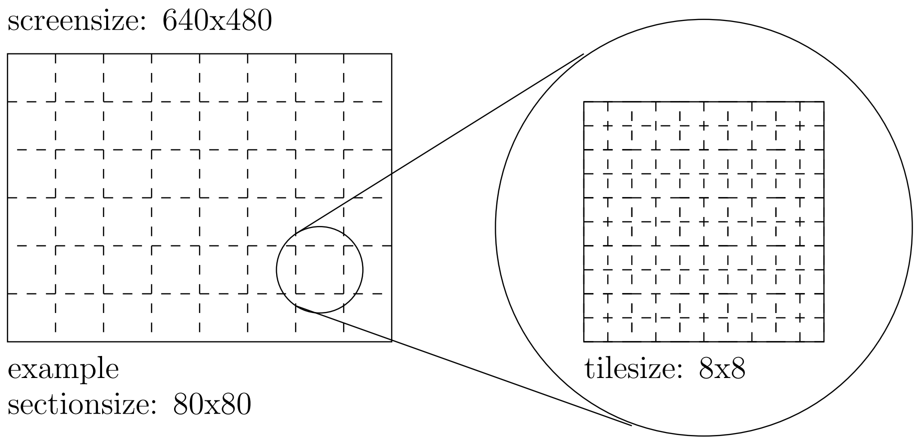


Figure 5.13: Example system partitioning in sections and tiles for a hierarchical rasterization scenario.

algorithm are replaced by sections that can now be larger than the 32 × 32 pixels, the optimal tile size proposed by [4].

To minimize the datafront traffic we can now increase the section size beyond 32 × 32 pixels. However, increasing the section size also results in more tiles in each section, resulting in larger tilemasks. This in turn increases the storage size required per triangle, reducing the total amount of triangles that can be stored on chip. Therefore, the aim in these simulations is to find a suitable section size that reduces the sorting workload on the microprocessor, but still results in a low amount of external traffic to be competative with a scenebuffer rasterization scenario.

The number of triangles that can be stored on chip determines the effectiveness of the direct-sorting unit. It is therefore paramount to increase the number of triangles stored on chip. However, the storage size of a single triangle is already quite large. We assumed the storage sizes for the triangle parameters as listed in Table 4.1. The total storage required for the parameters of a triangle then becomes 54 bytes. We added one byte for additional triangle information like, for example vertex orientation and antialiasing settings, resulting in a total size of 55 bytes (=440 bits) per triangle.

The size of the tilemask, which is also stored for every triangle, is dependent on the size of the sections. A larger section contains more tiles that need to be monitored during the direct-sorting algorithm and thus requires a larger tilemask to be stored on chip. The number of triangles that can be stored on chip can be calculated by rewriting Equation (3.3) to a more suitable form:

$$\#_{triangles} = \frac{\#_{total-gates} - size_{tile} \cdot 70 \cdot 6}{(size_{triangle} + \#_{tiles-per-section}) \cdot 6} \tag{5.1}$$

Where $\#_{total-gates}$ is the total hardware budget available for mobile graphics. As mentioned above we limited the search space by assuming a fixed tile size of 8 × 8 pixels thus the $size_{tile}$ parameter is 64. This parameter affects the amount of storage required for fragment values and reduces the amount of gates that can be utilized to store triangle information. Finally, the size of a triangle is, as again explained above, assumed to be 440 bits. Equation (5.1) then simplyfies to:

$$\#_{triangles} = \frac{\#_{total-gates} - 26880}{(440 + \#_{tiles-per-section}) \cdot 6} \tag{5.2}$$

Table 5.9 lists the maximum triangle window that can be attained for a range of section sizes used during simulation when assuming either a 200k gates budget or a 400k gates hardware budget.

sections	section size (pixel×pixel)	tiles per section	maximum triangles stored on-chip max. 200k gates	maximum triangles stored on-chip max. 400k gates
6	320×160	800	23	50
10	128×240	480	31	67
20	128×120	240	42	91
40	80×96	120	51	111
60	64×80	80	55	119
100	64×48	48	59	127
120	64×40	40	60	129
150	64×32	32	61	131
192	40×40	25	62	133
300	32×32	16	63	136

Table 5.9: Maximum available triangle window for different section sizes assuming a 200k gates budget and a 400k gates budget.

We observe in Table 5.9 that creating a system partitioning with more then 100 sections does not result in a large increase in triangle window for either a 200k gates and a 400k gates hardware budget. This indicates that there is an upper limit to the amount of workload that can be done in software in hierarchical rasterization. Increasing the software workload beyond this limit will not results in a significant reduction of the external traffic, because the effect on the triangle window is limited.

We used the above values as configuration for the data traffic simulations and acquired traffic figures for a hierarchical rasterization scenario for the complex Q3H benchmark. We have chosen this benchmark because it uses color blending and therefore also requires color reads during the rasterization phase, resulting in extra databack traffic. As an addition, we note that these reads during rasterization are bad for performance since they introduce wait stages in the rasterization pipeline.

The scenebuffer sorting in the hierarchical rasterization scenarios is done with bounding box testing. The traffic these scenarios generate are compared with the traffic in pure scenebuffer rasterization where sorting is done with linear edge testing. The results are presented in Table 5.10.

sections	hierarchical (200k)	hierarchical (400k)	scenebuffer (400k)	fullscreen
6	5.684	4.388	2.982	10.016
10	4.752	3.520	2.982	10.016
20	3.757	2.707	2.982	10.016
40	3.128	2.348	2.982	10.016
60	2.910	2.311	2.982	10.016
80	2.785	2.313	2.982	10.016
100	2.776	2.374	2.982	10.016
120	2.795	2.442	2.982	10.016
192	2.957	2.723	2.982	10.016
300	3.304	3.148	2.982	10.016

Table 5.10: Traffic (in MB/frame) for the scenebuffer and for two hierarchical rasterization solutions.

The external traffic figures in a fullscreen and a scenebuffer rasterization scenario are given as a reference. The first column indicates the number of sections used for the scenebuffer sorting algorithm, each section is further processed via the direct tile-based rasterization algorithm with a 8 × 8 pixels rasterizater. In other words, this column represents the amount of software workload that is perfomed on the microprocessor for sorting purposes. For fullscreen rasterization the microprocessor does not perform sorting

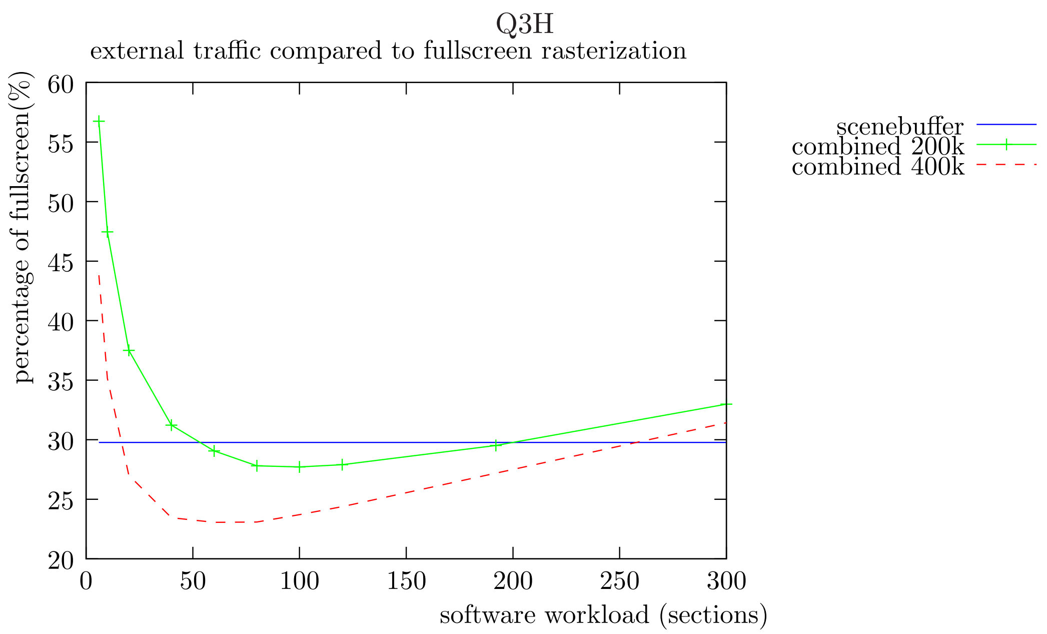


Figure 5.14: Comparing the external traffic in (%) of the fullscreen rasterization traffic for varying amounts of software workload.

and for scenebuffer sorting the microprocessor performs sorting to 300 sections, which corresponds to a tile size of 32 $\times$ 32 pixels, the largest tilesize that can be implemented with 400k gates (see Section 3.1).

Figure 5.14 shows a graph of the external traffic in percentages of the total traffic of a fullscreen rasterization. The traffic reduction of scenebuffer rasterization is constant and given as a reference. The scenebuffer rasterization reduces the external traffic to a mere 29.8 % of the traffic generated in a fullscreen rasterization system. The databack traffic for scenebuffer is off course fixed for every frame, but the datafront is calculated via simulation, where the triangle overlap was determined via linear edge testing.

The results in the table and the graph look very promising and indicate that the best balance from an external traffic point of view between hardware and software can be found between 20 and 150 sections. Within this region the hierarchical rasterization scenario with a 400k gates budget for an embedded rasterizer outperforms the scenebuffer rasterization from an external traffic point of view. Keep in mind that the software workload is also reduced when compared to the scenebuffer rasterization where a 400k hardware budget results in software sorting to 300 sections.

Even when using a lower hardware budget of 200k gates, which means that less triangles can be stored in the direct-sorting unit, the hierarchical rasterization approach results in less external traffic when compared to scenebuffer rasterization. In this case however the amount of sorting required from the microprocessor is higher. The scenebuffer sorting needs to sort the screen to at least 50 sections before the external traffic is reduced below the amount of traffic of a pure scenebuffer rasterization approach. (Please note that this comparison is with a 200k gate budget hierarchical rasterization approach and a 400k scenebuffer rasterization approach.)

The results also suggest that increasing the number of sections in a hierarchical approach above 150, or in other words, allocate more software resources for scenebuffer sorting, results in an increase of the total external traffic. This is caused by the datafront traffic component. The reduction in databack traffic due to smaller sections sizes is canceled out by the increase of traffic in the datafront traffic caused by dublication in the scenebuffer. This is also caused by the fact that an increase in the number of sections does not result in an equal increase in the triangle window as presented in Table 5.8.

The results show that system topologies with a decreasing number of sections to divide the screen, and thus increasing the hardware workload associated to sorting, result in increased external traffic. This is in agreement with the results of the experiments with direct rasterization simulations, where the amount of sections is reduced to an extreme, namely only one section. This is directly caused by the increased size of the tilemask that needs to be stored per triangle. This means that less triangles can be stored on chip and the direct-sorting algorithm is performed on a smaller triangle window reducing the overall effectiveness of the algorithm.

System partitioning schemes that divide the system in less then 10 sections should only be used in systems where the microprocessor is heavily strained. If the microprocessor is able to provide some sorting during rasterization we can reduce the external traffic considerably. If the direct-sorting unit is used to process smaller sections less triangles are processed per section and the triangle window in hardware is used more effective.

The results of these final experiments with a hierarchical rasterization scenario indicate that a limited amount of software sorting, sorting the triangles to, for example, 20 sections, results in a comparable traffic reduction as the previously presented scenebuffer rasterization, where triangles are sorted to 300 sections. In both cases the traffic reduction is measured by comparing it to the external traffic in a fullscreen rasterization scenario.

5.5 Attempt to system level evaluation

To realize the overall merit of our results we also need to examine the reduction of the software workload to some extend. We assumed the application and geometry stages of the graphics pipeline were implemented in software for all rasterization scenarios. One further assumption was that these stages are equal for all rasterization scenarios. The difference is in the sorting required for the rasterization stage for the scenebuffer and for the hierarchical rasterization scenario. No prior sorting in software is required for the fullscreen and the direct rasterization scenario.

In order to get a grasp on the workload reduction we look at the traffic results obtained in the simulation of the q3h benchmark used as the critical benchmark in Section 5.4. The software workload is pre-rasterization only and in order to determine the software workload for the different rasterization scenarios we need to known the number of triangles in a frame and the amount of overlap these triangles have with the number of tiles in the screen. The amount of overlap for a frame, where the screen is divided in a different amount of tiles or sections is given in Table 5.11. There are 43.707 total triangles in a frame.

screen division	number of sections	amount of overlap (bounding box)	amount of overlap (linear edge test)
20 × 15	300	140,669	116,228
16 × 12	192	114,530	99,135
10 × 12	120	94,231	84,894
10 × 10	100	88,143	80,398
8 × 10	80	81,468	75,384
10 × 6	60	77,278	72,479
8 × 5	40	68,256	65,669
5 × 4	20	58,251	57,165
5 × 2	10	52,773	52,619
2 × 3	6	50,634	50,340

Table 5.11: Total amount of triangle to tile overlap per frame for the q3h benchmark.

Based on these figures we can estimate the amount of software workload for the sorting of triangles to tiles. We can distinguish three parts in the software workload.

- First, we have the amount of eqautions required in order to constuct the bounding box for a triangle. These equations involve calculating $x_{min}, y_{min}, x_{max}$, and y_{max} for every triangle. To calculate these values we need 6 comparisons for every triangle.

- Second, the sorting of the triangles to tiles. This requires 2 comparisons per horizontal and 2 comparisons per vertical tile boundary per triangle. It is only one comparison for the left and right side of the screen. In total this comes down to 2 comparisons per horizontal or vertical screen divisioning. The amount of workload this generates is dependent on the total number of tiles in the screen and how the tiles are distributed in the screen.

- Third and finally, the actual save instructions required when storing a triangle to the scenebuffer. The triangle information needs to be stored in the scenebuffer in external memory. In the previous we assumed the exact triangle information was 55 bytes, which results in 55 store instructions per triangle. However, the exact triangle data only needs to be stored for every triangle once. For each usage of the triangle beyond the first one the sorting algorithm needs a the store instruction for a pointer to the memory location of the triangle. Off course for fullscreen and direct rasterization every triangle is only stored once (either to a temporary buffer in external memory or directly to rasterization hardware on chip).

Using the above assumptions we can get a grasp on how software workload is affected by different rasterization scenarios. Please keep in mind that we are comparing the amount of pure calculation instructions only. Any conditional instructions for software flow are omitted. Neither do we regard the difference in pure workload (in terms of energy) for, for example, a store instruction and a simple addition. Evaluating software workload with these effects requires exact software models and complex microprocessor and cache simulation.

Based on the above assumptions we calculate the software workload in calculation instructions via the following equations:

$$ins_{bb} = \#_{triangles} \cdot 6 \tag{5.3}$$

Where ins_{bb} is the total amount of instructions spend on calculating the bounding box for triangles in software.

$$ins_{sort} = \#_{triangles} \cdot (\#_{horizontal} + \#_{vertical}) \cdot 2 \tag{5.4}$$

Where ins_{sort} is the total amount of instructions spend on sorting the triangles to bins in software. For each horizontal tile boundary we need to compute 2 comparisons (Equations (2.5) and (2.7)) and similarly for each vertical tile boundary we need to compute 2 comparisons (Equations (2.6) and (2.8)). These comparisons need to be computed for every triangle.

$$ins_{store} = \#_{triangles} \cdot 55 + \#_{overlap} \tag{5.5}$$

where ins_{store} is the total amount of store instructions needed to save data to the scenebuffer. Every triangle is completely stored to the scenebuffer only once, using 55 store instructions. Every other usage of the triangle due to overlap in several tiles results in the storing of a single memory reference with a single instruction.

Using the above equations and the overlap data given in Tabel 5.11 we can compute a rough estimate of the software workload for the q3h benchmark. The calculated software worload (in instructions) for several different rasterization scenarios is given in Table 5.12.

rasterization scenario	number of sections	ins_{bb}	ins_{sort}	ins_{store}	total instructions	external traffic (MB/frame)
fullscreen	N/A	0	0	612k	612k	10.016
direct	N/A	0	0	612k	612k	*6.294
scenebuffer	300	262k	3,059k	752k	4,074k	2.982
hierarchical	192	262k	2,448k	726k	3,436k	2.723
hierarchical	120	262k	1,923k	706k	2,891k	2.442
hierarchical	100	262k	1,748k	700k	2,711k	2.374
hierarchical	80	262k	1,573k	693k	2,529k	2.313
hierarchical	60	262k	1,399k	689k	2,350k	2.311
hierarchical	40	262k	1,136k	680k	2,079k	2.348
hierarchical	20	262k	787k	670k	1,719k	2.707
hierarchical	10	262k	612k	665k	1,539k	3.520
hierarchical	6	262k	437k	663k	1,362k	4.388

Table 5.12: Software workload in instructions for a set of rasterization scenarios. (*: acquired by running a hierarchical approach with only 1 section, effectively running a direct rasterization scenario.)

These figures suggest that the software workload is significant and can substancially be reduced by reducing the number of tiles in the screen. Keep in mind that the hierarchical rasterization scenario works with sections instead of tiles, but from the software point of view the sorting is performed to larger sections reducing the workload on the microprocessor.

We finally present in Table 5.13 the software workload of the hierarchical rasterization scenarios is given as a percentage of scenebuffer rasterization. The table suggests what is to be expected; when sorting to a lower amount of sections the software workload is reduced. The amount of software effort is unaffected by the amount of hardware used to construct the hierarchical rasterization implementation.

In Table 5.10 we presented the external traffic figures of the hierarchical rasterization scenario, when compared to the external traffic figures of scenebuffer rasterization. In Figure 5.14 thereafter we visualized the traffic reduction (in %) in a hierarchical rasterization scenario when compared with scenebuffer rasterization. Here we again present the reduction of external traffic in the hierarchical rasterization scenarios, but this time

rasterization scenario	number of sections	software workload (%)	external traffic 400k budget (%)	external traffic 200k budget (%)
direct	N/A	15.02	-	-
fullscreen	N/A	15.02	-	-
scenebuffer	300	100.00	100.0	100.0
hierarchical	300	100.00	105.57	110.80
hierarchical	192	84.34	91.31	99.16
hierarchical	120	70.97	81.89	93.73
hierarchical	100	66.53	79.61	93.09
hierarchical	80	62.07	77.57	93.39
hierarchical	60	57.68	77.50	97.59
hierarchical	40	51.02	78.74	104.90
hierarchical	20	42.19	90.78	125.99
hierarchical	10	37.77	118.04	159.36
hierarchical	6	33.43	147.15	190.61

Table 5.13: Software workload reduction by decreasing the number of sections to sort and the external traffic reduction in a hierarchical rasterization scenario.

we also present the associated reduction in software workload in Figure 5.15.

This figure implies that a hierarchical rasterization approach reduces the software

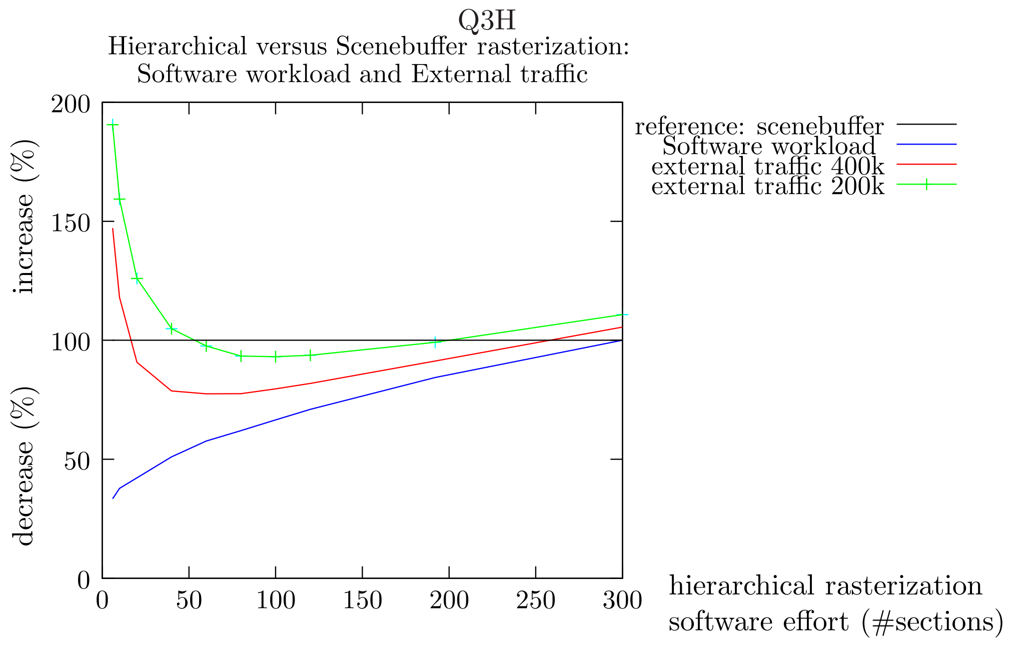


Figure 5.15: The reduction of external traffic (in %) and the reduction in software workload (in %) of hierarchical rasterization when compared with scenebuffer rasterization

workload related to sorting in the microprocessor as well as the external traffic related to accesses to and from the framebuffer generated by the rasterization hardware. It does however show a limit to what hierarchical rasterization can do. If too much of the software sorting is removed the external traffic is increased substancially due to the increase of databack traffic. According to the figure a software sorting algoritm that sorts the screen to 40 up to 100 sections results in the least amount of external traffic (for both a 200k and 400k gate approach) while reducing the sortware workload substancially when compared to the scenebuffer rasterization proposed in previous work.

In this chapter we see that the external traffic can be reduced when compared with the external traffic in a fullscreen rasterization scenario in varying manners and with varying degrees of efficiency. As seen in Table 5.7 the direct rasterization scenario can reduce the external traffic to 66-70 % without increasing the software workload when compared with fullscreen software workload for a large demanding benchmark as the q3h benchmark. This is still more then twice the amount of traffic the scenebuffer rasterization scenario generates and additionally this results is purely theoretical, since we noted that the amount of hardware needed to create the direct sorting unit in this scenario exceeds the hardware budget assumed at the start of this thesis and in previous work.

However, by allocating software resources in a hierarchical rasterization scenario we can actually further reduce the external traffic when compared to scenebuffer rasterization. In this case the microprocessor performs an amount of sorting before rasterization is started. Table 5.13 shows that this can be done without immediately relying solely on software sorting as is done in the scenebuffer rasterization scenario. Relying solely on software for the sorting results in a large increase of software workload for the microprocessor as is eminent from Table 5.12 and 5.13, Sorting the triangles to 300 sections requires about 5 times as much workload on the microprocessor when compared to the fullscreen or the direct rasterization scenario, who in theory do not require sorting at all. Furthermore, the external traffic actually increases with the number of tiles in a screen. Larger screen resolutions (from 640×480 and larger) therefore result in a large datafront traffic due to sorting the triangles to bins.

The hierarchical rasterization scenario is ideal for minimizing the increase of software workload on a tile-based rasterization system when compared to the increase of software workload in scenebuffer rasterization. In addition the external traffic due to sorting the triangles is minimized by keeping the number of bins to sort to low. Combining a small tilebuffer (which results in a low hardware cost) with reduced software workload (performing less sorting on the microprocessor) gives future system engineers an ideal solution where the sorting workload can be balanced more easily between hardware and software.

Conclusions 6

In this thesis we explore system configurations that could improve the power consumption on mobile devices capable of running complex three dimensional graphics applications, such as smart phones, handheld consoles, or personal digital assistants. In a computer graphics system the rasterization stage is the stage most suitable for optimization and is often implemented in hardware. Unfortunately, the common approach to accelerate the needed computations in hardware is not suited for mobile devices since it results in high traffic between the hardware accelerator and the system memory. This traffic is a major source of energy dissipation and should be minimized on mobile devices.

We present a brief overview of the concepts of tile-based rasterization, a rasterization strategy proposed in previous work that divides the display into smaller section which can be processed individually, in Section 6.1. Following, in Section 6.2, we follow with a summary of two new tile-based rasterization scenarios we contribute to the field of mobile graphics, namely direct and hierarchical (tile-based) rasterization. In contrast to the previously proposed scenebuffer (tile-based) rasterization scenario our proposed scenarios relieve computational stress from the systems microprocessor. We end by providing a number of recommendations for future study in this field in Section 6.3.

6.1 Summary

In mobile graphics system engineers are faced with an intriguing dilemma. To enable computer graphics on mobile devices we should use as little power as possible because mobile devices are powered by a limited power supply in the form of batteries. Additionally we can only use a small amount of hardware in order to result in a competitive product. It was estimated by ARM, a leading provider of mobile hardware solutions, that a hardware budget of only 200k-500k gates is available on-chip that can be utilized for graphics processing on mobile devices.

Rasterization is the critical stage of computer graphics that is often accelerated in hardware because in this stage the amount of computations increases dramatically. Traditional hardware rasterization is performed for a single triangle for the entire screen at once. In this thesis we therefore refer to such an approach as fullscreen rasterization. Unfortunately, fullscreen rasterization results in high traffic between the accelerator hardware and the framebuffer in external memory. External memory accesses are a major source of power dissipation, therefore fullscreen rasterization is not an ideal solution for mobile graphics.

To enable computer graphics on mobile devices tile-based rasterization was introduced in the scope of the GRAAL project [16]. In tile-based rasterization the screen is divided in tiles and the rasterization of tile can be done independently because the final color values of a pixel is not related to surrounding pixels and rasterization of tiles can therefore be performed sequentially by reusing a small hardware accelerator. Furthermore, all intermediate fragment values required for rasterization for a single tile are stored on-chip, reducing external traffic to a large framebuffer, and consequently reducing the overall power consumption.

In previous work a tile-based rasterization scenario was introduced that processes all tiles of a screen one by one in a fixed order. This scenario does not require the storage of intermediate fragment values to the framebuffer in external memory because the tiles are processed sequentially and every tile is only processed once. Only the final color values are stored to the external memory. This minimizes the external traffic between the rasterizer and the framebuffer in the external memory. We refer to this traffic as databack traffic.

However, this approach requires all the primitives in a frame to be stored in the external memory, in a memory buffer called the scenebuffer, prior to rasterization. This approach is therefore called scenebuffer rasterization. The triangles stored in the scenebuffer can be used in any number of tiles and must therefore be stored untill all tiles are completed, additionally, they need to be fetched from the external memory for every tile they are present in. retrieving the primitive data from the external memory for every tile obviously results in external traffic, we refer to this traffic as datafront traffic. It was shown in previous work that a tile size of 32×32 pixels results in an optimal choice considering the hardware required and the reduction of datafront traffic.

A tile size of 32×32 pixels requires approximately 393k gates to store the intermediate pixel values during rasterization. Given a 500k gate budget for graphics processing this leaves little room for the implementation of the actual rasterization hardware. This implies that the instruction sorting algorithm in scenebuffer rasterization is not performed on specialized hardware, but in software running on the microprocessor. During this research we assumed a display resolution of 640×480, which results in a total of 300 tiles to create a full screen image. This means that after a triangle is defined by the geometry stage of the graphics pipeline in software, the microprocessor is required to determine the overlap of that triangle with 300 different tiles.

This results in added workload for the microprocessor. Unfortunately, this microprocessor is the core of the mobile device, responsible for a range of other tasks next to computer graphics, running the operating system, providing user audio, handling user input, and since we aim at mobile devices, wireless communication. By implementing the sorting of triangles to 300 tiles we are straining the microprocessor and decreasing overall system performance of the mobile device. Therefore, in this thesis we investigated a new tile-based rasterization scenario that reduces the workload on the microprocessor when compared to scenebuffer tile-based rasterization, while still aiming to reduce the external traffic when compared to fullscreen rasterization in order to minimize power consumption.

6.2 Contributions

At the start of this research we had an indication that the software sorting as used in the scenebuffer rasterization puts to much strain on the embedded microprocessor. We tried to alleviate that affect by removing the software sorting from the system and propose a sortless form of tile-based rasterization. The goal of this sortless rasterization is to remove software workload from the embedded microprocessor, but it also needs to restrict external traffic to the external memory because accesses to the external memory consume a lot of power. Our contributions to the field of mobile graphics are as follows:

- First, we proposed direct (tile-based) rasterization. This rasterization concept is a sortless form of tile-based rasterization that removes the software workload from the embedded microprocessor entirely. We investigated the system modifications required for direct rasterization. Instead of sorting the incoming triangles in software they are immediately send to the rasterization hardware, where they are temporarily storing in a buffer. Because we want this buffer to be on-chip it has to be small and can therefore only hold a limited amount of triangles.

  To create a direct rasterization approach we presented three simple heuristics that can be implemented in hardware that choose a tile to process based on the currently available triangles. The functions of storing the triangles, selecting triangles to process, and selecting tiles to process are to be implemented in a direct-sorting unit in hardware. This hardware unit is to be placed between the microprocessor and the (tile-based) rasterizer. Additionally, to prevent unnecessary external traffic to and from the framebuffer in external memory we proposed certain hardware changes to the rasterizer.

- Second, we proposed hierarchical (tile-based) rasterization. We noted that scenebuffer sorting as proposed in previous work [2] and the direct-sorting unit as proposed in this work are not mutually exclusive in a tile-based rasterization system. We propose to combine both tile-based rasterization techniques into one hierarchical rasterization approach. In this hierarchical approach the software workload on the microprocessor can be reduced by sorting to fewer bins, while the efficiency of the direct-sorting unit is increased by only processing part of the screen at a time instead of processing the entire screen in one go.

In order to examine the feasibility of these two concepts when compared to scenebuffer rasterization we decided to compare the amount of external traffic between the SoC and the external memory generated during rasterization since this traffic is a major source of power consumption in a system. The power consumption is a critical design consideration for mobile devices.

In order to compare the external traffic we developed a simulation framework that computes the amount of external traffic for a given rasterization approach. This computation is split in two parts; (i) external traffic between the microprocessor and the scenebuffer generated during scenebuffer sorting before rasterization, called datafront, and (ii) external traffic between the rasterizer and the framebuffer during rasterization, called databack.

Because there are a large number of system parameters that can still be chosen (tile size, then amount of triangles stored on-chip, etc) we made the simulation framework as configurable as possible. We used the framework to evaluate the external traffic in scenebuffer, direct, and hierarchical rasterization for a number of benchmarks taken from [6]. We obtained results for a wide range of system topologies:

- The results indicate that the direct rasterization scenario is not as efficient in reducing external traffic as scenebuffer rasterization is for a wide range of chosen direct rasterization topologies. Experimental results for direct rasterization suggest a best avarage reduction of external traffic of 20% over the benchmarks when compared to fullscreen rasterization, while results for scenebuffer rasterization suggest an avarage reduction of 75% when compared to fullscreen rasterization.

  The best results were obtained for direct rasterization simulations where the smallest tile size (8×8) was chosen. Unfortunately, we noted that reducing the tile size increases the number of tiles in the screen and increases the size of the tilemask stored alongside a triangle inside the direct-sorting unit. This means less triangles can be stored on-chip and the eficiency of the direct-sorting algorithm goes down. Based on these results we think that a direct (tile-based) rasterization is not a very suitable implementation for future mobile graphics devices.

However, experimental results for hierarchical rasterization are a lot better. Since there is a large design space for the hierarchical rasterization approach, there is no clear way to predict an optimal division of the screen into sections and the sections into tiles. Therefore, to show the possibilities of the hierarchical rasterization approach, we presented two case studies with different hardware budgets (200k and 400k gates[1]). We examine the effect of varying the software workload by varying the number of bins the scenebuffer algorithm sorts to. The search space is limited by selecting the system topology that allows the most triangles to be stored on chip via an exhautive search algorithm assuming a tile-based rasterizer of 8×8 pixels.

We computed the external traffic in the hierarchical rasterization for the system topology acquired from the exhaustive search via simulations. Additionally, we perform a crude but effective computation of the software workload based on the screen divisioning in sections. Afterwards we compared the external traffic and the software workload to the external traffic and the software workload to scenebuffer rasterization with a tile size of 32×32 pixels (which was the optimal tile size concluded from [2]) as a reference.

- Results for the 200k budget hierachical rasterization approach indicate that a screen (total size: 640×480 pixels) divided into 60 sections (each of size: 64×80 pixels) sorting via a scenebuffer algorithm; and each section subdivided in 80 tiles (each of size: 8 ×8 pixels) sorted via a direct-sorting unit generates 2.4% less external traffic when compared to scenebuffer rasterization. In addition, it generates only 57.7% of the software workload. This means that a system implementation of half the comparable hardware budget still manages to reduce external traffic and software

[1]The hardware budget for a scenebuffer rasterization with tile size of 32×32 is also approximately 400k gates.

workload when compared to scenebuffer rasterization. We note that the external traffic can be further reduced with an extra 5.5% by using more sections and thus utilizing more software workload.

- Results for the 400k budget hierachical rasterization approach indicate that a screen (total size: 640 × 480 pixels) divided into 20 sections (each of size: 128×120 pixels)sorting via a scenebuffer algorithm; and each section subdivided in 240 tiles (each of size: 8 ×8 pixels) sorted via a direct-sorting unit generates 90.8% of the external traffic when compared to scenebuffer rasterization, while only generating 42.2% of the software workload. This means that a system implementation with a comparable hardware budget manages to reduce external traffic slightly and reduces the software workload considerably when compared to scenebuffer rasterization.

  Another advantage of hierarchical rasterization is also demonstrated in the 400k gate budget case. The results suggest that a large number of system topologies can outperform the scenebuffer rasterization scenario (eminent from both an external traffic reduction and a software workload reduction). By utilizing more software workload the external traffic can be reduced to 77.5% when compared to scenebuffer rasterization.

Based on these results we conclude that hierarchical rasterization is a well suited technique to balance the workload on a mobile graphics device. It demonstrates that the external traffic can be reduced when compared to scenebuffer sorting. This reduction is achieved by reducing the number of sections the scenebuffer sorting algorithm in software sorts to.

With less sections there is less triangle overlap to sections. Eventhough a large amount of triangle overlap does not result in a large amount of external traffic during the construction of the scenebuffer (the complete triangle data is stored only once, and every overlap is stored as a reference) it does increase the number fetches from the scenebuffer (the complete triangle data must be retrieved for every tile) resulting in a large amount of external traffic. Additionally, these accesses to the extern memory are reads, which could possible stall the rasterization process due to high latencies of read operations on larger memories.

Next to the reduction in external traffic is the reduction of software workload on the embedded microprocessor, which is off course the actual objective of this thesis. The results indicate that it may not be possible to implement a competitive rasterization approach to scenebuffer rasterization that completely removes software sorting as indicated by the results of the direct rasterization simulations, but the results of the hierarchical case studies suggest that is is possible to reduce both the software workload and the external traffic when compared to scenebuffer rasterization.

6.3 Recommendations

During this research a lot of interresting suggestions emerged that deserve examining, however we did not pursue these directions in this research due to timing issues. We present them here for interested readers.

6.3.1 Tile List Overlap Representation

We observed that the effectiveness of direct rasterization was increased for smaller tile sizes. However, we concluded these scenarios where unfeasible because the required hardware would exceed the hardware budget. During our simulations we assumed the hardware representation of triangle to tile overlap was implemented via a tilemask, requiring one bit per tile. Tilemasks are of fixed size and easy to implement in hardware, but can become quite large when we divide the screen to a large number of tiles.

In Section 3.2.2 we also proposed representing the triangle to tile overlap with a tilelist. Due to its dynamic nature, which makes it problematic to implement in hardware, this representation was not considered further. The storage size of a tilelist is dynamic and complex to implement in hardware, but when the screen is divided in a large number of tiles it might ultimately by smaller then a tilemask representation. This could mean that a direct rasterization scenario with a very small tile size, for example $8\times\ 8$ can be implemented with a lower hardware budget when compared to the assumed direct rasterization with a tilemask representation. The results obtained simulating a direct rasterization scenario using a $8\times\ 8$ pixels tile can then obtained with a more realistic hardware budget.

6.3.2 Dynamic Size of Sections

The hierarchical rasterization scenario works by dividing the screen and so sending less triangles to the direct-sorting unit in hardware for each seperate section. The amount of triangles that can be stored on chip is fixed so reducing the total number of triangles needed for a section (by reducing the section size) means that a larger ratio of the total triangles is available for the rasterization process. This reduces the amount of tile switches needed to completely finish a section and ultimately reduces the databack traffic.

In our simulation of a hierarchical scenebuffer and direct rasterization scenario we assumed the scenebuffer-sorting stage divided the screen into sections of equal size. However, with only a few alterations the hardware implementation of a direct-sorting unit designed for a large sections, for example $160\ \times 160$ can also be used to sort the triangles for a smaller section, for example 40×40. By disabling parts of the hardware a smaller tilemask can be constructed, more triangles can be stored on chip due to the smaller tilemask size, resulting in a more efficient pass of the smaller 40×40 tiles compared to the pass of the 160×160 tile.

It might be rewarding to study the effects of non-uniform or even dynamic sectioning. In nonuniform partitions the sectioning scheme could be implemented as in Figure 6.1. This approach assumes the level of detail is largest in the center of the screen. Assuming the concentration of triangles is largest in the center it aims to minimize the number of triangles send for these section to the direct-sorting unit by decreasing the section size in the center. Whether this actually improves performance is dependent on the type of application and the amount of overhead it imposes on the scenebuffer-sorting stage.

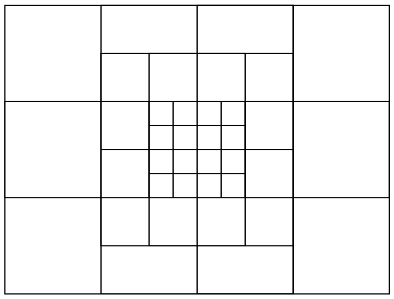


Figure 6.1: Decreasing the section size towards the center of the screen.

One could also consider dynamic sectioning schemes, where the size of sections is dynamically reduced in locations that are heavily set with triangles. Consider a region A heavily concentrated with triangles as depicted in Figure 6.2. If the screen was rasterized in the normal approach, with large sections using the tile-based rasterizer size depicted in the lower left section. The heavily concentrated region A would interfere with the rasterization of the entire top right of the screen.

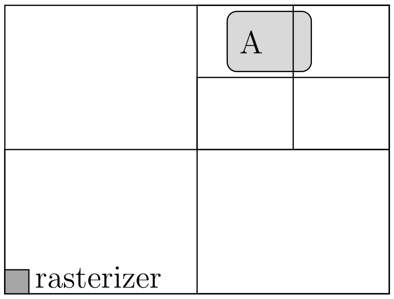


Figure 6.2: Decreasing the section size at regions with high level of detail.

A better approach would be to divide the topleft section in smaller sections. That way the two bottom parts of the section are not influenced by region A at all and the topright part is only partially influenced. Again this approach tries to minimize the number of triangles send for a single section to the direct-sorting unit. This approach should work for any application because it adjusts to the specific graphic application by creating smaller sections in areas with a high concentration of primitives. This approach does however requires extra overhead from the scenebuffer-sorting stage to be able to detemine regions that are densily populated with primitives.

6.3.3 Modifying the Rasterization Order

During our research we ensured that the rasterization order of primitives for each pixel was equal for all rasterization scenarios. This fundamental assumption is also the key why the simulator can skip texture mapping. Because of this assumption the number of texel fetches is equal for all rasterization scenarios. The rasterization order of primitives in fullscreen rasterization is fixed because primitives are send to the rasterizer directly after they are generated by the geometry stage. However, in direct tile-based rasterization we can easily chose a different rasterization order for the triangles that are currently in the triangle window.

One implementation could be to render triangles closest to the observer first by examining the depth coordinates of the triangle vertices in the sorting unit during the processing of a tile. This causes the closer primitives to be rendered first and ultimately lead to more discards in the depth test unit inside GRAAL or another OpenGL compliant tile-based rasterizer. This results is a decrease in usage of the rasterization pipeline units beyond the depth test hierarchical with less interaction to the on-chip stencil-, depth-, and framebuffers. In itself this recommendation only influences on-chip workload and does not reduce off-chip traffic so therefore it's effect on overall power consumption might be marginal. However, in combination with the next recommendation in Section 6.3.4 this approach could improve performance and reduce external traffic by removing texture fetches for triangles that are obscured by closer triangles.

6.3.4 Reducing Texture Traffic

During our research we assumed a rasterization pipeline similar to the model presented in Section A.2.3. Summarizing briefly, this model assumes fragments are first generated by some sort of rasterization unit. Consequently the parameters (depth coordinate z, colors RGBA and texture coordinates s/w, t/w and $1/w$) for the fragment are computed by an interpolation unit. In a conventional graphics pipeline and in the current design of the GRAAL tile-based rasterizer the interpolation unit computes the final color values based on the texels fetched from the external memory before passing a fragment to the depth test unit. In other words, we start testing if a fragment is actually visible only after it is completely defined.

This means that for some fragments we perform the texture fetching while the fragment could be discarded in the depth test performed imediately after texture mapping. Texture fetches are a large component of the databack traffic and it might therefore be prudent to perform some testing before the actual texels are fetched from external memory. In our simulator suite this was already assumed for the pixel ownership test and scissor test because these tests can easily be moved because they only rely on the fragment screen coordinates x and y.

The test that can actively reduce texels fetches is the depth and stencil test. It was noted that the stencil test is not used intensively by mobile applications, but the depth test is used in almost every application. Texture mapping itself is a clever technique to simplify the graphics application and is used in most applications. Alpha testing can not be used to reduce texture fetches since it relies on the fragments alpha value that can in some cases be modified by texture mapping. Therefore, by moving the depth and stencil testing in front of the texture mapping texel fetches of obscured fragments can be removed.

Consider the recommendation in Section 6.3.3, which proposed to modify the rasterization in order to increase the number of depth test fails. Increasing the number of depth test fails results in reduced workload in the tile-based rasterizer for every unit beyond the depth test unit. If the texture mapping unit is moved beyond the depth and stencil test unit, the increase number of depth test fails not only results in lower workload but also in a reduced number of texel fetches from the external memory.

Appendix: Computer Graphics A

This Appendix details some basic computations used in computer graphics and presents a commonly used inplementation of a graphics architecture in the form of a graphics pipeline. Section A.1 provides the briefest of introductions to computer graphics needed to grasp the inportance of rasterization, while Section A.2 presents the most widely used architecture to implement computer graphics in the form of a pipelined system with stages that can be implemented in both software and hardware.

A.1 Computer Graphics

This thesis mainly focuses on the implementation, and in particular the hardware acceleration, of the rasterization phase on mobile devices. In this chapter we provide a brief background on computer graphics in general, but only enough to comprehend the position and importance of the rasterization phase within the entire process of computer graphics. For a thourough background on graphics theory, including here omitted techniques like lighting, shading and texture mapping, the reader is referred to one of many excellent literary sources, for example [33].

Computer graphics is the field that allows the visualization of a three dimensional (3D) world onto a two dimensional (2D) screen with computer resources. The transformation of a 3D world defined in software to a 2D image on a screen is a computationally intensive process. This section only describes these basic computations that are required within this process:

- Scene definition (Section A.1.1),
- eyepoint and viewport selection (Section A.1.2),
- projection (Section A.1.3),
- rasterization (Section A.1.4), and
- parameter interpolation (Section A.1.5).

A.1.1 Scene Definition

Before the visualization process can actually be started an object, or a world, needs to be defined that is to be visualized. In computer graphics we refer to such an object or such a world as the scene. The scene is created in software application in a Cartesian coordinate system, where each point P is defined by $P = [x, y, z]$. This world can be anything the user of a graphics device desires to display on the screen. Usually, the scene is defined by a computer programmer and a user can virtually interact or move around the scene.

Basically a scene is created by placing primitive objects like cubes, spheres, cylinders, and other geometric objects at desired locations in an empty space. More complex objects can be constructed from a range of connected triangles. One example of such a complex object often used in computer graphics is a triangle mesh, a surface created from connected triangles commonly used to create terrain.

Objects are placed in this space by defining the objects vertices in Cartesian coordinates, referred to as the global coordinates. For example, a triangle is defined simply by defining the 3D locations of its three vertices. Geometric objects like cubes and spheres are often defined by a single center point and spacial attributes such as length and radius. At the start of the visualization process a scene is conceptually a predetermined set of simple drawing primitives defined in a 3D environment.

Moving objects within a 3D space is possible by transforming the coordinates of the objects vertices. The transformations that are used in computer graphics are always a combination of a rotation, a scalar multiplication, and/or a translation. It is common in computer graphics to describe a transformation as a multiplication with a 3×3 matrix. For example Equation (A.1) presents the matrix used to rotate a point around the z-axis with an angle ϕ.

$$\begin{bmatrix} cos\phi & -sin\phi & 0 \\ sin\phi & cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{A.1}$$

It is however impossible to define a translation transformation in a similar way with a 3×3 matrix. Equation (A.2) represents the translation of a point P with an arbitrary distance in vector form.

$$P_t = \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} x+a \\ y+b \\ z+c \end{bmatrix} \tag{A.2}$$

Where a, b, and c describe the distance to translate in the x, y, and z direction, respectively. The result of the operation is the translated point P_t. By introducing an additional coordinate, called the homogeneous coordinate (w), a translation is defined with a 4×4 matrix multiplication as in Equation (A.3).

$$P_t = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x+a \\ y+b \\ z+c \\ 1 \end{bmatrix} \tag{A.3}$$

During normal transformations the value of the homogeneous coordinate always remains equal to 1. However the homogeneous coordinate plays a vital role in perspective projection as discussed in Section A.1.3 and is required for perspective correct interpolation (Section A.1.5). In computer graphics all transformations on 3D points are defined in 4×4 matrices using the homogeneous coordinate.

A.1.2 Selecting an Eyepoint and Viewport

A scene can contain moving objects and can additionally be observed from any location within the 3D environment. Therefore, in order to display the scene on a display a snapshot is taken from a chosen location at a certain moment. In this snapshot all objects have a fixed location and the scene is also observed from a fixed position. In computer graphics we call the snapshot of the scene a frame. The frame is the start of the actual visualization process.

A frame is created by defining a point in global coordinates from where the scene is viewed, called the viewpoint or eyepoint. Based on this eyepoint we construct a new coordinate system, called the eye coordinate system with the origin at the eyepoint. The viewing direction, also known as the gaze direction g or center of projection (COP), is used directly to define the z-axis of the eye coordinate system via Equation (A.4).

$$z = \frac{g}{|g|} \tag{A.4}$$

Where $|g|$ is the length of vector g so that we acquire a coordinate vector z of unit length. The x- and y-axes of the eye coordinate system are defined perpendicular to this z vector with y being the upward direction with respect to the observer.

All vertices of all triangles in the scene are then transformed to this new eye coordinate system. The transformation to eye coordinates is achieved, like all other transformations in computer graphics, via matrix multiplications. After this coordinate transform the z-coordinate of a vertex represents the distance of that vertex with respect to the observer, while the x- and y-coordinates represent the horizontal and vertical displacement with respect to the observer as illustrated in Figure A.1.

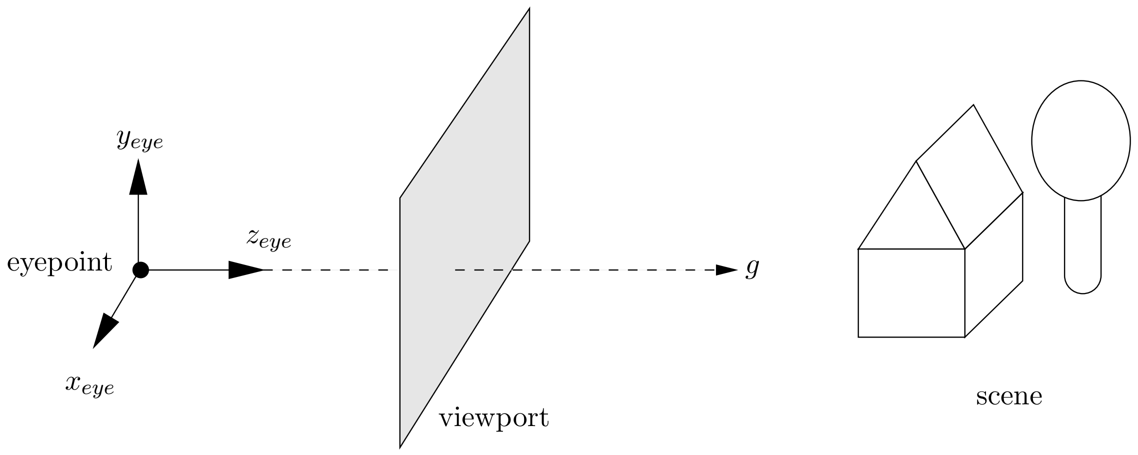


Figure A.1: The eye coordinate system.

The final step in this phase requires the user to define a plane of equal distance to the observer that represents the flat surface of the screen. This plane is referred to as the viewplane or viewport and is illustrated in Figure A.1. We now have a 3D scene defined in eye coordinates and a viewport that represents the screen of the device. Transforming the 3D scene to a 2D image is performed by the projection, the rasterization, and the interpolation phase. The computations in these phases, which represent the core of computer graphics, are presented in the next three sections.

A.1.3 Projecting the Scene on the Viewport

After all vertices of all primitives are transformed to eye coordinates and the viewport is defined we start the first computational step towards visualizing the scene, namely the projection phase. The goal of this phase is to map all the primitives defined in the eye space onto the viewport. In fact this is the phase were the 3D scene is converted to a 2D image. Two types of projection are used in computer graphics, namely orthographic projection and perspective projection. In orthographic projections the actual lengths of the primitive edges are preserved. In perspective projection the length of edges is distorted in order to provide an illusion of depth and result in a more realistic image. The difference between both projections types is illustrated in Figure A.2.

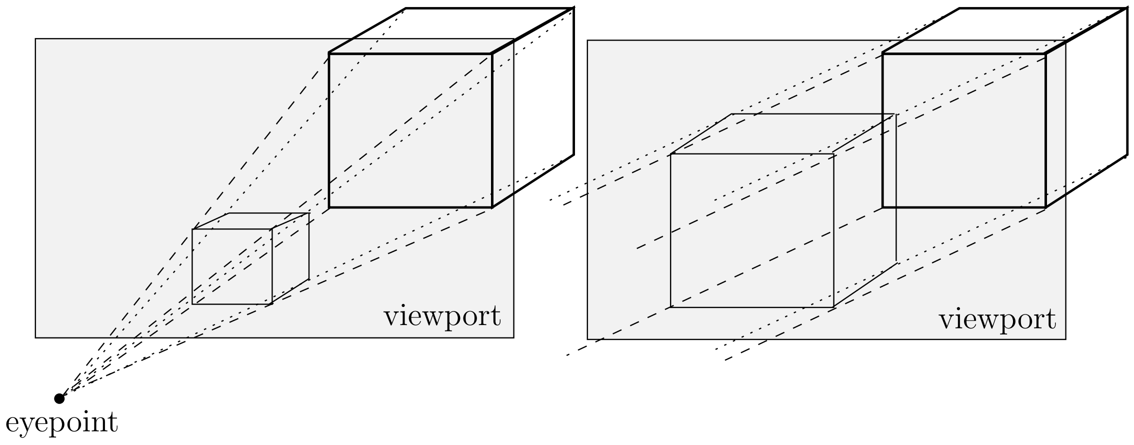


Figure A.2: Perspective (left) versus parallel (right) projection.

The perspective projection of a primitive onto a 2D surface is found by defining lines through the vertices of the primitive and the eyepoint. While the orthographic projection is found by defining parallel lines through the vertices of the primitive. When these lines are perpendicular to the viewport we speak of orthographic projection and when they have an angle with respect to the viewport we speak of oblique projection. The orthographic projection can be viewed as a special case of perspective projection with the eyepoint in infinity.

After defining the lines through the vertices the projection of a primitive onto the viewport can be found by calculating the intersections of these lines with the viewport. Since the eyepoint is defined as the origin in the new eye coordinate system, and the viewplane is a plane with equal z-coordinate the calculation of the intersections is fairly straightforward. In the case of perspective projection we use the diagram in Figure A.3.

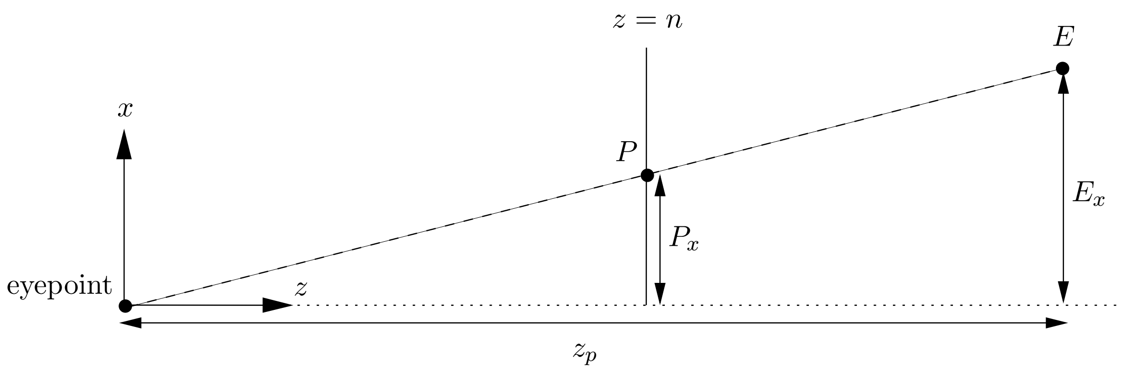


Figure A.3: Calculating the projection P of a point E.

Where $E = [E_x, E_y, E_z]$ is a point defined in eye coordinates and point $P = [P_x, P_y, P_z]$ is the projection of E onto the viewport. This figure depicts the diagram used to calculate the P_x value of the projection. A similar diagram is used for the P_y value. The line defined by $z = n$ represents the viewport, or in reality the flat surface of the screen. The P_x and P_y coordinates of a projection P of a point E can be computed via Equations (A.5) and (A.6).

$$P_x = \frac{n}{E_z} \cdot E_x \tag{A.5}$$

$$P_y = \frac{n}{E_z} \cdot E_y \tag{A.6}$$

After scaling P_x and P_y are used directly as the screen coordinates x_s and y_s needed to create the image on the screen. Since the projections are all onto the plane defined by $z = n$ the z-coordinates of the projections are not very usefull for hidden-surface removal algorithms. Therefore, E_z is not transformed, or transformed in a different manner in order to provide a useful P_z value for hidden-surface removal algorithms.

Unfortunately, division by E_z can not be achieved via matrix multiplications. In computer graphics this is again solved by utilizing the homogeneous coordinate. A perspective projection matrix is defined that calculates the projection of a point P. One example, taken from [33], is given in Equation (A.7). However, numerous other solutions exist that transform E_z differently.

$$\mathbf{M_p} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{A.7}$$

In this matrix, n denotes the z-coordinate of the viewport, in other words the distance of the observer to the viewport. f denotes the z-coordinate of the farplane, which is introduced to overcome resolution difficulties with the z-coordinate after perspective projection in actual implementations. All objects beyond the farplane are discarded.

The projection of a point $P = [P_x, P_y, P_z, P_w]$ onto the viewplane is found by multiplying point $E = [E_x, E_y, E_z, 1]$ with matrix $\mathbf{M_p}$ as in Equation (A.8).

$$\tilde{P} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} E_x \\ E_y \\ E_z \\ 1 \end{bmatrix} = \begin{bmatrix} n \cdot E_x \\ n \cdot E_y \\ (n+f) \cdot E_z - fn \\ E_z \end{bmatrix} \tag{A.8}$$

Recall from Section A.1.1 that the homogeneous coordinate w is equal to 1 for all points until the projection phase. Dividing the results of Equation (A.8) with the homogeneous coordinate, the process called homogenization in Equation (A.9), gives us the correct basis for the screen coordinates x_s and y_s calculated via Equations (A.5) and (A.6).

$$\frac{\tilde{P}}{\tilde{P}_w} = \begin{bmatrix} \frac{n}{E_z} \cdot E_x \\ \frac{n}{E_z} \cdot E_y \\ n + (1 - \frac{n}{E_z})f \\ \frac{E_z}{E_z} \end{bmatrix} = \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} \tag{A.9}$$

In this example of the perspective matrix the z coordinate of objects in eye space (E_z) is mapped to a P_z value in the region $[n, f]$. The exact value of the P_z-coordinate is no longer correct, but can still be used for hidden-surface removal algorithms as it still correctly represents the object order with respect to the observer. Other instances of the perspective matrix can map E_z to a different range, for example $[-1, 1]$.

A.1.4 Rasterization of the Projection

After the projection phase we have a 2D image to be displayed, namely the projections onto the viewplane of all the simple drawing primitives in the scene. Unfortunately, we only have a discrete number of pixels to display the image. The number of pixels we have to represent the image is dictated by the resolution of the screen. In order to display the 2D image on the screen the image needs to be sampled. This sampling process is referred to as rasterization computer graphics, due to the fact that most contemporary displays have pixels orientated in a fixed raster.

Each drawing primitive is defined by a discrete set of 3D points. For example, a triangle is represented by three points, and a square by four points. During the rasterization phase we identify all pixels in the screen that are inside a primitive. Several solutions to this problem exist, such as scanline conversion [33] and linear edge testing [31].

The scanline conversion algorithm is a process that iteratively traverses pixel centers from the left to the right (or vice versa) along a horizontal line, called a scanline. This algorithm first computes the intersections of the scanlines with the edges of the primitive as indicated in Figure A.4 for the triangle ABC. For primitives defined by more points than three this process becomes more complex because those primitives might not be

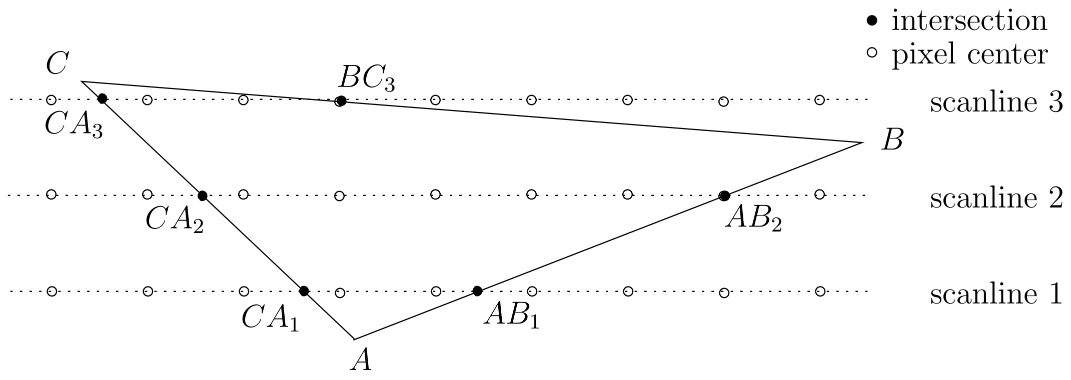


Figure A.4: Determining the intersections of scanlines and triangle edges.

convex and have several intersections along a scanline. Therefore actual designs that perform rasterization are based on the rasterization of triangles.

The intersections of edge AB with the scanlines can be computed with Equation (A.10). The intersections of the scanlines with the other edges are computed with similar equations.

$$x_{intersect}(AB, i) = x_A + (x_B - x_A) \cdot \frac{(y_i - y_A)}{(y_B - y_A)} \tag{A.10}$$

In this equation y_i denotes the y coordinate of scanline i. When the two intersections for a scanline are determined the algorithm identifies the pixels centers that lay between the two intersections as inside the primitive. When all the pixels on a scanline are identified the next scanline is selected and processed. This process continues until the entire primitive is processed. Figure A.5 illustrates the order in which triangle ABC is processed and which pixels are identified as being inside the primitive.

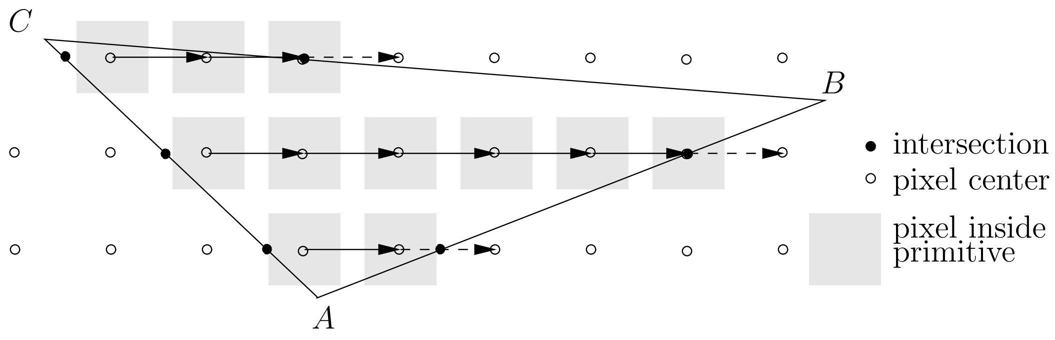


Figure A.5: Traversal of a triangle primitive with scanline conversion.

Basically the algorithm traverses pixel centers on a scanline from left to right until it finds a pixel center that is located outside the triangles edges. These pixels are not influenced by the primitive and upon encountering such a pixel the algorithm moves to the next scanline. The pixels identified as inside the triangle by this algorithm are signified by darkened squares.

A.1.5 Interpolation of per pixel Parameters

The depth (z), color (c), and possible texture parameters (s, t) for a primitive are only defined in the vertices of the primitive. After a primitive is rasterized the correct values for these parameters need to be interpolated for all the pixels inside the triangle based on the values of the parameters in the vertices. In computer graphics two types of interpolation are used to compute parameter values for pixels inside primitives, namely linear interpolation and hyperbolic interpolation.

Consider a triangle defined by vertices A, B, and C, where each vertex is defined by its screen coordinates (x, y) and several parameter values (p_1, p_2, ... , p_n) such as depth value, color values, or texture coordinates. Vertices A, B, and C can then be represented as:

$$\begin{aligned} A &= (x_A,\, y_A,\, p_{1A},\, p_{2A},\, \dots\,,\, p_{nA}) \\ B &= (x_B,\, y_B,\, p_{1B},\, p_{2B},\, \dots\,,\, p_{nB}) \\ C &= (x_C,\, y_C,\, p_{1C},\, p_{2C},\, \dots\,,\, p_{nC}) \end{aligned}$$

In linear interpolation we seek linear functions that uniquely define a parameter value for every point inside the triangle. The function to compute the parameter value p_i for any given point has the following form:

$$p_i(x, y) = c_0 \cdot x + c_1 \cdot y + c_2, \tag{A.11}$$

where the coefficients c_0, c_1 and c_2 can be calculated by noting the fact that the three vertices are correct solutions for the linear function. Hence the function for parameter p_i must conform to the following set of equations:

$$c_0 \cdot x_A + c_1 \cdot y_A + c_2 = p_{iA} \tag{A.12}$$

$$c_0 \cdot x_B + c_1 \cdot y_B + c_2 = p_{iB} \tag{A.13}$$

$$c_0 \cdot x_C + c_1 \cdot y_C + c_2 = p_{iC} \tag{A.14}$$

After solving the set of three equations for the coefficients c_0, c_1, and c_2 for every parameter (p_1, p_2, ... , p_n) we have a set of linear functions that allow us to compute the parameters for any point inside the triangle primitive. In real implementations we do not actually compute the entire linear function since we only need the parameter values for discrete locations, namely pixel centers. The linear function in (A.11) is therefore often defined in incremental form as in Equation (A.15).

$$p_i(x, y) = p_{init} + \frac{dp_i}{dx} \cdot \Delta x + \frac{dp_i}{dy} \cdot \Delta y \tag{A.15}$$

Where p_{init} is the value for the parameter p_i calculated once for an initial point (x_i, y_i) and Δx and Δy are the horizontal and vertical difference between a point

(x, y) and the initial point defined as $\Delta x = x - x_i$ and $\Delta y = y - y_i$. Computing the parameters in incremental form is a lot simpler to implement in specialized hardware than computing the parameter values via Equation (A.11) for every pixel location.

The drawback of linear interpolation is that it assumes that equidistant steps in the screen space, mainly steps from pixel to pixel, correspond to equidistant steps in eyespace. When perspective projection is used however, this is only valid for surfaces and lines that are coplanar with the viewport. Figure A.6 illustrates how linear interpolation along pixel centers in the viewport can result in non-linear interpolation along a surface or line in eye space.

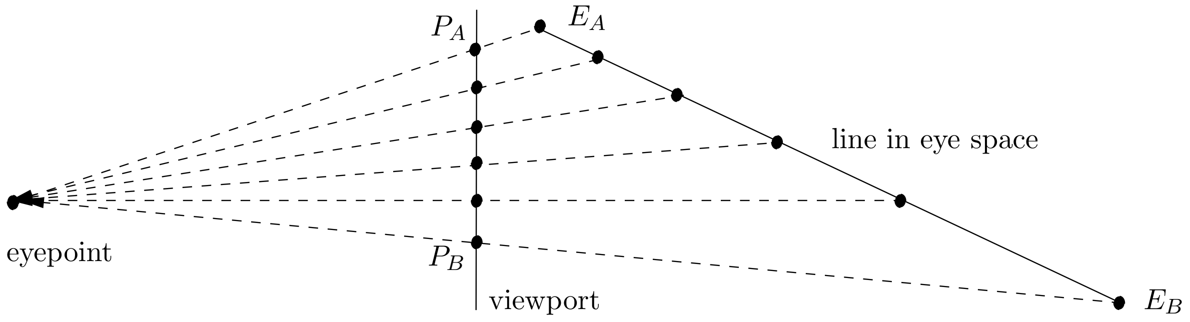


Figure A.6: Equidistant interpolation steps in screen space map to non-equidistant interpolation steps in eye space.

By incorrectly assuming that equidistant steps in the screen space correspond to equidistant steps in the eye space an error is introduced in the computation of the parameter values. Figure A.7 shows the mismatch of the computed parameter values with linear interpolation and the actual values of the parameters. In this figure the parameter value is assumed to be 0 at point P_A and 1 at point P_B.

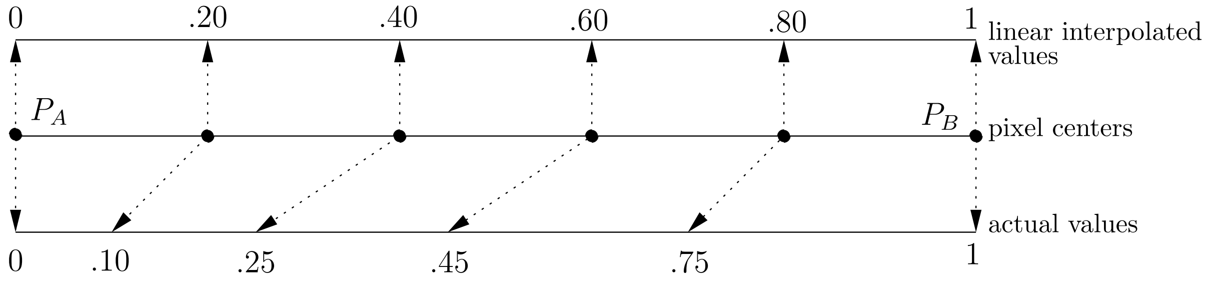


Figure A.7: Linear interpolation parameter mismatch.

While not providing a correct result linear interpolation is used to compute most parameter values for pixels inside a primitive. The depth value z is only used for hidden-surface removal and linearly interpolating this value does not lead to complications. In most applications the color values are also interpolated linearly in order to simplify the interpolation process. Eventhough the resulting image is not perspective correct the resulting image is often still acceptable to the human eye.

Hyperbolic interpolation [11] is used when we need perspective correct interpolation of the parameter values. In hyperbolic interpolation the homogeneous coordinate is again utilized to overcome the perspective projection distortion of sizes. All parameters for a vertex that require hyperbolic interpolation are divided by the homogeneous coordinate w and the reciprocal of the homogeneous coordinate is added to the vertex. Vertices A, B, and C are then represented as:

$$\begin{aligned} A &= (x_A,\, y_A,\, \tfrac{p_{1A}}{w_A},\, \tfrac{p_{2A}}{w_A},\, \dots\, ,\, \tfrac{p_{nA}}{w_A},\, \tfrac{1}{w_A}) \\ B &= (x_B,\, y_B,\, \tfrac{p_{1B}}{w_B},\, \tfrac{p_{2B}}{w_B},\, \dots\, ,\, \tfrac{p_{nB}}{w_B},\, \tfrac{1}{w_B}) \\ C &= (x_C,\, y_C,\, \tfrac{p_{1C}}{w_C},\, \tfrac{p_{2C}}{w_C},\, \dots\, ,\, \tfrac{p_{nC}}{w_C},\, \tfrac{1}{w_C}) \end{aligned}$$

In order to find the parameter values for a parameter p_i in an arbitrary point $(x,\, y)$ we use linear interpolation of the $\frac{p_i}{w}$ value and the $\frac{1}{w}$ value via Equation (A.11) or its incremental implementation (A.15). The perspective correct value for parameter p_i is found by dividing the interpolated $\frac{p_i}{w}$ value by the $\frac{1}{w}$ value.

A.2 Graphics Pipeline

The previous section described the theoretical basis and computations in computer graphics. This section describes the most widely used implemention of a system capable of rendering 3D scenes to a 2D screen. The system is build as a pipeline and hence referred to as the graphics pipeline. The pipeline is implemented in three stages: Application, Geometry, and Rasterization stage, as depicted in Figure A.8.

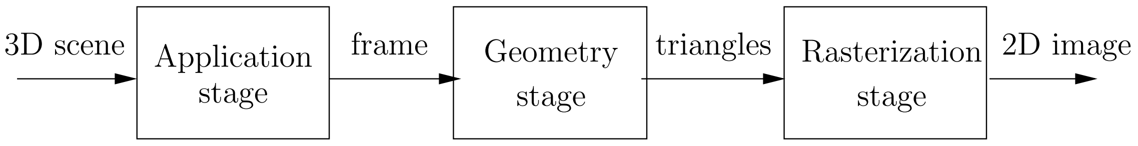


Figure A.8: Three computation stages in a graphics pipeline.

In this section we assume that the graphics pipeline is implemented in compliance with the OpenGL [32] Application Programming Interface (API). OpenGL provides a software programmer with the basic building blocks to construct a 3D world in software. OpenGL is adopted as the industry standard for computer graphics.

In certain parts we present specific OpenGL function calls required to perform certain operations. These will be limited to some global calls and calls that modify the computations inside the rasterization stage.

The exact parameters to these function calls are omitted and for the exact specifics about these OpenGL we refer the reader to one of the many excellent OpenGL on line manual pages[30]. The main focus in this research involves the rasterization stage of the graphics pipeline and this stage is therefore described in more detail than both the application and geometry stage.

A.2.1 Application Stage

In the application stage the input for the graphics pipeline is defined by the user as a virtual world in 3D space with a Cartesian coordinate system as described in Section A.1.1. All interaction with the user is restricted to the application stage. In this stage possible movement of objects, in the case of animated scenes, and placement are preformed on objects in the scene. Movement and placement of an object are performed by matrix multiplications on the vertices of the object.

In order to display a scene defined in any arbitrary 3D space the user specifies the eyepoint and a viewing volume. For perspective projection this volume is a frustum as depicted in Figure A.9 and for parallel projection this volume is a rectangular box. The surface of the viewing volume nearest to the observer is the viewport as described in Section A.1.2. Only the objects in the scene inside the viewing volumes are processed. All objects in the scene outside this volume are discarded and objects partially within this volume are clipped in the geometry stage.

The application stage is highly dependent on the nature of the application and the desires of the user. The application stage is therefore usually implemented in software.

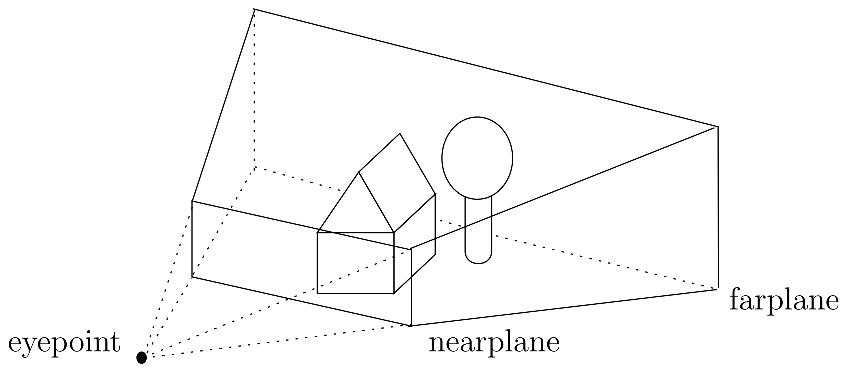


Figure A.9: Defining the viewing volume

OpenGL allows the user to construct a scene using simple geometric objects with function calls like `glSphere(...)` and `glCube(...)`, etc. In OpenGL objects are defined directly in eye space and by default placed at the origin. By defining translations with `glTranslate(...)` and rotations with `glRotate(...)` objects can be placed at any arbitrary location in eye space. Alternatively, the user can directly define polygons or triangle strips by defining the vertices using the `glVertex(...)` call.

By default the eyepoint in OpenGL is placed at the origin and the observer is looking at the direction of the negative z axis. The user can specify if parallel or perspective projection is used by calling either `gluOrtho2D(...)` or `gluPerspective(..)`. Note that there is no specific OpenGL function to specify an oblique projection. The call to `gluPerspective(..)` automatically sets up the viewing frustum as depicted in Figure A.9.

A.2.2 Geometry Stage

After the 3D world is completely defined in global coordinates in the application stage, the geometry stage is responsible for converting the 3D objects defined by the user to simple 2D primitives to be rasterized by the next and final stage. In actual implementations of computer graphics all primitives are broken down to triangles, since triangles are relatively easy to rasterize and interpolate in hardware, when compared to polygons defined by more than three vertices. The geometry stage performs the following functions, lighting and shading, projection, clipping, and screen mapping.

The lighting and shading function performs computations that simulate the effect of one or several sources of light in the scene defined by the user with a call to `glLight(...)`. These computations affect the color of the triangles vertices that are used in the rasterization stage to compute pixel colors. When Gouraud [23] shading is used this function computes a normal vector for each vertex, based on the normal vectors of the surfaces directly surrounding the vertex. The angle of this normal vector with the incident light rays determines the shade of the vertex color.

Probably the most important function of the geometry state is the projection function. This function transforms the 3D scene to a 2D projection of the scene on the viewport as described in Section A.1.3. As described either parallel or perspective projections can be used as desired by the user by calling either `gluOrtho2D(...)` or `gluPerspective(..)`.

After projection the geometry stage performs clipping on primitives that are on the boundary of the viewing volume. If necessary primitives on the boundary of the volume are broken down in several primitives to ensure that every primitive is inside the viewing volume.

The final function, screen mapping, scales the x and y coordinate of the primitives into screen coordinates that are directly used to index the pixels of the screen. The scale factors are again derived from the viewing volume defined by the user in the application stage.

In the geometry stage calculations are still done on a per-vertex or per-object basis. Some implementations of the graphics pipeline have implemented certain functions, usually lighting and shading, of the geometry stage in hardware. Most graphics pipelines however perform the geometry stage entirely in software.

A.2.3 Rasterization Stage

The next step in displaying the scene onto a 2D display is the rasterization of the primitives as defined in screen coordinates by the geometry stage. The term rasterization is used when an object of higher resolution is visualized on a screen with only limited pixel positions organized in a raster. Due to the fact that the number of calculations from this point on increases substantially the rasterization stage is usually implemented in special hardware.

To keep the rasterization hardware as simple as possible the basic primitive used in rasterization in computer graphics is a triangle. Triangles are suitable as an input for accelerator hardware because they are always planar and convex. Table A.1 presents the usual representation of the vertices that compose a triangle when it is passed to the rasterizing stage.

parameter	description
x, y	screen coordinates
z	depth coordinate
r, g, b	color values
a	alpha value (used for visual effects)
s, t	texture coordinates
$1/w$	homogeneous coordinate

Table A.1: Triangle vertex parameters.

As calculated during projection in the geometry stage we have the location of the vertices in their screen coordinates x and y. Each vertex is further associated with several parameters like, the depth coordinate z, the color and alpha values, and the texture coordinates used to map a preprocessed 2D image on the surface of the triangle. Finally, in order to allow hyperbolic interpolation for parameters that require perspective correct interpolation, as described in Section A.1.5, the $1/w$ parameter is added to each vertex.

The rasterization stage itself is again implemented in a pipelined fashion. One implementation, but not necessarily the only possible one, of the rasterization pipeline is presented in Figure A.10. In this pipeline triangles are introduced at the first stage defined in the parameters as explained above. All the pixels inside a triangle, identified via either scanline conversion or linear edge testing as explained in Section A.1.4, are send along the next stages of the pipeline as fragments. A fragment is a distinct pixel location with the associated parameter values interpolated based on the parameter values in the triangles vertices. It is important to note that a single triangle can result in any number of fragments. The number of computations in the rasterization stage therefore increases dramatically when compared to the geometry stage.

Some of the resulting parameter values for pixels are stored in special buffers during the rasterization of triangles. Since multiple triangles can map to the same pixel we need to store the parameter values during rasterization in order to determine the final pixel color values based on all the triangles that influence the pixel. Values that need to be stored during rasterization include the color values and depth coordinates. A new parameter introduced by the OpenGL standard, the stencil value, is also stored for each pixel. These buffers are referred to as the framebuffer for the color values, the depth buffer for depth values, and the stencil buffer for stencil values. Before the rasterization of triangles for a frame is started it is common to clear the content of these buffers with a call to the `glClear(...)` function.

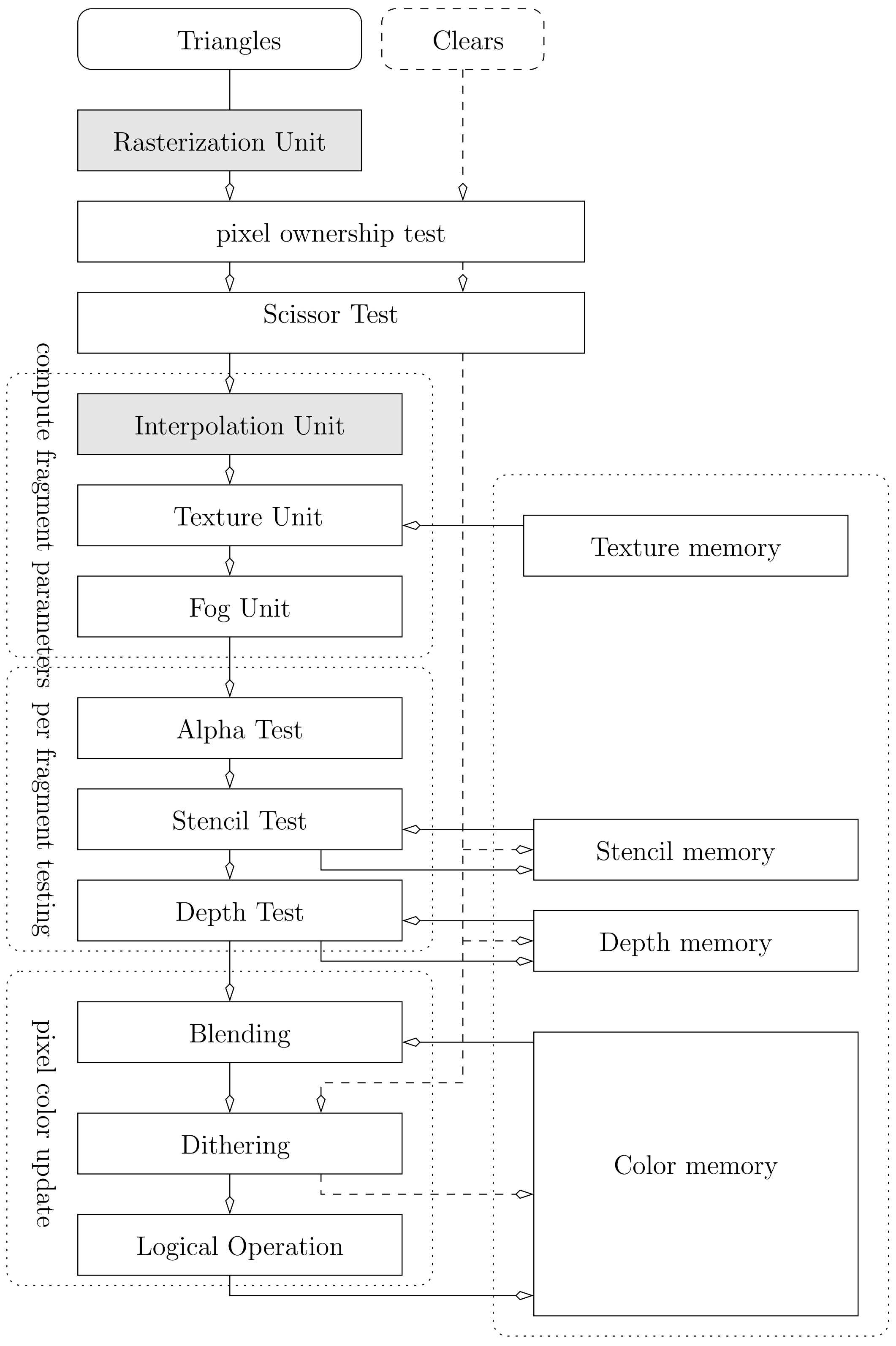


Figure A.10: Rasterization Pipeline in Computer Graphics.

In the rasterization pipeline several units are present that are required in order to comply with the OpenGL standard. OpenGL provides the user with several functions that are used to implement special imaging techniques, like color blending and texture mapping. The exact settings for these units can be defined by the user by performing special OpenGL function calls.

The first unit in the pipeline indicated by a darkened rectangle, rasterization, performs the rasterization of a triangle as described in Section A.1.4. This unit utilizes either scanline conversion or linear edge testing to identify pixels inside the current triangle. As aleady mentioned the computations for linear edge testing are independent for each pixel and are therefore perfectly suited to be implemented in hardware. The result of the rasterization is a stream of fragments representing pixels inside the triangle.

The pixel ownership test is used to determine the owner of a certain pixel in the case that multiple OpenGL programs are running in a windowed environment. If these windows overlap the top window is the owner of the corresponding pixels where the two windows overlap. An OpenGL application can not modify pixels in the framebuffer that are not owned. The setting for the ownership test should be obscured from the user and handled automatically.

The scissor test as defined by the OpenGL can discard fragments based on the pixel location. The user defines a scissor rectangle by using the function call `glScissorRect(...)`. All fragments outside this rectangle are discarded and pixels outside the rectangle are not modified by the rasterization pipeline. The scissor test is the only OpenGL test that also applies to calls to `glClear(...)` used to clear the color, depth, and stencil values in the memory.

The interpolation unit is responsible for computing the parameter values for the fragments inside the current triangle. As required by the application either linear interpolation or hyperbolic interpolation, as presented in Section A.1.5, can be utilized. After this unit a fragment is ready for its traversal down the pipeline and is it represented as:

$$frag = (x, y, z, r, g, b, a, s, t),$$

where all parameters correspond with the parameters in Table A.1.

The following unit, the texture unit, is responsible for texture mapping, a technique used to overlay 2D images stored in memory, called textures, onto surfaces of an object. When texture mapping is enabled the texture unit uses the s and t parameter of the fragment to determine which value of the texture to retrieve. The value retrieved from the texture memory is called a texel and is used to either modify the color or the depth of the fragment. Dependent on the desired image quality multiple texels can be retrieved in order to calculate the final color of a single fragment by averaging between the retrieved texels.

The fog unit is added to the graphics pipeline to provide the user with a means of creating fog in a 3D scene. The fog parameters can be set with a call to `glFog(...)`. This unit obscures objects behind a layer of ”fog” by blending the incoming fragment color with a fog color with a certain blending factor that is dependent on the z-coordinate. For objects increasingly further from the observer the fog blending factor increases and objects are less visible for the observer.

The alpha test is a simple test that discards fragments from the pipeline based on the alpha value of the fragment. The alpha test is often used to discard ”invisible” texels in texture mapping. The alpha test is a simple relational test, for example $a > a_{ref}$, that determines if a fragment is to traverse further down the pipeline. Where a_{ref} is a parameter provided by the user with a call to `glAlphaFunc(...)`.

The stencil test is also a simple relational test like the alpha test described above, but the stencil test is not based on any parameter within the incoming fragment. Instead the stencil test is performed on the value currently present in the stencilbuffer for the pixel corresponding with the fragment. Fragments that do not pass the test are discarded from the pipeline and based on the outcome of the stencil test[1] an update is made to the stencil buffer at the location of the fragment. Stencil testing can be used in complex scenes in order to determine shadows via the use of shadow volumes [20]. The parameters of the stencil test can be set with a call to `glStencilFunc(...)`, while the desired update to the stencilbuffer can be set with a call to `glStencilOp(...)`.

The depth test is the implementation of the hidden-surface removal algorithm in the graphics pipeline. Objects that are far from the observer can be entirely or partially hidden behind objects that are closer to the observer. This test provides a per pixel comparison of fragment depth values (z). By using a depth buffer we remove the need to sort the incoming primitives from back to front. The depth value of the last fragment that passed the depth test is stored in the depth buffer. Fragments that do not pass the test are discarded, while fragments that pass the test cause the depth value in the depth buffer to be updated before traversing to the next unit in the pipeline. The depth test is the only OpenGL test that is enabled by default and its parameters can be set with a call to `glDepthFunc(...)`.

The blending unit is used to blend the incoming fragment color with the color currently present in the framebuffer. Blending is often used to handle transparent objects by allowing objects behind a transparent object to still influence the final image. If transparency is used the alpha value of the incoming fragments and the alpha value currently stored in the framebuffer are used to determine the blending factors. Blending is disabled by default and blending settings can be modified with a call to `glBlend(...)`. If the blending function is disabled the fragment color parameters are simply maintained.

The dithering unit is used to enhance the quality of the image when only a limited color precision is available. For example, if there are only 4 bits allocated to represent color parameters large flat surfaces are likely to be in exactly the same color value. This appears unnatural and degrades image quality. Dithering applies a random noise factor to a fragments color value based on the screen location of the fragment. The implementation of the dither unit is machine dependent (based on the color precision) and the user can only enable or disable it in OpenGL by calling `glEnable(GL_DITHER)` or `glDisable(GL_DITHER)`. By default dithering is enabled in OpenGL.

The logical operation unit is the final unit in the graphics pipeline and performs a logical operation on the color of the incoming fragment and the color value currently stored in the framebuffer. Several different types of logical operation can be selected by the user by calling `glLogicOp(...)`, among these are XOR, NOR, INVERT, etc. It is

[1]If the stencil test passes the outcome of the depth test also influences the update

important to note that enabling the logical operation disables the blending operation.

After the logical operation unit the resulting color values in the fragment are stored in the framebuffer. For every frame all triangles are send to the rasterization pipeline and every triangle results in a number of fragments send down the pipeline. After all the triangles in the frame are processed in this way the framebuffer holds the resulting 2D image of the scene to be displayed on the screen.

To summarize, common implementations of the graphics pipeline are constructed in three stages, the application stage, geometry stage and rasterization stage. After all the triangles in the scene have passed these pipeline stages the color values in the framebuffer are ready to be diplayed on the screen. All user interaction is restricted to the application stage and after this stage a scene is defined in a 3D coordinate system. Due to the application dependent nature this first stage is usually implemented in software.

The geometry stage projects the objects in the scene onto a 2D plane and decomposes all polygons to triangles to serve as input for the final, rasterization stage. The geometry stage can be implemented either entirely in software, entirely in hardware or a combination of hardware and software depending on the required performance and available hardware budget.

The rasterization stage, as its name implies, perfroms the mapping from triangles to screen pixels. In this process the most important computations involve the rasterization of triangles and the parameter interpolation in order to produce fragments. The fragments are subjected to several graphics tests and eventually result in updates to the framebuffer. The rasterization computations are performed per fragment and the number of computations explodes when compared to the application and geometry stages, where computations are performed per vertex. Therefore rasterization is usually implemented in hardware to order to increase performance and to reduce workload on the main processor.

Bibliography

[1] Tomas Akenine-Möller and Jacob Ström, *Graphics for the masses: a hardware rasterization architecture for mobile phones*, ACM Trans. Graph. **22** (2003), no. 3, 801–808.

[2] I. Antochi, *Suitability of tile-based rendering for low-power 3d graphics accelerators*, Ph.D. thesis, Technical University Delft, October 2007, p. 148.

[3] I. Antochi, B.H.H. Juurlink, A. G. M. Cilio, and P. Liuha, *Trading efficiency for energy in a texture cache architecture*, Proceedings of the 2002 Euromicro conference on Massively-parallel computing systems, April 2002, pp. 189–196.

[4] I. Antochi, B.H.H. Juurlink, and S. Vassiliadis, *Selecting the optimal tile size for low-power tile-based rendering*, Proceedings ProRISC 2002, November 2002, pp. 1–6.

[5] I. Antochi, B.H.H. Juurlink, S. Vassiliadis, and P. Liuha, *Efficient state management for tile-based 3d graphics architectures*, Proceedings of the 15th Annual Workshop on Circuits, Systems and Signal Processing, ProRISC 2004, November 2004, pp. 336–340.

[6] ———, *Graalbench: A 3d graphics benchmark suite for mobile phones*, Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools, June 2004, pp. 1–9.

[7] ———, *Memory bandwidth requirements of tile-based rendering*, Proceedings of the Third and Fourth International Workshops SAMOS 2003 and SAMOS 2004 (LNCS 3133), July 2004, pp. 323–332.

[8] ———, *Scene management models and overlap tests for tile-based rendering*, Proceedings of the EUROMICRO Symposium on Digital System Design, 2004 (DSD 2004)., August 2004, pp. 424 – 431.

[9] *Arm graphics solutions overview*, http://www.arm.com/products/multimedia/graphics/index.html.

[10] Anthony C. Barkans, *High quality rendering using the talisman architecture*, HWWS '97: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware (New York, NY, USA), ACM, 1997, pp. 79–88.

[11] James F. Blinn, *Hyperbolic interpolation*, IEEE Computer Graphics and Applications, IEEE, july 1992, pp. 89–94.

[12] Businessweek, *Pwc: Video game industry to drive entertainment sector.*, http://www.businessweek.com/innovate/content/oct2005/id20051007_999151.htm, 2005.

[13] Milton Chen, Gordon Stoll, Homan Igehy, Kekoa Proudfoot, and Pat Hanrahan, *Simple models of the impact of overlap in bucket rendering*, HWWS '98: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware (New York, NY, USA), ACM, 1998, pp. 105–112.

[14] D. Crisu, S. D. Cotofana, S. Vassiliadis, and P. Liuha, *3d graphics tile-based systolic scan-conversion*, Thirty-Eighth Asilomar Conference on 3d graphics, November 2004, pp. 517 – 521.

[15] ______, *Efficient hardware for antialiasing coverage mask generation*, Proceedings of Computer Graphics International Conference 2004 (CGI 2004), June 2004, pp. 257–264.

[16] ______, *Graal - a development framework for embedded graphics accelerators*, Proceedings of Design, Automation and Test in Europe (DATE'04), February 2004, pp. 1366–1367.

[17] ______, *Logic-enhanced memory for 3d graphics tile-based rasterizers*, Proceedings of the 2004 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS 2004), July 2004, pp. II–237–II–240.

[18] D. Crisu, S. Vassiliadis, and S. D. Cotofana, *A proposal of a tile-based opengl compliant rasterization engine*, 2002.

[19] D. Crisu, S. Vassiliadis, S. D. Cotofana, and P. Liuha, *Low cost and latency embedded 3d graphics reciprocation*, Proceedings of 2004 IEEE International Symposium on Circuits and Systems (ISCAS 2004), May 2004, pp. II–905 – II–908.

[20] Cass Everitt and Mark J. Kilgard, *Practical and robust stenciled shadow volumes for hardware-accelerated rendering*, 2003.

[21] Richard Fromm, Stylianos Perissakis, Neal Cardwell, Christoforos Kozyrakis, Bruce Mcgaughy, David Patterson, Tom Anderson, and Katherine Yelick, *The energy efficiency of iram architectures*, In the 24th Annual International Symposium on Computer Architecture, 1997, pp. 327–337.

[22] Tohru Furuyama, *Trends and challenges of large scale embedded memories*, Proceedings of the IEEE 2004 Custom Integrated Circuits Conference, 2004, 2004, pp. 449–456.

[23] H. Gouraud, *Continuous shading of curved surfaces*, IEEE Trans. Comput. **20** (1971), no. 6, 623–629.

[24] Khronos group, *Opengl es - the standard for embedded accelerated 3d graphics*, http://www.khronos.org/opengles/, 2008.

[25] Ziyad S. Hakura and Anoop Gupta, *The design and analysis of a cache architecture for texture mapping*, ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture, 1997, pp. 108–120.

[26] Kiyoo Itoh, *Trends in low-voltage embedded-ram technology*, Proceedings of the 23rd International Conference on Microelectronics, volume 2, 2002, may 2002, pp. 497–501.

[27] Kiyoo Itoh, Katsuro Sasaki, and Yoshinobu Nakagome, *Trends in low-power ram circuit technologies*, Proceedings of the IEEE, volume 83, issue 4, 1995, april 1995, pp. 524–543.

[28] ARM Limited, *AMBA^{TM} specification*, 1999.

[29] *The Mesa 3D Graphics Library*,
http://www.mesa3d.org.

[30] *OpenGL Manual Pages*,
http://www.opengl.org/sdk/docs/man/.

[31] J. Pineda, *A parallel algorithm for polygon rasterization*, SIGGRAPH '88 (15th Annual Conference on Computer Graphics and Interactive Techniques, Atlanta, GA, August 1–5, 1988), 1988, pp. 17–20.

[32] M. Segal and K. Akeley, *The opengl graphics system: A specification*, 1999.

[33] P. Shirley, *Fundamentals of computer graphics*, A.K. Peters, Ltd, 2005.

[34] *Nintendo DS (WIKI)*,
http://en.wikipedia.org/wiki/Nintendo_DS.

[35] *PSP(WIKI)*,
http://en.wikipedia.org/wiki/PSP.