Abstract

Graphics on a computer are often handled by a graphics pipeline. Rasterization is an important stage in this pipeline. It converts the basic elements of computer graphics, triangles, into the basic elements of a screen, pixels. This stage is very computation intensive and has a large memory footprint. The last decade a lot of research has been dedicated to tile-based rasterization. This technique divides an image into smaller images called tiles. These can be stored on a smaller memory, hence reducing the memory footprint of the rasterization process. Several software and hardware implementations of tile-based rasterization exist. They use a single general purpose processor, or make use of multiple specialized cores, such as Graphical Processing Units (GPUs). Although GPUs prove to be very fast in graphical applications, their effectiveness in running other applications is limited. This is a potential drawback in embedded systems where the available resources are very limited. In Embedded Systems typically Multiple Processors are used in a System on Chip, an MPSoC. This work will make a parallelization study to investigate the performance of rasterization on a MPSoC. To perform this study we modify a tile-based rasterizer to make efficient use of multiple embedded processors. This modified rasterizer is used to evaluate the impact of various configurations on the execution time of the rasterizer. For example the size of an image tile, the size of the communication buffer and the number of processors will be varied in our experiments. Increasing the size of an image tile proves to decrease the execution time only to a certain point, after which the execution time will start to increase. Furthermore, using a larger communication buffer increases the load balancing and decreases the execution time. It is concluded that with the right load balancing adding processors to the system will decrease the time needed to rasterize an image.
Tile-Based rasterization on an embedded Tile-based MPSoC
Implementation of the Rasterization stage of the OpenGL pipeline on a multi processor system on chip

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Johan Splinter
born in Purmerend, The Netherlands
Tile-Based rasterization on an embedded 
Tile-based MPSoC

by Johan Splinter

Abstract

Graphics on a computer are often handled by a graphics pipeline. Rasterization is an important stage in this pipeline. It converts the basic elements of computer graphics, triangles, into the basic elements of a screen, pixels. This stage is very computation intensive and has a large memory footprint. The last decade a lot of research has been dedicated to tile-based rasterization. This technique divides an image into smaller images called tiles. These can be stored on a smaller memory, hence reducing the memory footprint of the rasterization process. Several software and hardware implementations of tile-based rasterization exist. They use a single general purpose processor, or make use of multiple specialized cores, such as Graphical Processing Units (GPUs). Although GPUs prove to be very fast in graphical applications, their effectiveness in running other applications is limited. This is a potential drawback in embedded systems where the available resources are very limited. In Embedded Systems typically Multiple Processors are used in a System on Chip, an MPSoC. This work will make a parallelization study to investigate the performance of rasterization on a MPSoC. To perform this study we modify a tile-based rasterizer to make efficient use of multiple embedded processors. This modified rasterizer is used to evaluate the impact of various configurations on the execution time of the rasterizer. For example the size of an image tile, the size of the communication buffer and the number of processors will be varied in our experiments. Increasing the size of an image tile proves to decrease the execution time only to a certain point, after which the execution time will start to increase. Furthermore, using a larger communication buffer increases the load balancing and decreases the execution time. It is concluded that with the right load balancing adding processors to the system will decrease the time needed to rasterize an image.

Laboratory : Computer Engineering
Codenumber : CE-MS-2011-26

Committee Members :

Advisor: Prof. Sorin Cotofana, CE, TU Delft
Advisor: Dr. Anca Molnos, CE, TU Delft
Member: Prof. Kees Goossens, TU Eindhoven
Member: Dr.ir. A.J. van Genderen, CE, TU Delft
“Nothing is particularly hard if you divide it into small jobs”

–Henry Ford

iii
## Contents

List of Figures ................................................. x

List of Tables ............................................... xi

Acknowledgements .............................................. xiii

1 Introduction .................................................. 1
   1.1 Contributions ........................................... 4
   1.2 Overview ................................................. 6

2 Background and Related Work ............................... 9
   2.1 Background on Rasterization ........................... 9
      2.1.1 OpenGL and Rasterization ......................... 9
      2.1.2 Tile-based rasterization .......................... 10
   2.2 Background on Platforms and Processors .............. 14
      2.2.1 Trends in Processors, Energy and Memory ......... 14
      2.2.2 Processing Units ..................................... 14
      2.2.3 Programmable Stream processors ................ 16
   2.3 Related work ............................................. 17
      2.3.1 Research on tile-based rasterization ............. 17
      2.3.2 Software implementations of rasterizers ......... 17
   2.4 Summary .................................................. 19

3 Parallel tile-based rasterizer on a desktop PC ........... 21
   3.1 Initial tile-based rasterizer .......................... 21
      3.1.1 Input of the initial rasterizer ................. 21
      3.1.2 Functionality of the tile-based rasterizer ....... 22
   3.2 Parallel Tile-based Rasterizer ........................ 25
      3.2.1 Queen ................................................ 26
      3.2.2 Worker .............................................. 26
      3.2.3 Communication ....................................... 26
      3.2.4 Removal of the tile switch ....................... 29
      3.2.5 Summary .............................................. 30

4 Parallel tile-based rasterizer on CoMPSoC .............. 31
   4.1 CoMPSoC platform ........................................ 31
      4.1.1 Processing tile ....................................... 31
      4.1.2 CompOSE ............................................. 32
      4.1.3 Applications and Tasks ......................... 33
      4.1.4 The monitor .......................................... 33
List of Figures

1.1 Example of a CoMPSoC configuration .................................. 3
2.1 Creating a tilemask .................................................... 12
2.2 A Process Network .................................................... 16
3.1 Hexadecimal presentation of the TKI-instructions .................... 23
3.2 A polygon can be divided into triangles ............................ 23
3.3 Structure of an element ............................................. 25
3.4 The Queens sends data to be processed to multiple Workers .... 26
3.5 The different forms of communication between the Queen and a Worker .................................................... 27
3.6 Structure of the packed element .................................... 29
3.7 The Queens sends packed elements to multiple Workers ........ 29
4.1 Processing Microblaze Processor ..................................... 32
4.2 FIFO communication on the CoMPSoC .............................. 33
4.3 Function calls ......................................................... 40
4.4 Execution time with and without optimizations ................... 44
5.1 Images used as input for measurements .............................. 48
5.2 Function calls .......................................................... 50
5.3 Cumulative execution time of functions .............................. 52
6.1 Execution time with different Worker allocations over three processing tiles .................................................... 55
6.2 Execution time of rasterization with different FIFO sizes on a 168 by 108 image .................................................... 56
6.3 Execution time with varying image tile width ....................... 58
6.4 Execution time with varying image tile height ..................... 59
6.5 Execution time rasterizing Gears with varying image tile size ... 60
6.6 Execution time rasterizing Gears with varying image tile size ... 61
6.7 Execution time rasterizing Gears with varying image tile area ... 62
6.8 Profiling functions with a varying tile area using Gears ........ 63
6.9 Profiling functions with a varying tile area using Highlight ... 64
6.10 Profiling functions with a varying tile area using Scube ........ 64
6.11 Profiling functions with a varying tile area using Surfgrid ... 65
6.12 Possible processing tile configurations .......................... 66
6.13 Sensible processing tile configurations ........................... 67
6.14 Execution time with different worker allocations over one processing tile .................................................... 69
6.15 Execution time with different worker allocations over two processing tiles .................................................... 70
6.16 Execution time with different worker allocations over three processing tiles .................................................... 71
6.17 Execution time with different worker allocations over one processing tile .................................................... 73
6.18 Execution time with different worker allocations over two processing tiles .................................................... 74
6.19 Execution time with different worker allocations over three processing tiles .................................................... 74
6.20 Comparing LET with BBOX on different configurations with one processing tile .................................................. 76
6.21 Comparing LET with BBOX on different configurations with two processing tiles ................................................. 76
6.22 Comparing LET with BBOX on different configurations with three processing tiles ............................................... 77

A.1 Pixels and vectors ................................................................. 85
A.2 A sphere build up from a different number of triangles .......... 86
A.3 The three stages of a general graphics pipeline .................... 87
A.4 Simple projection of an image in the application stage ............ 87
A.5 The pipeline of the geometry stage ....................................... 88
A.6 Projecting a scene seen from the viewing frustum ................. 91
A.7 Projected scene seen from the viewing frustum ..................... 91
A.8 Image after rasterization ...................................................... 92
A.9 Graphical representation of the OpenGL pipeline ................. 93

B.1 A polygon can be divided into triangles .................................. 97
B.2 Scanlines on a grid of Pixels ............................................... 98
B.3 Scanlines intersecting with triangle edgeds ............................ 99
B.4 Identification of pixels inside triangle ................................... 99
List of Tables

2.1 Performance of processors in DMIPS ................................. 16

4.1 Size of buffers ......................................................... 35
4.2 Cycles needed for floating point operations ...................... 41
4.3 Arithmetics on fixed point integers ............................... 42
4.4 Cycles needed for Integer operations ............................ 42
4.5 Cycles needed for Integer operations ............................ 43

5.1 Characteristics of input TKI-files ................................. 47

6.1 Notation for processing tile configuration ...................... 54
6.2 Image configuration for FIFO size measurements .............. 56
6.3 Image configuration for image tile width measurements ...... 58
6.4 Image configuration for image tile height measurements ..... 59
6.5 Image configuration for image tile height and width variation . 60
6.6 Fastest image tile sizes for Gears ............................... 61
6.7 Area size of image tiles ............................................. 62
6.8 Area size of image tiles for profiling ............................ 63
6.9 Processing tile configurations using three processing tiles .. 68
6.10 Image configurations using three processing tiles ............ 69
6.11 Image configurations using up to 4 Workers .................... 72

B.1 Vertex parameters .................................................. 98
In the many months I’ve been working on this thesis I could rely on the support of a lot of people. Of course there are my parents, who made it possible for me to start my study and supported me in continuing it, for better and worse. Then there were my colleagues at the Student Association, the ETV. Only guessing can tell us how many weeks I lost during my coffee break at the board room, like nobody can quantify how much frustration was relieved when I came back to my room, always a little bit later then planned.

Then there are the colleagues sitting next to me, staring to a computer screen, whether it be some kind of source code or some assembly code, some strange signals or a complaining compiler, we always could ask each other for help or for a good laugh. Although I doubt they have a general idea of what I’ve been doing, as I doubt I understand their work, they helped me answering a lot of questions.

And of course my girlfriend helped me. I do not know whether she was the greatest support for me, or a better guardian angel to all the people and things around me I might have broken if she wouldn’t have been prepared to listen to me complaining about lost triangles and malfunctioning platforms.

I got a lot of support from the staff. In the first place my daily supervisor, Anca, but also Radu, Ashkan and Andrew are to be thanked. Working on an experimental platform has a lot of drawbacks. One can never be sure if a problem is caused by a limitation of the platform, a bug, or simply by a stupid programming fault. Fortunately each time I got stuck at least one of them was willing to help me out. He, or she pointed out my programming mistakes, or explained the working of the platform yet another time. But mostly I was impressed when I found a real bug in the platform. Within a few days someone would come up to me and explain how I could solve this bug myself, or tell me they fixed the bug and I could just download the newest version and be happy. I guess that’s one of the bright sides of working with an experimental platform and having most of the developers working in the same hallway.

Finally there are some friends I was not able to spend as much time with as I was used to. I am sorry for that, but I am really glad for all the distraction you’ve been providing me with.

Thanks to you all!

Johan Splinter
Delft, The Netherlands
October 18, 2011
Introduction

Computer graphics are an important field in computer engineering and computer science among others. Computers are used to visualize scientific data, to design complex buildings and machines, or to play games with an impressive level of realism. Graphical elements on a computer interface are often defined by geometrical shapes such as squares, lines and triangles, sometimes described by complex mathematical formulas. These images can be two dimensional planes or three dimensional models. Most displays nowadays, whether they are classic CRTs, modern AMOLED displays, wide screen theater projection screens or small mobile displays, work with a raster of pixels. A very computation intensive job is the rasterization of these geometrical shapes into a pixel-based image, or bitmap, that can be displayed on screen.

Graphics on a computer are handled by a Graphical Pipeline. For three dimensional images this is a 3D graphics pipeline. The job of this Pipeline is the projection of a three dimensional model on a two dimensional area: the screen. This job is usually performed in three stages. The first stage of a 3D Graphics Pipeline is the application stage. This is the stage in which an application generates 3D models or scenes. These models are passed onto the next stage, the geometry stage. This stage converts the three dimensional scene to a two dimensional scene characterized by polygons or triangles. The last stage is the rasterizer stage. This stage converts the triangles into a grid of pixels. If one pixel intersects with two triangles, only the foremost triangle will be displayed on this pixel. This way only one color-value per pixel has to be stored, along with a depth map. This depth map is used to determine whether a pixel in the buffer needs to be overwritten. When a buffered pixel is behind a new pixel, both the color-value and the depth value will be overwritten. It is also possible to combine the color values of different triangles into one new color using weights or transparency factors. Therefore one large memory is needed to store all the color values and depth values of the entire image. Since this memory is used to store the entire image frame, it is called the framebuffer.

Compared to small memories, larger memories tend to be slow and expensive. To diminish the need for one large memory for rasterization a technique called tile-based rasterization has been proposed. This techniques divides an image, or frame, into smaller images, called tiles or image tiles. Instead of rasterizing the entire image at once, as with frame-based rasterization, only one image tile is rasterized at a time. The buffer needed for the rasterization of one tile is called the tile buffer. The memory used to store the tile buffer can be fast and relative cheap due to its limited size. When a tile is rasterized the result, i.e., the color value of each pixel, is written to the framebuffer. The framebuffer used for tile-based rasterization can be smaller than the framebuffer used with frame-based rasterization. This has a number of reasons:

- Using frame-based rasterization not only color values, but also a number of other values, like depth, are stored in the framebuffer. With tile-based rasterization
these values only are stored in the tile buffer. Only color values are stored in the framebuffer with tile-based rasterization.

- During the rasterization process a number of temporary values is written to the buffer. These values can be read, evaluated and when needed overwritten. This involves a large number of memory transactions. With tile-based rasterization all these transactions take place on the tile buffer. With frame-based rasterization on the other hand these transactions are performed in the framebuffer.

- With tile-based rasterization only final color values are written to the framebuffer. Therefore a lower precision for these values can be used. On the tile buffer a higher precision has to be used because the color values can be used for computations, e.g., when transparency is involved. With frame-based rasterization all color values need higher precision.

Because smaller memories can be both cheaper and faster tile-based rasterization reduces the total costs for rasterizing hardware. It is also important to note that smaller memories use less energy. This may not be a big advantage on high-end systems, but it is a welcomed feature on especially mobile devices.

Because of the high computational demand of rasterization, this stage is often performed by dedicated hardware. Modern personal computers are equipped with a processor designed to handle computer graphics, called the Graphical Processor Unit or GPU. A modern GPU has a complex proprietary architecture that consists of up to 3000 million (Nvidia GeForce GTX 560 Ti [4]) transistors. Recently the usage of traditional mobile phones has been declining in favor of smartphones. Smartphones are an example of mobile devices that are able to display complex computer generated graphics, including three dimensional models. Therefor many modern mobile devices are also equipped with a GPU. This GPU is relatively simple compared to the GPU of a personal computer and often combined with a general purpose Central Processing Unit (CPU) within one system, a so called System on Chip. An example is the Nvidia Tegra 2 SoC. This SoC has only 260 million transistors [28].

GPUs are optimized to perform Graphical functions. A recent development in PC’s is the increased usage of the GPU for non-graphical tasks. However, the GPU can only be used for a limited type of functions. The programming model is unusual and tied to graphics, while the programming environment is tightly constrained [15]. When an application on a mobile phone has low graphical requirements but high computational demands that can not efficiently be performed on the GPU, hence part of the processing power of the SoC would not be used.

To limit the energy usage of an embedded system, while increasing the computational power, multi-core solutions are used. A multi-core SoC contains multiple processors integrated in one chip. If a large number of processors is integrated a scalable form of communication is required. This communication can be performed using a Network on Chip. Compared to the use of an application specific processor such as a GPU this has a number of advantages:

- A platform consisting of several CPUs can be programmed with known paradigms and tools.
• From a programmer's point of view, this solution is more flexible than a system consisting of a combination of heterogeneous processors like a CPU and a GPU.

• A network of processors can be used for a large scale of applications other than rasterization.

This work makes a parallelization study to investigate the performance of rasterization on a MPSoC. With these aforementioned advantages and disadvantages in mind, we attempt to answer the following open questions:

1. How can a tile-based rasterizer make effective use of a homogeneous network of embedded processors with distributed memory?

2. What are the trade-offs in designing and configuring this rasterizer. What different options can be tuned to increase the performance of this rasterizer running on a multi-processor system on chip?

To answer these questions, we need the appropriate hardware and software, namely:

• A hardware platform that provides multiple processing cores along with the needed memory.

• A software implementation of a tile-based rasterizer that can effectively use multiple processing cores.

As hardware platform, we choose the Composable Multi-Processor System on Chip, CoMPSoC platform [13]. This is a platform that can be synthesized to run on an FPGA. It can be configured to contain different types of resources, including memory and processing tiles. While normally being called tiles, to distinguish between the aforementioned image tiles they are called processing tiles in this work. An example of an implementation of a CoMPSoC platform is visualized in Figure 1.1.

![Figure 1.1: Example of a CoMPSoC configuration](image)

The processing tiles visualized in Figure 1.1 contain:

• A Processor, namely a Microblaze core.
CHAPTER 1. INTRODUCTION

- Local memory, used to store the instructions, data private to the tile and data used for communication between tiles.

- A DMA block to provide access to the shared memory on a remote tile.

Next to a processor core and different memory blocks the processing tile can contain a DMA block to send data to a shared memory, or read data from a shared memory.

Each processing tile can run a number of processes, called tasks in CoMPSoC terminology. Communication between processes is mainly performed using FIFO buffers. These FIFOs can reside on one processing tile, or have a producing task on a different processing tile than the consuming task. To switch between different processes on a processing tile different mechanisms are available.

1.1 Contributions

This work is a parallelization study that aims to compare the relative increase in performance obtainable by increasing the number of processors used for rasterization and varying other parameters, like the size of a communication buffer or the size of an image tile. Because we use an experimental embedded platform this work does not aim to compare the performance of a tile-based rasterizer on the CoMPSoC platform to the performance of any commercial rasterizer. In this section we first present the contributions of our research. Before these contributions can be made some tools have to be developed. The second part of this section elaborates on the characteristics of these tools and how they are implemented.

We divide the rasterization process in two types of processes. Using the model of a hive with ants or bees, these processes are called “Worker” and “Queen”. Only one Queen process is needed. This process will decode instructions and send triangles and rasterization instructions to the other processes. The other processes are Workers. The rasterizer can use one or more Workers. Each Worker will rasterize a single image tile at a time. A software FIFO will be used to send data, e.g., triangles that have to be rasterized, from the Queen to the Workers.

To measure the performance we select a number of images and rasterize them with the tile-based rasterizer. The execution time needed for rasterizing is used as a metric for evaluations. Using this metric we evaluate the influence of:

- The number of processing tiles used for rasterization. The minimal number of tiles needed is one. The total memory available on the platform is limited by the capacity of the used FPGA. Adding processing tiles limits the available memory per tile. For this work the maximum number of processing tiles is limited to three. We evaluate whether adding processing tiles indeed decreases the execution time.

- The number of processes used. If the first processing tile runs a Queen process, while the other processing tiles run one Worker per tile, it is interesting to determine what process is the bottleneck of the system. When the workload of the Queen exceeds the workload of the Worker, the Queen will become the bottleneck. On the other hand, if the workload of the Worker gets to high the Queen will become
idle for a reasonable portion of the execution time. Using the aforementioned scheduling mechanism it is possible to have one Queen and one or more Workers on one processing tile. This means that the processor time is equally shared among the processes. If one Queen and one Worker reside on the same tile, each of them will use a share of the processor time. This is especially interesting if the Workers are the bottleneck of the rasterizing process. By increasing the number of Workers the workload per Worker can be decreased. It is intuitive to increase the number of processes per processing tile on all the processing tiles to two for better load balancing.

- The size of an image tile. Previous works showed that the size of an image tile influences the speed of the rasterizer [6]. We want to investigate the effect of different tile-sizes on our implementation.

- The size of the FIFO. The Queen dispatches processing data to the Workers over FIFOs. Each Worker has a dedicated FIFO. If one of the FIFOs is full, the Queen can no longer send to it. The Queen will block, while other Workers might be waiting for new instruction data in the FIFO. If this effect causes only one Worker process to be active at the same time, adding more processing tiles has little positive impact on the performance. Increasing the number of Workers per processing tile might even have a negative impact on the performance. Increasing the size of the FIFO will most likely have a positive influence on the performance of the rasterizer.

Before the research objectives discussed can be met, we need a hardware platform and a software platform to perform tile-based rasterization on a system with multiple processor cores. The starting point for the hardware is the CoMPSoC platform. This platform can be instantiated with different parameters. For this work we design a CoMPSoC-instance with the following characteristics:

- We need a framebuffer to save the rasterized image. For this we use a shared memory. All processing tiles need to be able to write data to this memory.

- To perform an in-depth study the platform needs a large number of processing tiles. There is however a limited amount of memory available. Thus trade-off between the available memory per processing tile and the number of processing tiles has to be made.

- The processing tiles need local memory to store the tile buffer. This memory should be sufficient to store the tile buffer of an image tile.

- The size of the FIFO is of interest for the speed of the rasterizer. We want to be able to change this size between different measurements.

As a starting point for the software we use the tile-based rasterizer presented by M.C.A. van der Weide in [31]. This rasterizer runs as a single process on a desktop PC and needs to be modified to run on the CoMPSoC platform. For this the following modifications are implemented:
• The tile-based rasterizer runs in a single process. As mentioned above the program is split into two parts. The first part is called Queen. This part of the program that decodes instructions into triangles. The second part of the program is called the Worker. The worker receives triangles from the Queen and rasterizes them. Multiple Worker processes can be instantiated. After decoding the instructions to a triangle the Queen decides to which of the Workers the triangle needs to be sent.

• The Queen sends triangles to the Workers using FIFOs. Because the processing tiles use local memory for most operations, they cannot share variables. Instead for each Worker a FIFO is created. The producer of all these FIFOs is the Queen process, while the Workers are consumers.

• The modified tile-based rasterizer runs on a desktop PC, but needs to run on an embedded platform using Microblaze processors. This limits the size of arrays, because the memory available to the Microblaze is smaller than the memory typical available on a desktop PC. Decoding the instructions, an important task of the Queen, is done on a low level using bytes instead of words. It is important to note difference in endianness between the Microblaze and the x86 architecture. Furthermore the memory on a x86-architecture is byte-aligned. This is taken in account when modifying the program for the word-aligned Microblaze architecture.

• The output result of each Worker is a rasterized image tile in the tile buffer. Multiple image tiles form one image. Therefore the tile buffers are merged into the framebuffer. This framebuffer should be stored in a memory that is available to all processing tiles, namely the shared memory. This memory is accessed through a DMA-controller. The Workers are modified to copy each pixel in the tile buffer in their local memory to the right location in the framebuffer on the shared memory.

• The number of Workers is limited. If we rasterize one image tile per processing tile, the size of the image is the product of the number of processing tiles and the image tile size. To get a larger degree of freedom in choosing parameters like image size, image tile size and number of Workers, the concept of sections is used. Multiple image tiles form one section. Multiple sections form the entire image. The Queen decodes instructions to triangles that are send to the Workers. The Queen has not enough memory to save these triangles. When all instructions are decoded and a section has been rasterized the Queen will start rasterization again by decoding the first instruction. Thus each instruction will be decoded once per section. If the number of sections is high this uses a significant amount of time and becomes the bottleneck of the rasterization process. From the Queens’ perspective it is attractive to limit the number of sections and thus maximize the size of an image tile.

1.2 Overview

The overview of this thesis is as follows. Because knowledge of the basic concepts of rasterization is important for the understanding of this work, Chapter 2 presents the most
1.2. OVERVIEW

important concepts of rasterization and the details involved with tile-based rasterization. This chapter continues with an overview of hardware solutions for rasterization. First trends in computer engineering are discussed in short. With this background different kind of processing units, like CPUs and GPUs, that can be used for rasterization are introduced. Chapter 2 concludes with previous work on the topic of tile-based rasterization on a multi-core system. The implementation of the rasterizer used for this research is dealt with in Chapter 3 and 4. Chapter 3 starts with explaining the work-flow of the original software based rasterizer. After this we present the modifications needed to parallelize this rasterizer. In Chapter 4 we first introduce the CoMPSoC platform. Next we present the modifications needed to run the parallel rasterizer on this platform. This chapter further discusses our optimizations to increase the performance of this processor on the CoMPSoC platform. Using this implementation some experiments are conducted. Chapter 5 explains the experimental setup used to perform these experiments. The results of the experiments are presented in Chapter 6. In this chapter we show how the execution time of the rasterizer changes when parameters like the size of the FIFOs, the size of an image tile or the configuration of the processing tiles are modified. These experiments give ground to the conclusions and suggestions for further research in Chapter 7.
This chapter starts with introducing the concepts of rasterization in general and tile-based rasterization in particular in Section 2.1. With this knowledge we will take a brief look at hardware that can be used for the rasterization process. More specific, different types of processors will be discussed in Section 2.2. Finally in Section 2.3 previous work related to tile-based rasterization on a Multi-Processor System on Chip will be discussed.

2.1 Background on Rasterization

Rasterization is the process in which geometrical shapes, or primitives, are converted to picture elements (pixels), or fragments. On personal computers rasterization is often performed using libraries, e.g., Direct3D (developed by Microsoft) or OpenGL (developed by the Khronos Group). Since the Direct3D library is only available for platforms developed by Microsoft, on other platforms, e.g., UNIX variants, OpenGL (OpenGL Graphics Library) is the most used 3D-library. This section introduces this library first. Next, a technique called “tile-based rasterization” is introduced. This technique is used in the rest of this thesis.

2.1.1 OpenGL and Rasterization

OpenGL is a computer library that enables drawing of 3D images. For this thesis it is assumed this library is used, unless stated otherwise. More details about OpenGL are given in Appendix A.

3D images can be created on several abstraction levels. The highest level describes objects like people, furniture, buildings, astronomical objects etc. On a somewhat lower level the objects are constructed from three dimensional geometric objects, like cubes, cylinders, spheres etc. In the OpenGL environment, these three dimensional geometrical objects consist of two dimensional geometrical objects, like squares and polygons. All these objects are constructed from triangles.

OpenGL can be used to create high level object from lower level objects. E.g., a number of cylinders can be put together to form the wheel of a car. Multiple cubes and boxes can represent the coachwork of a car. Four wheels and the coachwork together can form a car. The high level objects can be manipulated. They can be moved or animated, the color and lighting can be modified and the shape can be transformed. The OpenGL pipeline breaks the high level objects into smaller objects, like boxes and cylinders and finally triangles. Next the pipeline will handle several steps, like transformation, lighting and rasterization, to finally be able to present the objects on the screen.

Because screens and printers do not support triangles, but are manufactured to work with pixels instead, a translation has to be done to map the triangles to pixels. Since
pixels are sorted in a raster this process is called rasterization. Rasterization is a calculation intensive step, because several calculations have to be done for each pixel. A modern computer screen at Full HD resolution (1080p) has \(1080 \times 1920 = 2,073,600\) pixels, thus millions of computations are needed to rasterize a projection that entirely fills the screen. In some applications this should be done 60 times per second to achieve a frame rate high enough for smooth animations. More details about rasterization can be found in Appendix B.

To calculate and store the value of each pixel several **buffers** are used. The most important one is the RGB-buffer, which contains the color values of each pixel. Each color is a combination of the three primary colors Red, Green and Blue. These colors are called “channels” in computer graphics. Another important buffer is the depth buffer, or Z-buffer. Each pixel has a depth associated with it. If the value of a new pixel is calculated, its z-value is compared with the corresponding value in the Z-buffer. Only if the calculated pixel is in front of the pixel already stored in the buffer the buffer will be overwritten. Otherwise the calculated values will be discarded. This process is called “hidden surface removal” (see Appendix B.4).

Besides the RGB-buffer and the Z-buffer several other buffers are needed for rasterization, such as a Stencil Buffer (S-buffer) and a Culling Buffer (C-buffer). Although the only values visible to the end user are in the RGB-buffer, the other buffers are indispensable in the calculation of the RGB-values.

### 2.1.2 Tile-based rasterization

The size of the buffers mentioned in 2.1.1 depends on the size of the entire image. Thus rasterizing a large image requires a large memory. The CoMPSoC system does not provide a large memory local to the processor. Instead the platform uses small local memories, in the order of hundreds of kilobytes. The platform also provides a larger memory. This memory however is external to the processors. Using an external memory requires a lot of communication overhead. This will both increase energy usage and slow down the execution speed. To enable the rasterization of large images on processors that have small local memories, tile-based rasterization has been proposed.

Using this technique the screen is divided into smaller rectangular parts, called tiles or image tiles. These image tiles are rasterized one at a time, thus requiring a smaller memory compared to when the entire image is rendered at once. Once a tile is rasterized, its RGB-values can be written to a global RGB-buffer, the framebuffer. Once each tile is rasterized, this framebuffer as a whole will be written to the screen and displayed to the user. In Figure 2.1(a) a raster of image tiles is projected on the image of a triangle.

The local memory has to contain all the buffers mentioned above. The size of these buffers depends on the size of the image tile rasterized on that Microblaze. An image of, e.g., 10 by 10 pixels has 100 pixels. Each pixel has a RGB-value, thus the RGB-buffer has to be able to contain the value of 100 pixels. The other buffers, such as the A-buffer, S-buffer, C-buffer and Z-buffer, are only needed during the rasterization process. When the rasterization is done only the RGB-buffers are needed for representing the image. The other buffers can be disregarded. Thus all the buffers have to be allocated on the local memory for image tile-rasterization. The only buffer that has to be allocated
on the global memory is the framebuffer. Since multiple image tiles can be rasterized sequentially in the same local memory, the total amount of memory needed can be smaller using tile-based rasterization. Furthermore, the size of the required precision for the framebuffer differs from the tile buffer. The RGB-values in the RGB-buffer are only used for displaying the image. In general a color depth of 24 bits will suffice for this purpose. A color depth of 24 bits provides one byte (8 bits) or $2^8 = 256$ levels per channel, or a total of over 16 million colors. The RGB-value of the tile buffer on the other hand can be used for other computations. If transparency, or blending is supported the RGB-value of a pixel can be used to compute the new value of a pixel. Therefore the precision of this value should ideally be higher than the precision of the final color presented to the user. Often instead of integers this process uses floating point variables. If each channel has a precision of 32-bits (four bytes) for example, the RGB-buffer in the tile buffer will need $3 \times 4 = 12$ bytes per pixel instead of 3.

The value of the tile buffers is used to fill the framebuffer. Since all tile buffers together form the framebuffer, but the tiles have no overlap, each single pixel in the framebuffer belongs to one and only one tile buffer. Unfortunately no such statement can be made about triangles. A triangle can belong to one or multiple image tiles. A triangle can even belong to no image tile at all, if the triangle is not within the borders of the resulting image.

In Figure 2.1(a) a triangle is presented along with a grid. This grid represents 16 image tiles used during the rasterization of this triangle. The triangle covers several tiles, but not all of them. During the processing of the upper left image tile, for example, the triangle can be neglected without influence to the rasterized image. To save communication overhead and computation time on the rasterizing processor the triangle should not be dispatched to the process responsible for rasterization of the upper left image tile. Therefore the process responsible for dispatching the triangle data to the rasterizing process creates a tilemask for each triangle. This mask contains a one for each image tile that intersects with the triangle, and a zero for all other image tiles. In [7] two algorithms that can be used for the creation of the tilemask are presented. The first is called Bounding Box test (BBOX). This algorithm is fast but not very precise. The second algorithm is called Linear Edge function Test (LET). This function is preciser but also more time and resources consuming. In the remainder of this section we present both algorithms in more detail.

### 2.1.2.1 Bounding Box test

The Bounding Box test, or BBOX, is the first algorithm we discuss regarding intersect testing. Instead of calculating exactly which tile a triangle intersects, it draws a rectangular box in which the triangle fits exactly. Consider a triangle with corners $A$, $B$ and $C$. The coordinates of these corners are given by $x_A, y_A, x_B, y_B, x_C$ and $y_C$. The entire triangle now fits within a rectangle bounded by $bb_{x_{left}} = \text{MIN}(x_A, x_B, x_C)$ and $bb_{x_{right}} = \text{MAX}(x_A, x_B, x_C)$ horizontal, and $bb_{y_{bottom}} = \text{MIN}(y_A, y_B, y_C)$ and $bb_{y_{top}} = \text{MAX}(y_A, y_B, y_C)$ vertical.

Each image tile is a rectangle that can be characterized by four coordinates. We call them $tile_{x_{left}}, tile_{x_{right}}, tile_{y_{bottom}}$ and $tile_{y_{top}}$. If one of the inequalities in Equation 2.1 holds then the triangle intersects the image tile.
evaluates to true, we know that the bounding box of the triangle does not intersect with
the current image tile. In words: the bounding box is placed entirely left, right, above
or below the current image tile.

\[
\begin{align*}
bb_{x_{\text{right}}} &< tile_{x_{\text{left}}} \\
bb_{x_{\text{left}}} &> tile_{x_{\text{right}}} \\
bb_{y_{\text{top}}} &< tile_{y_{\text{bottom}}} \\
bb_{y_{\text{bottom}}} &> tile_{x_{\text{top}}}
\end{align*}
\] (2.1)

The result is visualized in Figure 2.1(b). The image tiles that intersect with the
bounding box are colored gray. It is obvious that some image tiles, e.g., the upper left in
this example, will intersect with the bounding box of a triangle, but not with the triangle
itself. This means the triangle is processed by more image tiles then needed, which leads
to communication overhead and computational overhead in the rasterizing process. To
circumvent this a more precise algorithm is designed, the Linear Edge function Test
(LET).

2.1.2.2 Linear Edge function Test

The LET-algorithm uses the edges of a triangle instead of the edges of a bounding box.
Assume again a triangle is represented by the coordinates \(x_A, y_A, x_B, y_B, x_C\) and \(y_C\). We
describe point \(B\) using the position of \(A\), thus \(x_B = x_A + dX\) and \(y_B = y_A + dY\). The
dge from \(A(x_A, y_A)\) to \(B(x_A + dX, y_A + dY)\) is defined in Equation 2.2 In Equation 2.3
we present the incremental form of this formula. This form is easier to compute once
the first value is known.

\[
E_{LAB}(x, y) = (x - x_A) \cdot dY - (y - y_A) \cdot dX 
\] (2.2)

\[
E_{LAB}(x + \delta x, y + \delta y) = E_{LAB}(x, y) + \delta x \cdot dY - \delta y dX 
\] (2.3)

The value of \(E_{LAB}(x, y)\) can be used to calculate whether a random point with
coordinates \((x, y)\) is to the left of line \(L_{AB}\) \((E_{LAB}(x, y) < 0)\), to the right of line \(L_{AB}\)
(\(E_{LAB}(x, y) > 0\)) or exactly on the line \(L_{AB}\) \(E_{LAB}(x, y) = 0\). To define “left” and “right” we assume a counter-clockwise oriented triangle. This assumption is common in rasterization: OpenGL differentiates between the front the back face of a triangle using this assumption.

Next we take a square with a center point at \((x_{cs}, y_{cs})\) and a total width of \(l\). Then, according to [7], the conditions in Equation 2.4 can be used to determine whether this square intersects with our triangle.

\[
\begin{align*}
E_{LAB}(x_{cs}, y_{cs}) & \leq \frac{l}{2} \cdot (|x_B - x_A| + |y_B - y_A|) \\
E_{LBC}(x_{cs}, y_{cs}) & \leq \frac{l}{2} \cdot (|x_C - x_B| + |y_C - y_B|) \\
E_{LCA}(x_{cs}, y_{cs}) & \leq \frac{l}{2} \cdot (|x_A - x_C| + |y_A - y_C|)
\end{align*}
\] (2.4)

To apply these formulas to a rectangular image tile with a width different from the height, the coordinates of the triangle can be normalized. Normalization of the triangle coordinates can also be used to normalize \(l\) in Equation 2.4 to one, thus easing calculations.

In Figure 2.1(c) the gray image tiles are the image tiles determined to contain the triangle using the LET algorithm.

2.1.2.3 Comparison of LET and BBOX

In [7] both algorithms are compared. While the BBOX-algorithm is by far the simplest, it has the disadvantage of having false positives. These false positives are image tiles that do not actually contain the triangle, but are marked to contain the triangle. The triangle will be sent to the rasterizer assigned to the image tile. The penalty caused by these false positives depends on the implementation of the rasterizer. In previous works the triangles are saved in a shared memory. Our implementation uses a FIFO over a Network on Chip to send triangles to the destination process. The speed penalty for communication might be higher compared to [7]. On the other hand, the LET-algorithm is more complex. While the tile-based rasterization can be dispatched to multiple cores, each responsible for the rasterization of another image tile, the sorting and tilemask creation is performed on one core. Depending on the speed of the processors and the workload on this core compared to the rasterizing cores, it might be acceptable to increase the number of false positives while decreasing the workload on the first core. Therefore both algorithms will be implemented and compared.

The previous sections discussed the concepts of tile-based rasterization. To implement these concepts we need a hardware platform. Therefore the next section will introduce some hardware platforms capable of performing rasterization.
2.2 Background on Platforms and Processors

Rasterization can be performed in both dedicated hardware (e.g., GPUs) and software running on general purpose hardware (e.g., CPU’s). Both techniques however require a certain amount of memory. This chapter introduces some hardware platforms able to perform rasterization. the distinction between dedicated Graphical Hardware and General Purpose Hardware is diminishing. Modern GPUs are capable of performing a more extensive range of computations, some of which used to be the domain of CPUs. This is enabled by the introduction of Programmable Steam Processors. Therefore some time will be spent on this type of processor.

2.2.1 Trends in Processors, Energy and Memory

The ever decreasing size of transistors has some consequences for different resources used in (mobile) computers. The steadily increasing speed of microprocessors opens up new possibilities for the usage of computers and mobile devices such as smartphones. More complex programs however require more memory. Fortunately the size and the speed of memory is also increasing, albeit not with the same rate. Both faster processors and larger memories however tend to use more energy. Although the heat dissipation is a challenge on high-end desktop PCs, the demand for high power is of a much bigger importance on a large scale, like servers or supercomputers, or small scale, like mobile devices. The need for high capacity batteries is increasing, but the technology to provide them has not been developed yet.

The speed of memory and processors and the capacity of batteries and memories show a more or less exponential growth. The speed of this growth however differs. While the size of memory and the speed of processors increase with 50% per year, the speed of the memory and the battery capacity grows a lot slower. It is therefore worthwhile to investigate techniques that will save energy consumption and memory, possibly at the expense of extra computations.

2.2.2 Processing Units

To investigate the results of previous work, and the decisions we took in this study, some knowledge on different types of processors is required. This section elaborates shortly on different popular processing units. All those processors can be used for rasterization. In Section 2.3 we discuss some implementations that make use of the processing units discussed in this section.

2.2.2.1 Graphical Processing Unit

A Graphical Processing Unit (GPU) is designed to implement a 3D graphics pipeline, e.g., the OpenGL pipeline as presented in Appendix A. In [22] the task of a GPU is explained as follows:

"The task of any 3D graphics system is to synthesize an image from a description of a scene–60 times per second for real-time graphics such as videogames. This scene contains the geometric primitives to be viewed as
well as descriptions of the lights illuminating the scene, the way that each object reflects light, and the viewers position and orientation.”

GPUs are developed do deliver high performance in scene rendering, of which rasterization is only a single step. While they offer high speed, they also consume a lot of power, and are responsible for a big part of the total power dissipation in a desktop PC [27]. Moreover, GPUs are not very flexible. Although programmable Shaders [22] enable GPUs to use their resources for a large grid of data-parallel floating point computations, GPUs are not optimized for regular control flow like branches. Moreover, GPUs use a single memory which is very large and not local to the so-called programmable unified processors. Thus a lot of power is consumed in write and read operations.

2.2.2.2 Intel x86 based Processors

Most desktop PC’s and some Computer Servers use chips based on Intel’s x86 architecture. They deliver the power needed for both complex computations and regular usage. Their versatility imposes some drawbacks. They have been backwards compatible since 1978 [9]. Their size has grown exponentially since. For mobile devices they have become too large, too expensive and consume too much power. Although Intel has done some effort to introduce new mobile solutions, e.g., the Atom processor and Moorestown architecture, Intel’s head of the Ultramobility division admitted in 2008 that “Intel’s Atom processor had not yet matched the battery-life characteristics of the rival ARM processor for devices the size of a phone” [32].

2.2.2.3 ARM Processors

Intel’s rival ARM is the current leader on the market of mobile processors. According to [21] about 98 percent of all mobile phones use at least one ARM-designed core on their motherboards. Instead of developing processors, ARM Holdings designs architectures which can be fabricated by other companies, like Samsung and Qualcomm. ARM based processors are smaller and more energy efficient compared to the x86 architecture. Also they are available as soft cores to be used on an FPGA board. FPGA development boards with an onboard ARM processor are also available.

ARM claims to address the widest range of performance points, from the smallest GPU in the world, through full multi-core scalability beyond 1080p to General Purpose computing on GPU (GPGPU) for visual computing and multi-purpose applications with Mali Graphics Hardware [17]. Measurements or reports about the working and performance of this platform, however, have not been found.

2.2.2.4 Microblaze

Xilinx, manufacturer of FPGA development boards, is the developer of the Microblaze. The Microblaze core is a 32-bit RISC Harvard architecture soft processor core with a rich instruction set optimized for embedded applications [19]. A soft processor core is a processor that does not exist as an ASIC, but is only available on FPGAs.

Processors based on the x86-architecture, ARM-architecture and Microblaze processors are general purpose processors, or CPUs. Table 2.1 presents their performance
measured using the Dhrystone benchmark. It can be concluded that the performance of the Microblaze-core, the processor used in this work, can be best compared to an ARM Cortex-M3, a processor designed for “low-cost platforms for a broad range of devices including microcontrollers”.

<table>
<thead>
<tr>
<th>Processor</th>
<th>DMIPS/MHz</th>
<th>Max. frequency</th>
<th>Max. DMIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microblaze</td>
<td>1.30</td>
<td>200 MHz</td>
<td>260</td>
</tr>
<tr>
<td>ARM Cortex-M3</td>
<td>1.25</td>
<td>120 MHz</td>
<td>150</td>
</tr>
<tr>
<td>Intel Atom N270</td>
<td>2.4</td>
<td>1.6 GHz</td>
<td>3846</td>
</tr>
<tr>
<td>Intel Core i7-990X</td>
<td>46.0</td>
<td>3.46 GHz</td>
<td>159000</td>
</tr>
</tbody>
</table>

Table 2.1: Performance of processors in DMIPS

2.2.3 Programmable Stream processors

Modern multimedia applications have, among others, two distinguishing features compared to most other applications. These applications have the following properties:

1. They have a high demand on computational resources (e.g. the CPU)
2. They have inherent data parallelism

This data parallelism is visible in the amount in which the same computation has to be performed on various data, without the result of these computations depending on one another (Single Instruction Multiple Data, or SIMD). To cope with this kind of computations the Stream Processor has been proposed. A Stream Processor is able to deal with streams and kernels. A stream is a data stream, which can be a simple scalar or a complex object. A kernel is a computation that has to be performed on a stream over and over again. Each kernel has one or more input streams and one or more output streams. Once the needed input is available, the kernel will consume the input and perform the required computations. After it is finished it will present the result as one or more output streams, and wait for new data at its input.

A multimedia application can be described as number of kernels, connected by streams. These kernels can do computations in parallel, but also be chained together, so that the output of one or more kernels behaves as the input for another kernel or number of kernels. This way a network of kernels can be created (see Figure 2.2).

Figure 2.2: A Process Network
2.3 Related work

Several previous studies have approached the energy consumption and flexibility of rasterization. This chapter discusses these approaches, which include tile-based rasterization, embedded rasterization and parallel rasterization.

2.3.1 Research on tile-based rasterization

The author of [8] proposes a formula to predict the expected savings of tile based rasterization compared to traditional frame based rasterization. The optimal size of a tile in this work is determined to be 32 by 32 bits. [5] researches the suitability of tile-based rasterization for mobile devices. While tile-based rasterization is already used on high performance devices, this work looks specifically to mobile applications. To enable a quantitative approach it introduces GraalBench, a set of 3D graphics workloads representative for contemporary and emerging mobile devices. The work shows that with a tile size of 32 by 32 pixels the total amount of external traffic to off-chip memory is reduced. A larger tile would further decrease the bandwidth, but at the cost of the need for an unrealistic amount of local memory. A remark is made, however, that the reduction in bandwidth compared to traditional rendering depends significantly on the workload. A two stage approach for dividing triangles among tiles (or bins) is proposed. Stage management algorithms are presented to reduce communication overhead.

To research the influence of several factors on the execution speed of rasterization the flexibility of software is used. Instead of designing dedicated hardware, most research is performed with a software implementation of a rasterizer.

2.3.2 Software implementations of rasterizers

Rasterization is the most memory and calculation intensive step in the OpenGL pipeline. While the preceding steps in the OpenGL pipeline are mainly based on triangles, the rasterization stage is concerned with pixels. In the average OpenGL scene the number of pixels will surpass the number of triangles by at least an order of magnitude. Thus, while each operation in the rasterization stage is simple, especially compared to some matrix-based operations in preceding stages, the number of operations involved causes the rasterization stage to be fairly computation intensive. Therefore it is often performed in hardware. Nevertheless several software rasterizers are available. Some of these will be discussed below.

2.3.2.1 Mesa

The most widespread and mature software rasterizer is the rasterizer of Mesa. Mesa is an Open Source implementation of the entire OpenGL pipeline. While it can use a graphical card for the final stages, including rasterization, it is also able to do software rasterization, e.g., on systems without graphical hardware. The Mesa implementation is widely used, especially among Open Source initiatives, since it is available under the GPL-compatible MIT License. The disadvantage of Mesa is, due to its maturity,
its complexity. The main advantage of using Mesa as a starting point for a tile-based parallel rasterizer would be the continuing development of Mesa.

### 2.3.2.2 Graal

In [11] a development framework for graphics accelerators is introduced. The framework, called Graal, aims to be an open system which offers a coherent development methodology based on an extensive library of SystemC RTL models of graphics pipeline components. Although the framework is developed at the TU Delft, most involved researchers are no longer available at the University. Besides, the system is highly dependent on SystemC, a C++ library used to run cycle accurate simulations of hardware. The drawback of this system is its speed compared to a plain C/C++ implementation. Furthermore, the codebase had virtually no maintenance since the last publication in 2004. The codebase of Graal provides a back-end and a front-end. The back end is based on Mesa and creates a TKI-file existing of low level OpenGL commands used to draw triangles to the framebuffer. The front-end uses these instructions to draw the final pixel based image.

### 2.3.2.3 Graal for ARMuLator

In [10] a software simulation of an ARM processor is used for the implementation of a graphics pipeline based on Graal. This simulation, ARMuLator, is augmented with a framework. The entire simulation is written in SystemC. SystemC is a set of C++ classes which enable event-driven hardware simulation. This is faster compared to VHDL and Verilog, and because it is a superset of C++, it is relatively easy to use parts of the codebase of existing software written in C or C++. Due to its nature, however, it is harder to synthesize than genuine hardware design languages like VHDL and Verilog. Moreover, compared to an FPGA implementation, running a simulation in SystemC simulation can be very time consuming. This makes it hard to perform a large number of experiments with different configurations.

### 2.3.2.4 Tile-based simulator

To overcome the drawbacks of the previously introduced software rasterizers, the author of [31] proposed his own implementation to simulate tile-based rasterization of the TKI-files provided by the back-end of Graal. Since the main purpose of [31] is the investigation of memory usage, the actual generation of an output image was not the main goal of the simulator, and could be turned off. Furthermore, the rasterizer does not support some features, such as textures, and is not optimized for speed but for the minimization of external memory usage only. Finally one of the research goals was the use of a buffer that forms a hybrid between tile-based en frame-based rasterization. This involved writing unfinished tiles to the framebuffer and reading them back again when more data was available. This technique proved to be only hypothetical advantageous, and complicated the rasterizer by introducing a special buffer. In spite of these disadvantages this software rasterizer was the most usable rasterizer due to its simplicity, and sufficed for the proof of concept of tile-based rasterization on an embedded MPSoC system. Therefore the codebase of this simulator is used as a starting point for the current research.
2.3.2.5 Mobile Graphics

Mobile devices have limited available power, but require high image quality because the screen is relatively close to the viewer’s eye. These are the starting assumptions of [2]. The work presents three key innovations which aim to increase image quality and/or lower memory bandwidth. These innovations are:

1. Reducing the number of samples for anti-aliasing
2. Texture compressing to reduce memory access for texturing
3. A Scanline based culling scheme that decreases the reads from Z-buffer.

The first two innovations are orthogonal to tile-based rasterization. Although they both improve the quality/memory access ratio, they can be used combined as well as apart. The third technique aims at reducing reads from the Z-buffer. One of the aims of tile-based rasterization is to keep a local Z-buffer, so the need for remote reads is highly reduced. Implementing the Scanline based culling scheme might decrease the improvements gained by tile-based rasterization and vice versa.

2.3.2.6 Parallel rasterization

Following the work in [22] GPU designers have incorporated many rasterization algorithms over the years. They all are claimed to exploit one crucial observation: each pixel can be treated independently. Some exotic machines had a processor for each pixel. Nowadays, a modern GPU can have up to 384 parallel CUDA cores [24]. These cores are multipurpose cores that can perform different steps in the OpenGL pipeline.

In [4] software rasterization on an Intel processor is described. The aim of this article is to move the rasterization stage from dedicated hardware (e.g. a GPU) to general purpose hardware (e.g. a CPU). The used processor, a P54C core, supports vector computations. Vector computations are a type of Single Instruction Multiple Data (SIMD), on which one computation is done on an entire one-dimensional array of scalars (a vector) instead of on a single scalar. Because this is not a multi-processor approach, communication of data between processing elements is not as big a drawback as with a multi-core or multi-processor computation. The Larrabee is designed to be a combination of a General Purpose Processing Unit and a Graphical Processing unit in one, a GPGPU, and would be released in 2010. However, Intel canceled the hardware release in the fall of 2009 because it was not hitting performance targets. Intel stated it would continue the development as a software platform instead.

2.4 Summary

This chapter presented the background information for the research presented in this thesis. We started with introducing the most important concepts of rasterization and tile-based rasterization. Because the concepts need to be implemented on a platform, we gave a short introduction in dedicated rasterization hardware and continued with general purpose processors. The last can be used to implement a rasterizer written in software.
This chapter finished with introducing related work concerning tile-based rasterization. With this knowledge we will introduce the implementation used for this study in the next chapters. Chapter 3 will start with a software rasterizer developed for use on a desktop PC. This rasterizer will be parallelized. In Chapter 4 the parallel rasterizer will be modified to run on the CoMPSoC platform.
The rasterizer designed for this work is based on the tile-based rasterizer created in [31]. Therefore the working principles of the original rasterizer are explained in Section 3.1. Before porting it to the CoMPSoC platform the initial rasterizer is modified to run multiple processes in parallel. The modifications required are explained in Section 3.2. The result is a parallel rasterizer that runs on a desktop PC. In Chapter 4 we will implement this parallel rasterizer on a Multi-Processor System on Chip.

### 3.1 Initial tile-based rasterizer

The software rasterizer introduced in Section 2.3.2.4 is used as a starting point for the parallel or multiprocess rasterizer. Therefore the functionality of this rasterizer will be discussed here. It is important to note this rasterizer was designed to implement the last stage of the OpenGL pipeline (see Appendix A) on general purpose hardware. The rasterizer can be executed on a x86 desktop PC running Linux and rasterizes an image in a tile-based fashion (See Chapter 2.1). In Section 3.1.2 we will discuss the functionality of this rasterizer. First the input of the rasterizer will be discussed in Section 3.1.1.

#### 3.1.1 Input of the initial rasterizer

The rasterizer needs two kinds of input to create a rasterized image. Those are configuration parameters and TKI-instructions. The configuration parameters are used to modify the way the image is rasterized. TKI-instructions are instructions used by the OpenGL pipeline to pass instructions from the geometry stage to the rasterizing stage and are a representation of the scene being rasterized. First we introduce some parameters for the configuration. Next we discuss the TKI-instructions.

##### 3.1.1.1 The configuration parameters

The configuration parameters define how the image is rasterized and what the output will look like. The most important parameters for this work define:

- The image size. The image resulting from the rasterization process has a size measured in pixels and defined by the variables resx and resy. Triangles outside the image defined by this size will not be rasterized, triangles partially outside this region will only partially be rasterized.

- The section size. Instead of rasterizing the entire image in one step the image can be divided in sections. A section can be regarded as an independent image. The TKI-instructions are traversed once per section and the resulting image will be
written to a separate file. These files can be stitched together to form the entire image. Each section has a width defined by $secsizex$ and $secsizey$. The height of the image, $resy$ is an integer multiple of the height of a section. Similarly, the width of the image, $resx$, is an integer multiple of the width of a section, $secsizex$.

- The image tile size. An image can be divided into sections. A section is subdivided into image tiles. Again, the width of a section, $secsizex$ is a multiple of the width of an image tile, $dimx$. Likewise $secsizey$ is a multiple of $dimy$. The difference between an image tile and a section is at this point somewhat vague. Each TKI-instruction will be decoded for all image tiles within one section at the same time. The resulting image will be written to the same file for all the tiles within one section. Note that it is possible to create an image with just one section, or a section with just one image tile. In an image with just one image tile the parameters $dimx$ will equal $secsizex$ and $resx$, while the values of $dimy$ and $secsizey$ will match $resy$.

- The zoom factor. The TKI-instructions used for rasterization are aimed at a certain image size. If the image size defined by the configuration parameters is lower than the image size of the TKI instructions the result will be a cropped version of the intended result. Instead the image can be scaled so that the entire scene can be displayed. It is also possible to enlarge the image if the intended resolution is below the image size of the configuration parameters. For this process the parameters $zoomx$ and $zoomy$ are used.

### 3.1.1.2 The TKI-instructions

TKI-instructions are instructions used by the OpenGL pipeline to pass instructions from the geometry stage to the rasterizing stage. A version of the Mesa library (see Chapter 2.3.2.1) is modified to save these instructions to a file instead of passing them on to the rasterization stage of Mesa. These saved instructions can be used by the rasterizer to create images. These instructions are used for the first time in [11] and called TKI-instructions in this work, based on comments found in the code base of [11]. They can include instructions for state changes (e.g., change of the color), buffer operations (e.g., to empty a buffer or draw a buffer to the screen) or instructions to draw a primitive (see Chapter 2.1) to the buffer. In Figure 3.1(a) a hexadecimal representation of these instructions is presented. The original rasterizer reads these instructions from a file.

### 3.1.2 Functionality of the tile-based rasterizer

The previous section explained the input of the rasterizer. This section discusses how this input is processed.

#### 3.1.2.1 Decoding the instructions

Rasterization has been explained as the process of transforming a triangle-based image into a pixel-based image. The input of the rasterizer however exists of TKI-instructions instead of triangles. Therefore we need a conversion from instructions into triangles.
We call this decoding the instructions, a step performed by the function `decode()`. The original rasterizer uses a file with instructions in a binary format as input. Image 3.1(a) already showed a hexadecimal presentation of this file. Before we can decode the instructions we need them in a structured format. The step of converting the binary file to instructions is explained first.

The first four bytes of the instruction file give the length of the first opcode, including its operands. Each next operand is preceded by its length. The length is presented as an integer in Little Endian format. Care has to be taken, since the Microblaze processor is a Big Endian architecture, thus the byte order should be reversed when the rasterizer is used on the Microblaze architecture. In Figure 3.1(b) the first step of decoding the TKI-file is presented graphically: the length of the opcodes and operands is marked by red rectangles. This first byte after the length represents the opcode of the instruction, the rest of the bytes until the next length represents the operands. All instructions are decoded into an opcode with an operand and saved in an array called `scenebuffer`.

The rasterizer is based on the OpenGL pipeline. The OpenGL pipeline is essentially a state machine. The state of this machine is of importance for the rasterization stage. Therefore each triangle that has to be rasterized is accommodated with a `state` variable. Based on the coordinates of the triangle a tilemask is generated. To link the triangle to its state and tilemask a new type is introduced, the `element`. Each `element` contains a pointer to:

![Hexadecimal presentation of the TKI-instructions](image1)

![A polygon can be divided into triangles](image2)
• A triangle. The building blocks of a three-dimensional image are triangles or polygons. Each (non-convex) polygon can be converted into a number of triangles, as visible in Figure 3.2. Therefore, without loss of generality, this work only concerns with triangles as the input of the rasterizer.

Each triangle consists of three points, for convenience called A, B and C. Each of these points has three coordinates; x, y and z. A fourth coordinate is used when homogeneous coordinates are used. This so called w-coordinate can be used for perspective correct interpolation, but is not supported for this work. The x- and y-coordinates correspond to a pixel on the screen. When multiple pixels occupy the same x- and y-coordinates, the z-value is used to determine which pixel has to be displayed.

Next to its coordinates, each point has a color, defined by its RGBA (Red, Green, Blue and Alpha) values. When textures are used, each point will also need the coordinates of this texture. This information is encapsulated by the s, t and q values of each point. More on this subject can be found in Appendix B. At this point, textures are not supported by the rasterizer.

• A tilemask. The tilemask has already been discussed in Chapter 2.1.2, but is summarized here. Each section is divided in multiple image tiles. The tiles are rectangular blocks of equal size, without overlap, that together form a section. Each triangle can intersect with zero, one or multiple tiles. The tilemask is an integer array with a number of items equal to the number of image tiles. A one in the array means the triangle intersects with the related tile. When the integer is a zero, the triangle is not displayed in the corresponding tile. This means the element holding the tilemask does not have to be dispatched to the Worker responsible for the rasterization of this tile.

• A state. Appendix A briefly presents the OpenGL pipeline as a state machine. State-information, saved in the variable state, includes color-information and whether certain features, such as culling or blending, are enabled or disabled. This information is not only relevant to the current element (or its triangle), but also to the next. Each decoded element has the same state as the previous element, unless the state is changed explicitly. Thus, the state of each new element is initialized with a copy of the current state. Instructions that modify the state of the OpenGL State Machine modify the current state.

Next to these pointers the element can contain instructions that are not related to triangles, such as operation performed on the buffer. The structure of an element is visualized in Figure 3.3. The task of the decode() function is to decode each instruction in the scenbuffer and convert them to a number of elements.

3.1.2.2 Element buffer

The elements decoded by decode() are stored in a buffer of limited size. When this buffer is full the elements in the buffer will be executed. Executing an element means
3.2 Parallel Tile-based Rasterizer

The tile-based software rasterizer mentioned above runs a single process. We split the functionality of this rasterizer in a way that makes it possible to run the program in parallel on multiple processors or processor cores. In this section we discuss how the original rasterizer is modified into a parallel tile-based rasterizer that can effectively use multiple processing elements.

To ease the discussion of the parallel rasterizer two entities are introduced: the Queen
and the Worker. As in a hive, there can be only one Queen. The Queen is the process which divides the workload. This workload exists of the discussed elements, states and triangles. Multiple Worker processes can be created. They process the data provided by the Queen to perform the actual rasterization. A simplification of the collaboration between the Queen and multiple Workers is visualized in Figure 3.4. The Worker and the Queen are introduced first in Section 3.2.1 and Section 3.2.2. Next the communication between the Queen and the Worker is discussed in Section 3.2.3. Finally the removal of the tile switch functionality is discussed briefly in Section 3.2.4.

3.2.1 Queen

The Queen is the process that performs relative simple operations, essentially triangle based. Then it dispatches more computation intensive tasks, mainly pixel based operations, to a number of Workers. While there is only one Queen, a multitude of Workers is able to perform operations on different parts of the image in parallel.

The Queen is responsible for reading the file with TKI-instructions, the TKI-file, and the configuration file. It decodes instructions from the TKI-file as mentioned before using the decode() function. These instructions are used to create elements. Multiple instructions together can form one element, but one instruction can influence multiple elements.

3.2.2 Worker

Each Worker is executed as a separate process. The Worker is a process that receives elements from the Queen. Its main tasks is the rasterization of triangles, using a modified version of the execute() function. Each Worker rasterizes one image tile. Therefore it rasterizes the part of the triangle that intersects with its image tile. Also it can receive instructions to clear the tile buffer. The clear operations are also communicated via elements.

3.2.3 Communication

The communication between the Queen and the Workers is done via various channels. The channels are discussed below. In Figure 3.5 the three data structures that are used for communication between the Queen and the Workers are visualized.
3.2. PARALLEL TILE-BASED RASTERIZER

3.2.3.1 Initialization

The parallel software rasterizer running on a Linux desktop PC is implemented using POSIX Thread or Pthread. Each Worker is initialized as a separate Pthread on the same physical machine as the Queen. Pthreads allow different forms of communication. One form is by passing a pointer during initialization of a Pthread. This form is used during the initialization of a Worker-thread. The pointer points to a data structure called argument. The argument contains:

- A pointer to the framebuffer of the current section
- A pointer to the FIFO used for communication between the Queen and the Worker
• The dimensions of the image tile
• The dimensions of the section
• The position of the image tile within the section

For each tile in a section a new Worker is initialized. The information contained by the argument is needed only once per image tile. When rasterization of the section is finished the Worker process will quit. For the next section new Worker processes will be initialized.

3.2.3.2 FIFO buffer

When the Worker is initialized it will wait for new elements, which it will receive through a First In First Out Buffer, or a FIFO. This FIFO is created using the C-heap \[23\] library. The buffer can hold a number of tokens. Both the size of the token and the maximum number of tokens are fixed in advance. An efficient number of tokens in the FIFO is determined later on, using some experimental investigation. As discussed before, an element points to a tilemask, a triangle and a state. The tilemask of course can be dropped, but the element itself, the state and the triangle are used by the Worker. Instead of using three different FIFOs to send these structures to the Workers, a new structure, the packed element is introduced.

Each Worker is the consumer of one FIFO. The Queen is on the producer side of all these FIFOs. Using the tilemask of each element, the Queen will only put a packed element into the FIFOs corresponding to the Workers the packed element is relevant to.

3.2.3.3 Packed Element

The packed element is a simple structure containing an element, a triangle and a state. Contrary to the element, the packed element does not contain pointers to the structures, but actual copies of them. This has the disadvantages of using more memory space. An advantage is that after the creation of a packed element, the original element can be modified or destroyed without consequences for the packed element. Another advantage is that the Worker processes do not need to have access to the local memory of the Queen when all the structures, as opposed to pointers to these structures, are send over the FIFO. In Figure 3.6 the packed element is visualized.

Figure 3.4 can be adapted using the packed element as communication structure. This is shown in Figure 3.7. Instead of using the packed element for communication between the Worker and the Queen we could have modified the element to contain structures instead of pointers to structures. The packed element has been designed to ease the introduction of more efficient communication schemes in the future. By keeping the state and triangle in two different structures a communication scheme might be devised that makes proper use of the fact that the state does not change for each triangle. This optimization has not been further investigated in this work and is subject to future work.
3.2. PARALLEL TILE-BASED RASTERIZER

3.2.3.4 Writing of result

When the Queen has decoded all instructions and send the last element to the appropriate Workers, it signals the Workers that their work is done. This is done by setting a special value to the type property of an element. When a Worker receives this value it will exit. The Queen can synchronize to this by joining a Worker. Joining a process means the process that initializes the join will lock until the joined process has finished. The Queen will join each of the Worker threads sequentially to make sure all Workers have finished rasterizing. When this is the case the Queen will write the framebuffer to a file and start with rasterizing the next section.

3.2.4 Removal of the tile switch

The variable called currenttitle is used to store the focus of the rasterizer. The function tileswitch() was used to calculate the next active tile. The parallel rasterizer however is not rasterizing a single image tile at a time, but has a number of image
tiles that are currently in focus. Since the number of image tiles in one section has to match the number of Workers, all image tiles will have focus concurrently. Therefore the functions and variables related to the image tile switch have been discarded. The element buffer, discussed in Section 3.1.2.2 has also been removed.

3.2.5 Summary

In this chapter we introduced the functionality of a tile-based rasterizer designed to run on a desktop PC. The input of this rasterizer was discussed as well as the most important functions and data structures. This rasterizer has been split into two types of separate processes. We call one of these processes Queen. We need only one instantiation of this process, that distributes elements to one or more instantiations of the second process: the Worker. The Worker process receives the elements and rasterizes them to an image. Since one Queen process and multiple Worker processes can run in parallel we called this program the parallel rasterizer. This parallel rasterizer is the starting point of the next chapter, where we will port it to CoMPSoC.
In Chapter 3 we designed a parallel rasterizer for a desktop PC running Linux. Because our goal is to evaluate the performance of a parallel rasterizer on an embedded Multi-Processor System on Chip we need to port this rasterizer to the CoMPSoC platform. In Section 4.1 we introduce this platform as a template that can be instantiated with different configurations. In Section 4.2 we discuss the requirements of the platform concerning this study. We discuss the trade-offs that had to be made and explain the decisions that were made. The limitations of this prototype will be discussed in Section 4.3. The flowchart of the parallel rasterizer on the CoMPSoC platform is presented in Section 4.4. Because the performance of the CoMPSoC platform is limited compared to a desktop PC, the execution time of the parallel tile-based rasterizer will be relatively long. We are interested in decreasing the execution time by using multiple processors and therefore focus on relative values. Running a large number of simulations on a slow platform requires a lot of time to run experiments however. Therefore some improvements to speedup rasterization are implemented. In Section 4.5 these instruction-level optimizations are discussed.

4.1 CoMPSoC platform

The hardware used for our study is the Composable Multi-Processor System on Chip, or CoMPSoC platform. It can be configured to consist of multiple different devices, called tiles. For this work the next tiles are used:

- Multiple processing tiles functioning as processing elements. These are presented in Section 4.1.1. Each processing tile executes an operating system called CompOSe, introduced in Section 4.1.2. This Operating System supports running multiple applications and tasks on one processing tile, as discussed in Section 4.1.3.

- A Microblaze processor functioning as a monitor. This monitor is explained in Section 4.1.4.

- A shared SDRAM memory, presented in Section 4.1.5.

These tiles are connected via the Æthereal Network on Chip (NoC) [13].

This section explains some concepts used on the CoMPSoC platform. First the processing tiles mentioned above are discussed.

4.1.1 Processing tile

The processing elements on the CoMPSoC platform are currently Microblaze processors. They are placed in a tile. These tiles consist of the Microblaze Processor itself, some memories and a DMA.
CHAPTER 4. PARALLEL TILE-BASED RASTERIZER ON COMPSOC

The processing tile has different types of memory, namely:

- Local memory including:
  - Instruction memory, containing the instructions.
  - Data memory, containing global variables.
  - Stack memory, for variables declared on the stack.
  - Heap memory, for dynamically allocated variables.

- Communication memory, including:
  - Memin memory, for incoming data.
  - Memout memory, for outgoing data.

A graphical presentation of a processing tile is shown in Figure 4.1. In this picture the communication memory is presented as two separate blocks, to emphasize the DMA is connected to the memout. This means data has to be copied to a region in the memout-memory before it can be sent over the DMA. Note that data read by the DMA will also be placed in the memout-memory.

Data on the stack is allocated on the usual way in C/C++. For allocating data on the heap the `os_malloc()` function is used instead of the usual `malloc()` function. This function is implemented in the operating system running on each tile: CompOSe.

4.1.2 CompOSe

Each processing tile on the CoMPSoC platform executes a lightweight operating system called CompOSe [14]. This operating system implements the important features of the platform including FIFO communication, running multiple tasks and monitoring the system. These features are discussed in the next sections.
4.1.3 Applications and Tasks

The CoMPSoc platform is designed to run multiple applications in a composable manner: the applications should not interfere with each other. In this work only one application will be created on the platform.

An application can run multiple tasks. These tasks can run on the same physical processor or on different processing tiles. Tasks within an application communicate with each other using FIFOs. FIFOs on the CoMPSoc platform are an implementation of the C-HEAP protocol \[23\]. This protocol uses administration variables to keep track of the number of elements in the FIFO. The content of the FIFO should also be stored in the available memory. It is possible to store the FIFO content on the shared memory, but this is at the cost of access time. Instead the data is stored at the consumer side. When the consumer and producer of FIFO-data reside on the same Microblaze, all data needed for the FIFO is stored in local memory. Otherwise the allocation of data is pictured in Figure 4.2. In this picture Tile 0 is the producer of data while Tile 1 represents the consumer. The arrows show the direction of the data flows between the processing tiles. The interconnect is implemented by the Network on Chip, or NoC.

Since the computational power of a processing tile is limited, load balancing is important. If a task has to wait for input from another tile it will be idle until this input is available. This may lead to inefficient use of the available processing power.

4.1.4 The monitor

The monitor is connected to the other Microblaze-processors via a bus, the Xilinx Fast Link (FSL), and to the PC connected to the FPGA board. This way the monitor is able to send data to the PC, which can print this data in a terminal. Several APIs exist to let
one of the Microblaze processors print to the screen of the connected pc. The function
mon_debug_info(int) prints any 32-bit value to the connected PC. Another important
function is mon_curr_global_time_info(). As the name suggests this function prints
the current global time, or the value of the clock counter, to the connected PC.

4.1.5 SDRAM memory
The shared memory implemented in the CoMPSoC platform is accessed via the DMA
block on the processing tiles. The function hw_dma_send() can be used to send a specified
number of bytes from the communication memory to a specified location of the shared
memory. To read from the shared SDRAM memory the function hw_dma_receive() can
be used. This function copies the specified number of bytes from the shared memory to
the communication memory of the processing tile. Since os_malloc can only be used to
allocate space on the local memory, allocation of data in the communication memory or
the shared memory has to be done manually.

4.2 CoMPSoC prototype for tile-based rasterization
In Section 4.1 we introduced the CoMPSoC platform. This platforms is a template that
can be initialized with a wide range of configurations. The instantiation of a specific
configuration on a specific FPGA-board is called a prototype. For this work we design
a prototype that has the required characteristics to perform our study. This section
discusses the choices made in designing this prototype.

4.2.1 Number of processing tiles
At least one processing tile is needed to run the parallel tile-based rasterizer. Since
the goal of this work is to determine the effectiveness of using multiple processing tiles,
one processing tile will not suffice. If one wants to use one task per processing tile,
an instantiation with two processing tiles is still too limited. Therefore the platform
prototype will be synthesized with three processing tiles. Adding more processing tiles
will decrease the memory available per processing tile and thus decrease the possibilities
to experiment with different image tile sizes.

4.2.2 Local memory on processing tiles
The Queen processes each instruction and element one by one, and reuses the space
available for storing these structures. Therefore the Queen needs very little memory on
the heap.

The Worker on the other hand uses the heap not only for storing the elements
received from the Queen, but more importantly for storing the tile that is currently
rasterized. The space needed on a processing tile thus depends on the number of Workers
on this processing tile and the size of an image tile. The memory needed by the Queen
has little influence on the total memory requirements. To store the image tile being
rasterized a large number of buffers is needed, together forming the tile buffer. The size
4.2. COMPSOC PROTOTYPE FOR TILE-BASED RASTERIZATION

<table>
<thead>
<tr>
<th>Buffer</th>
<th>Type used for storing</th>
<th>Size in bytes per Pixel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z-buffer</td>
<td>fixed</td>
<td>4</td>
</tr>
<tr>
<td>S-buffer</td>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>RGBA-buffer</td>
<td>color</td>
<td>16</td>
</tr>
<tr>
<td>Triangle</td>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>S-modified</td>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>S-valid</td>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>Z-modified</td>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>Z-valid</td>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>C-modified</td>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>C-valid</td>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>28</strong></td>
</tr>
</tbody>
</table>

Table 4.1: Size of buffers

of each of them depends on the number of pixels in an image tile and the data types used to store the data in this buffer.

The buffers are summarized in Table 4.1. There are two non-primitive types used in this table. The first, fixed, is discussed in subSection 4.5.2.2. The color is a data type used to represent the color of a value. It includes the Alpha-value. Since each value is encoded in a fixed-type, and a fixed-type needs 4 bytes to be stored, the color of one pixel can be stored in $4 \times 4 = 16$ bytes.

Each pixel in the tile buffer needs 28 bytes to be stored. The heap size of the processing tile is set to 0x38000h, or 229 376 bytes. Thus the number of pixels per tile should stay below $229376 / 28 = 8192$ pixels. E.g., an image of 180 by 120 pixels (a ratio of 3:2) consists of in total 21600 pixels. If these pixels are divided over three tiles each tile will store 7200 pixels, which is below the imposed limit. In practice this is too much for the tile with the Queen and one Worker. Therefore the resolution is reduced to 168 by 108 pixels.

The Stack size of the processing tiles is set to 0xF00h or 3840 bytes. This is small compared to the heap size but proves to be sufficient in practice. As will be explained in Section 4.3.2, the TKI-instructions of the image to be rasterized are linked with the executable files of the rasterizer and thus stored in the instruction memory. To be able to rasterize even simple images a reasonable amount of instruction memory is needed. The instruction memory size is set to 0x20000h, or 128KiB.

We set the size of the communication memories, memin and memout, to 0x2000h, or 8KiB. This memory is used by the FIFO. Part of the memout memory is reserved for use by the DMA block.

### 4.2.3 Size of shared memory

The shared memory is set to 0x10000h or 65 Kilobytes. This is enough for 16384 pixels. Assuming again a picture ratio of 3:2 this results in a maximum image of size 156 $\times$ 104. Note however that the image size also depends on e.g., the image tile size.
4.3 Limitations of the CoMPSoC prototype

The prototype introduced in Section 4.2 has been designed to run the parallel rasterizer introduced in Section 3.2. The CoMPSoC platform however has some limitations compared to the desktop Linux PC used to run the parallel rasterizer. The most important limitations are discussed in this section.

4.3.1 Memory size limits and their implications

The allocation of the available SDRAM memory on a processing tile is configurable. Due to the limited size of an FPGA the memory available on a processing tile however is limited. By increasing the number of synthesized processing tiles the memory per processing tile will decrease. By using the possibility of running multiple tasks at one processing tile the available memory per Worker process is limited even further. The size of a section is the product of the size of an image tile and the number of Workers.

The rasterizer is able to rasterize multiple sections in sequence. To rasterize large images sections can be used. The number of sections per image is not limited by an upper bound, thus the size of an image on the platform will only be limited by the size of the global memory.

In Chapter 3.1.2.1 we discussed the TKI-instructions which are stored in the scenebuffer. This scenebuffer causes a number of difficulties. In the first place, the length of the array is not known at compile time. The TKI-file should be traversed once before allocating memory for the array. The amount of memory needed for this operation is another limitation. Since the size of a TKI-file is not limited in any way, it is not possible to guarantee that the memory needed for the scenebuffer is available.

Finally the processing and decoding of the TKI-file to create the scenebuffer takes, depending on the number of instructions, a significant amount of time. This has to be done before the actual rasterizing can be done and it is hard to distribute this task over multiple processors. This means the Worker processes have to wait until the entire TKI-file is processed. To speedup the rasterizer, while diminishing the memory usage, instead the TKI-file itself is opened once and read again for each instruction. The disadvantage of this process is that the decoding of the binary code into opcodes and operands has to be repeated once for each section.

4.3.2 Absence of file system

The CoMPSoC platform and its operating system have no notion of a file system. This is not unusual for an embedded system. The original rasterizer however uses the file system for three purposes: reading the file with TKI-instructions, reading the configuration file and writing the rasterized image. This functionality has to be implemented in another way.

The TKI-file and configuration file can be supplied over a network or bus (e.g. USB) or could be included in the source code. For simplicity, the last option is chosen with a small modification. Instead of including both files in the Queen during compilation, the files are parsed into a C++ source file. This file is compiled into a single object. The
resulting file is linked with the executable during the linking phase. This way only one file has to be recompiled before another image can be rasterized.

The rasterizer uses a single framebuffer per section, a so called section buffer. Each Worker writes the rasterized image tile to the same section buffer. While this is easy on a desktop PC, it imposes several difficulties on the ComPSoC platform. First of all, the Workers can be configured to run on different processing tiles, and thus not have access to the same memory. Secondly, the final result of the rasterization process is not a number of image tiles, but the combination of these image tiles: the framebuffer. Therefore the concept of section buffer is removed. Instead a tile buffer and a framebuffer are introduced. The tile buffer is local to a Worker while the framebuffer has to reside on a shared medium, the shared SDRAM memory. The tile buffer is used during rasterization, and therefore has to include all the buffers mentioned in Chapter 2.1.1. The framebuffer on the other hand is only needed for writing and reading the RGB-buffers. When an image tile is fully rasterized it will be merged into the framebuffer on the right coordinates.

While the rasterized image can be written to the framebuffer on the shared memory, there is no easy way to get it out of there. FPGAs with a display or a display connection exist. This may create a possibility for displaying the image directly on the board in the future. For the moment however another solution is chosen. After all Workers have finished the rasterization of the last section the Queen writes the content of the memory to the monitor that displays it to the user’s console. The monitor of the ComPSoC platform is able to handle 32-bit integers. Because the memory of the Microblaze is word-aligned, thus also depends on 32-bit values, the RGB-buffer is replaced with a RGBA-buffer. Although the Alpha-channel is not actively used in current implementations, this makes it easy to represent a color value as an integer. A small script can read these integers and reconstruct the image for reviewing.

4.3.3 Limited output

One of the features available on regular desktop PCs is the possibility to print strings of arbitrary size to the terminal or write them to a file. Although this is less and less used for consumer applications, this is a very useful way for checking the functionality of the program. It is also used for debugging applications.

On the ComPSoC platform this kind of feedback is very limited. The CompOSe operating system currently only supports sending integers over the FSL-link to the monitor. The monitor is able to send them to the PC which is connected to the FPGA-board the platform is running on. Therefore a list has been made with integers and the message, be it an error, a warning or just information about the status of the program.

4.3.4 Word Addressable versus Byte Addressable

Some actions performed in the tile-based rasterizer move the value of a couple of bytes into one 32-bit variable. While this is possible on the PC, it depends on the assignment of the characters whether the Microblaze processor can handle this operation at once. Copying the first up to the fourth byte of a character array into a 32-bit variable is no problem, but choosing the second up to the sixth byte to form a new 32-bit variable is
not possible on the Microblaze. While the x86 simulation will do this as expected, the Microblaze will once again copy the first up to the fourth byte into the new variable. Therefore these operations will have to be performed byte-wise instead of word-wise.

4.3.5 Communication

The desktop version of the rasterizer uses three main communication channels, as represented in Figure 3.5. They are as follows:

1. During initialization an argument is send to the workers. This argument contains a pointer to data needed by the Worker to rasterize a tile. Because the Queen does not initialize the processes on the CoMPSoC platform this form of communication does not exist. Furthermore sending a pointer to the Workers does not suffice, since the Worker may run on another tile than the Queen and thus have no access to the local memory of the Queen. Instead the initializing values are packed into a special type of packed element called the start element. This element contains information about the tile and section currently rasterized.

2. By joining a Worker process the Queen will only continue when this Worker process, implemented by a Pthread, is exited. On the CoMPSoC platform this behavior is simulated by introducing an extra FIFO from each Worker to the Queen. When the Worker is finished rasterizing a tile and has merged the tile buffer with the framebuffer, it will send an integer over this FIFO. After the Queen has send the last element of a section to the Worker process, it will wait for this integer.

3. Via FIFOs elements are send to the Workers. Since the FIFO on the PC version and on the CoMPSoC platform are both implementations of C-Heap, little modifications are needed to port FIFO communication to the CoMPSoC platform.

4. The tile buffers are merged via a shared array (the section buffer or framebuffer). As mentioned above the Worker will use a special FIFO to communicate to the Queen that it is finished rasterizing the current tile. Before this action is performed the Worker will write the content of the local tile buffer to the shared framebuffer. Since this framebuffer resides on the shared SDRAM memory this can be performed without interference of the Queen. Instead the DMA block on the processing tile is used.

4.4 Implementation of the parallel rasterizer on the CoMPSoC

The limitations in the previous chapter give rise to a number of modifications in order to enable the tile-based rasterizer to run on the CoMPSoC platform. The most important functions of the parallel rasterizer modified for the CoMPSoC platform are presented in Figure 4.3. On the left the most important functions of the Queen process are depicted. The right block presents the flow in the Worker process. The two most computation intensive functions are:
• The function runframe(). This is the most computation intensive function on the Queen. It is called once for every section of the image. It reads each instruction one by one and converts it to elements. These elements are sent to the Worker processes. When the image tile size is fixed, the size of the section increases with the number of Workers, and the number of sections per image decreases. Since runframe() is run once per section the workload of the Queen decreases with an increasing number of Workers.

• The function process_triangle(). When the Worker is initialized it will receive elements from the Queen. Most of these elements will contain triangles that need to be rasterized. First a triangle is translated to the local coordinates of the image tile. Next the function process_triangle() is called for the current triangle. It encapsulates the most computation intensive functions performed by the Worker. The function convert_triangle() performs the pixel identification of the triangles (see Appendix B.2). When each pixel inside a triangle is identified the functions get_fragment() performs the parametrization of each pixel inside the triangle (see Appendix B.3).

4.5 Instruction-level optimizations

The Microblaze processor can be synthesized with varying features, such as the length of the pipeline. In essence however, it is a very simple processor with relatively low performance. An example is the low performance in dealing with floating point variables. Although the absolute speed is not very important for the research in this work, the design optimized for performance is chosen for several reasons, including the ability to do a high number of experiments in a reasonable time. Therefore optimizing the execution time of the rasterizer has some advantages. An improvement can be made by replacing the floating point variables with fixed point variables. Another improvement can be made by minimizing the number of divisions.

4.5.1 Divisions

One of the tasks of a processor is performing so called Arithmetic Operations. These involve addition, subtraction, multiplication and division. The speed of addition and subtraction are usually relatively high, both might be performed in one clock cycle. The speed of multiplication will be somewhat lower, because in most implementations one multiplication requires a number of additions, where the number of additions will depend on the bit width of the variables. The slowest arithmetic operation is usually the division. Table 4.2 and Table 4.4 show that the Microblaze is no exception to this rule.

A compiler might be able to improve the speed of a program in replacing divisions by multiplications with the reciprocal. E.g. \( \frac{a}{b} \) can be replaced by \( a \times \frac{1}{b} \). Of course this only improves performance when \( \frac{1}{b} \) can be computed at compile time (e.g., when \( b \) is constant) or when the number of divisions with \( b \) is high enough to compensate the time spent in computing the reciprocal of \( b \) at runtime.
After examination of the rasterizer all divisions that are done multiple times with the same divisor are replaced by once calculating the reciprocal, and next multiplication with the reciprocal. This way the optimizations no longer depend on the compiler. A drawback is the decreasing code readability.
4.5. INSTRUCTION-LEVEL OPTIMIZATIONS  

4.5.2 Fractional number representations

Several ways to represent fractional numbers binary exist, but the most used ones are the floating point representation and the fixed point representation. They are discussed in this section.

4.5.2.1 Floating points

Floating points are variables with a very large range. The used Microblaze has a bit-width of 32 bits. IEEE specifies the format of a floating point variable in standard IEEE 754-2008. A floating point of 32-bits is called a “single precision” floating point and consists of three parts: a sign (1 bit), an exponent (8 bits) and a significand precision (24 bits of which 23 actually stored).

The value of a floating point can be computed by multiplying the significand with two to the power exponent, or

$$value = -1^{sign} \times 1.\text{significand} \times 2^{(exponent-127)}$$ (4.1)

The advantage of this format is the large range of numbers that can be stored. A 32-bit signed integer can have no positive nonzero value smaller than 1, and none larger than $2^{31} - 1$ (0x7FFFFFFFh), thus the largest value is only $2^{31}$ times larger than the smallest nonzero value. With single precision floating points on the other hand, the smallest positive nonzero value is $2^{-126}$ while the largest value is $(2 - 2^{-23}) \times 2^{127}$. The largest floating point can thus be in the order of $2^{254}$ times larger than the smallest nonzero value. The precision over this range however changes with the size of the value.

While the advantage of floating points is clear, namely its large range, the presentation format has several disadvantages. The most important one in this case is its complexity. Arithmetics applied to floating point numbers involve splitting the number in its three parts and performing different computations on each part. E.g. for a division of a floating point number by another floating point the significand parts will have to be divided while the exponents should be subtracted. Although the Microblaze has a specialized Floating Point Unit (FPU) the arithmetical operations applied to floating points are very slow [18], as presented in Table 4.2,

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instruction</th>
<th>Clock Cycles Needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>fadd</td>
<td>4</td>
</tr>
<tr>
<td>Subtraction</td>
<td>frsub</td>
<td>4</td>
</tr>
<tr>
<td>Multiplication</td>
<td>fmul</td>
<td>4</td>
</tr>
<tr>
<td>Division</td>
<td>fdiv</td>
<td>28</td>
</tr>
<tr>
<td>Conversion to Signed Integer</td>
<td>fint</td>
<td>5</td>
</tr>
<tr>
<td>Conversion from Signed Integer</td>
<td>fitt</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 4.2: Cycles needed for floating point operations
4.5.2.2 Fixed Points

Another way to represent fractional numbers is the fixed point number representation. Fixed point numbers are scaled integers. The scaling factor is known before any manipulations to the value are performed, where the scaling factor of a floating point number is encoded in its binary representation. E.g. when the scaling factor is set to 100, the fractional number 134.2 can be presented as \(134.2 = 134 \times 100 = 13420\). Arithmetics on fixed points are relatively easy, because the scaling factors of two fixed point values are equal. Addition and subtraction can even be performed without knowing the scaling factor. If \(a_{\text{fixed}}\) is the fixed point representation of a fractional value \(a\), and the scaling factor is \(scale\), the operations in Table 4.3 are equivalent.

<table>
<thead>
<tr>
<th>Operation on fractional number</th>
<th>Operation on fixed point representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a = b + c)</td>
<td>(a_{\text{fixed}} = b_{\text{fixed}} + c_{\text{fixed}})</td>
</tr>
<tr>
<td>(a = b - c)</td>
<td>(a_{\text{fixed}} = b_{\text{fixed}} - c_{\text{fixed}})</td>
</tr>
<tr>
<td>(a = b \times c)</td>
<td>(a_{\text{fixed}} = \frac{b_{\text{fixed}} \times c_{\text{fixed}}}{scale})</td>
</tr>
<tr>
<td>(a = b / c)</td>
<td>(a_{\text{fixed}} = \frac{b_{\text{fixed}}}{c_{\text{fixed}} \times scale})</td>
</tr>
</tbody>
</table>

Table 4.3: Arithmetics on fixed point integers

Unfortunately most programming languages and processors have no direct support for fixed point representation. Therefore a small class is created that forms a wrapper around an integer. This class uses operator overloading to perform the most common tasks on fixed points. For some operations, like addition, subtraction and comparison with two fixed point values, the regular integer operations can be used. For other operations, like multiplication, division and conversion to or from other primitives a scaling is needed. By choosing the scaling factor carefully, that is by letting \(scale\) be a power of two, these scaling operations can be performed by logic bit shifts. Since bit shifts are fast operations the use of fixed point values should add minimal overhead compared to the use of integers, while it provides a large speedup compared to the use of floating points.

The speed of integer arithmetics is presented in Table 4.4.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instruction</th>
<th>Clock Cycles Needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>add</td>
<td>1</td>
</tr>
<tr>
<td>Subtraction</td>
<td>rsub</td>
<td>1</td>
</tr>
<tr>
<td>Multiplication</td>
<td>mul</td>
<td>1</td>
</tr>
<tr>
<td>Division</td>
<td>div</td>
<td>32</td>
</tr>
<tr>
<td>Barrel Shift Left</td>
<td>bsll</td>
<td>1</td>
</tr>
<tr>
<td>Barrel Shift Right</td>
<td>bsrl</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.4: Cycles needed for Integer operations

Because the Microblaze does not have a fixed point unit, the operations performed on fixed point values are a combination of several instructions. The speed of these combinations is presented in Table 4.5.
4.5. INSTRUCTION-LEVEL OPTIMIZATIONS

<table>
<thead>
<tr>
<th>Operation</th>
<th>combination of instructions</th>
<th>Clock Cycles Needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>add</td>
<td>1</td>
</tr>
<tr>
<td>Subtraction</td>
<td>rsub</td>
<td>1</td>
</tr>
<tr>
<td>Multiplication</td>
<td>mul, bsrl</td>
<td>1 + 1 = 2</td>
</tr>
<tr>
<td>Division</td>
<td>div, bsll</td>
<td>32 + 1 = 33</td>
</tr>
<tr>
<td>Conversion from Integer</td>
<td>bsll</td>
<td>1</td>
</tr>
<tr>
<td>Conversion to Integer</td>
<td>bsrl</td>
<td>1</td>
</tr>
<tr>
<td>Conversion from Float</td>
<td>fadd, fmull, fmint</td>
<td>4 + 4 + 5 = 13</td>
</tr>
<tr>
<td>Conversion to Float</td>
<td>fto, fddiv</td>
<td>4 + 28 = 32</td>
</tr>
</tbody>
</table>

Table 4.5: Cycles needed for Integer operations

Comparing Table 4.5 with Table 4.2 shows that only applying division and conversion on fixed points is slower than applying the same operations on their floating point equivalent. Therefore it is important to minimize these computations during runtime. In Section 4.5.1 the elimination of divisions was already discussed. Eliminating conversions to floating points is done by replacing all floating points to fixed points. Unfortunately, since the input instruction format (the TKI-file) uses floating points, a minimum number of runtime conversions is needed.

4.5.2.3 Choosing the scaling factor

The most important disadvantage of fixed points over using floating points is its limited range. Since fixed point values are created by a small wrapper around Integers, the largest representable value is only $2^{31}$ times bigger than the smallest representable nonzero value. The size of the largest value is dependent on the scaling factor. While the scaling factor can be of virtually any size, in practice there is only a small usable range. If the scaling factor is too high, the precision of the value diminishes, since the precision is over the entire range equal to the smallest nonzero value. By decreasing the scaling factor to a small size however the maximum representable value also decreases. Operations that lead to values that are larger than $2^{32} \times scale$ will cause undefined behavior. Unfortunately both very large and very small numbers are used in the rasterization process. The z-value given by the instruction file is in the order of $2^{16}$, while values smaller than $2^{-16}$ are used in the computations for fragment colors. Fortunately, the exact z-value of a triangle is not important, as long as the order does not change. By subtracting a value in the order of $2^{16}$ from the z-value before it is converted to a fixed point the value will stay within an acceptable range.

As shown in Table 4.4 multiplications are just as fast as bit shifts on the used Microblaze. It is however possible to synthesize a Microblaze processor that is optimized for area instead of speed. On this version the shift is indeed a faster operation than the multiplication. More important, the division on the Microblaze indeed is very slow. This is an important motivation to choose the scaling factor to be a power of two. Experimenting with different values for $scale$ shows that $2^{14}$, or 16384 is the best value for this purpose. Because the precision for fixed point values is constant over the entire range, while the precision for floating point values is proportional to the size of the value, the
results of computations with either representations will not always be exactly the same. Replacing floating point representation with fixed point representations does therefore alter the resulting image, but not in a way that is visible to the bare eye.

4.5.3 Results of optimizations

![Figure 4.4: Execution time with and without optimizations](image)

After implementing the mentioned optimizations the entire program is executed with a small image using a single processing tile. The cycle count is compared to the cycle count of the original. The replacement of floats by integers results in a large increase in cycle count, which roughly triples. This might be explained by the way the fixed points are implemented. Because the arithmetic operators are overloaded, instead of compiling the instruction to an opcode supported by the Microblaze, a subroutine call might be involved. Branching to a subroutine and returning from there takes 4 to 5 cycles. This explains the increase of the execution time. Fortunately the compiler supports optimization flags as parameters. With this flag turned on the overloaded operators are compiled inline, that is, no branches or subroutines are involved. The compiler supports 3 levels of optimization. The difference between level 1 and level 2 however is limited, while turning to the highest level the result of the rasterization process is compromised. Therefore the rest of the measurements are performed with level 2 optimization turned on.

For a fair comparison the original version of the program, using divisions and floating points, is also compiled with different optimization levels. Although the optimization level can be increased to 3 without compromising the result, the speed is lower than the
speed of the optimized version using fixed points. The results are visualized in Figure 4.4. Chapter 5 explains how these measurements are performed. The experiments performed in the rest of this work will use the optimized version of the rasterizer.

4.6 Summary

This chapter opened with an introduction to the CoMPSoC platform. An instance of this platform usable for parallel tile-based rasterization has been designed. The parallel tile-based rasterizer presented in Chapter 3 has been ported to this platform. To increase the performance a number of instruction level optimizations have been discussed. The resulting tile-based rasterizer running on CoMPSoC is used in the next chapters to perform a number of experiments.
In the previous chapters we introduced a parallel tile-based rasterizer. This rasterizer is used as a prototype implementation to examine the possible benefits of tile-based rasterization running on multiple processing elements. The main benefit of this implementations compared to the single-process implementation should be a decrease in execution time. In Chapter 6 we present the results of a large number of measurements to study whether this decrease in execution time can be realized. The experimental setup of these measurements is explained in this chapter.

In Section 5.1 four images are introduced. These images are used in the rest of this work to measure the execution time for rasterizing an image. Section 5.2 defines the way execution time of the rasterizer is measured. To understand how the execution time can be decreased it is useful to know in what code fragments of the applications the most time is spent. In Section 5.3 an investigation is made on what functions are responsible for a significant contribution to the execution time.

### 5.1 Input images

In Chapter 2.3 Graalbench is mentioned (see also [5]). This is a set of benchmarks used to perform measurements on rasterizers. The benchmark consists of a set of TKI-files. These instruction files are carefully chosen to represent a wide range of real-life applications. To rasterize the images on the parallel tile-based rasterizer they are converted to an object file and linked to the executable used on the CoMPSoC platform. Unfortunately the synthesized version of the CoMPSoC platform used in this work has insufficient instruction memory for any of the images used in Graalbench. Therefore a set of simpler images has been rasterized with the modified Mesa front-end to generate TKI-instructions. The images are visualized in Figure 5.1. Although, compared to Graalbench the characteristics of these images are less representative, they suffice for our performance evaluation. Some of their characteristics are presented in Table 5.1.

<table>
<thead>
<tr>
<th>Name</th>
<th>Size in bytes</th>
<th>Nr. of instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gears</td>
<td>38344</td>
<td>2321</td>
</tr>
<tr>
<td>Highlight</td>
<td>27281</td>
<td>1307</td>
</tr>
<tr>
<td>Surfgrid</td>
<td>10311</td>
<td>496</td>
</tr>
<tr>
<td>Scube</td>
<td>43039</td>
<td>2050</td>
</tr>
</tbody>
</table>

Table 5.1: Characteristics of input TKI-files
5.2 Profiling

Main-stream platforms often offer the ability to analyze a program during runtime. Profilers give a detailed report of all the functions in the program that are used, and how much time is spent running these functions. Using this report a software developer can spend time optimizing the part of the program that takes the most time to run. These profilers use samples on a regular basis to determine what function is currently running [29]. Unfortunately no such profilers exist for the CoMPSoC platform. Instead we use the internal timers of the platform.

In Chapter 4 the function called mon_curr_global_time_info() is mentioned. This function prints the current internal time, the value of the clock timer, to the PC connected to the FPGA. By printing the clock timer at the beginning of a function and at the end of a function the time spent inside the function can be calculated.

One big difference with sample based profiling has to be made. When one function calls another function, the timer will not be stopped. The times measured are therefore cumulative: if a timer is started on invocation of function b() by function a(), and function b() will call function c() before returning to function a(), the timer is only stopped when function b() returns. Thus the time measured for function b() will include the time spent in function c().

This technique has a drawback on measuring small functions. The function to print the internal timer value takes a few cycles to be executed, causing the accuracy of the measurements to be too limited for small functions.
5.2.1 Timers

Two small macros, called \texttt{STARTTIME} and \texttt{STOPTIME}, are created for time measurements. They both send a specified integer to the monitor, followed by a request to the monitor to print the current timer value. These values are registered as \textit{starttime} or \textit{stoptime}. By subtracting the \textit{starttime} from the following \textit{stoptime}, the time in between, called \textit{runtime}, can be calculated. By adding all the values calculated this way, the entire time spent between different calls of \texttt{STARTTIME} and \texttt{STOPTIME} can be calculated.

5.2.2 Timer Overflow

The global time printed by \texttt{mon\_curr\_global\_time\_info()} is stored in a 32-bit value. This means it can represent $2^{32} = 4294967296$ different values. On a system running at 100MHz 4294967296 cycles can be run in a little bit under 43 seconds. For programs running longer an overflow occurs. This means the value following $2^{32} - 1$, or 0xFFFFFFFFh, will be 0. This makes it harder to measure long execution times.

Within programs that are running a short time a small trick can be performed. If one is sure the program does not need more than $2^{32}$ cycles, it is easy to add $2^{32}$ to negative values of \textit{runtimes}. This will suffice for profiling most of the functions in the rasterizer. When the runtime is expected to be larger then $2^{32}$ however it is not safe to perform this trick, since a multiple of 32 might have to be added to \textit{runtime} to get the right result. Therefore a new macro is introduced, called \texttt{SHOWTIME}. This macro prints the current timer value. We call this value \textit{showtime}. If this macro is called often enough any overflow can easily be detected. Each time two consecutive timer values, be it \textit{showtime}, \textit{starttime} or \textit{stoptime}, can be compared. If the first represents a higher value then the second, one overflow has occurred. If $n$ overflows have been detected, the values of $n \times 2^{32}$ can be added to every next timer.

The moment to call \texttt{SHOWTIME} has to be chosen with care. If the macro is called too often the program will be slowed down. However, by not calling the function often enough an overflow might be missed. It is determined through experimentation that by calling \texttt{SHOWTIME} right after the invocation of the \texttt{decode()} function reliable measurements can be made.

5.3 Profiled functions

In the next chapter we present the results of measurements on the parallel tile-based rasterizer. These measurements are performed using the technique described above. When the execution time of a particular function is performed \texttt{STARTTIME} is issued in the beginning of the function, while \texttt{STOPTIME} is issued in the end of the function, before returning. \texttt{SHOWTIME} is called in the beginning of the \texttt{decode()} function.

5.3.1 Function hierarchy

Figure [4.3] in Chapter [4] shows the hierarchy in which the most important functions are called. For convenience this figure is repeated in Figure [5.2]. Since the Queen and the Workers are initialized as separate processes, they are presented in two separate blocks,
which can only communicate via FIFOs. The Queen process starts with the function called \texttt{runframe()}. Within this function \texttt{decode()} is the first important function to be called. This function results in an element that might be sent to the FIFO of one or more workers. Note that this diagram is a simplification: not every call to \texttt{decode()} will be followed by writing an element to \( n \) FIFOs. \texttt{decode()} and \texttt{write_fifo()} will be called as long as there are still elements to be rasterized for the frame. When all elements are processed the Queen will wait until all workers are finished rasterizing the current
image tile. Remember that one Worker will rasterize one image tile, and that all Workers together rasterize one section. Thus when all the workers are finished rasterizing a tile, a section will be finished. That means \texttt{runframe()} is finished. The Worker will check whether this was the last section. If this is the case the Queen is finished. Otherwise \texttt{runframe()} will be invoked again.

Meanwhile the Worker is also initialized. It starts with reading the FIFO connected to the Queen. The first \texttt{element} received from the Queen gives initialization data. After this data is processed the Worker reads the FIFO again. The next \texttt{elements} received from the Queen contain triangles to be rasterized. These triangles first have to be translated to the coordinates of the current tile, this is done in \texttt{translate()}. Then \texttt{process_triangle()} is called. Within this function \texttt{convert_triangle()} and \texttt{get_fragment()} are called. The last is called until all fragments (pixels) are calculated. When this is done the Worker returns to \texttt{read_fifo()}.

A special case occurs when the element received by the FIFO is a \texttt{finishelement}. In that case the rasterized tile will be written to the shared memory. This is done with the function \texttt{merge_into()}. When the tile buffer is merged into the framebuffer the Queen will be signaled the Worker is done with this tile, and the Worker waits for a new element.

### 5.3.2 Initial profiling

Before measurements on different parts of the parallel tile-based rasterizer are performed it is important to decide what functions have to be measured. A simple image is rasterized at a resolution of 10 by 10 pixels to measure the execution time of all functions shown in Figure 5.2. The result is graphically presented in Figure 5.3. Some conclusions can be drawn from this initial profiling.

This measurement is done on a single tile. One Queen process is initialized on this core. Measurements are performed with both one and two Worker processes on the same tile. In the left of the chart functions of the Queen process are presented: \texttt{queen()}, \texttt{runframe()}, \texttt{decode()} and \texttt{write_fifo()}. The right part of the chart reflects the Worker with the functions \texttt{read_fifo()}, \texttt{translate()}, \texttt{merge_into()}, \texttt{process_triangle()}, \texttt{convert_triangle()} and \texttt{get_fragment()}.

The time spent in Queen is measured from the start of the program until the last section is rasterized. Since this value is almost equal to the total time spent in \texttt{runframe()}, measuring the execution time of \texttt{runframe()} in the future is sufficient to determine the total execution time. Furthermore a large amount of the time spent on the Queen is spent in \texttt{decode()}. Since this is the most important task of the Queen, this is not a big surprise.

It is important to realize that almost no time is spent in \texttt{write_fifo()}. The time spent in this function includes the time the Queen has to wait before it can put a new \texttt{element} in the buffer because the buffer is still full. Since the FIFO in this example can hold only one token, this means the Queen is the bottleneck of this execution. This assumption is reinforced by the time spent in \texttt{read_fifo()}. The time spent in this function is not only the time needed to actually receive the \texttt{elements}, but includes the time the Worker is waiting for an input as well. The fact that this is the function that
takes the majority of the time on the Worker suggests that most of the time no token is present in the FIFO.

Although some time is spent in `merge_into()`, most of the remaining time on the Worker is spent in `process_triangle()`. Therefore the next functions are chosen for profiling:

- `runframe()`, which represents the time spent in rasterizing all the sections, measured on the Queen.

- `write_fifo()`, which represents the communication time and the time the Queen process is blocked because the FIFO is full, measured on the Queen.

- `read_fifo()`, which represents the communication time and the time the Worker process is clocked because the FIFO is empty, measured on the Workers.

- `process_triangle()`, which represents the time spent in rasterizing the triangles, measured on the Workers.

![Cumulative execution time of functions](image)

**Figure 5.3: Cumulative execution time of functions**

### 5.3.3 Summary

In this section we profiled the most important functions in the rasterizer. We found out that the time spent in `runframe()` is representative for the total execution time of the Queen. The time spent in `write_fifo()` is for a large fraction caused by stalls when the Worker(s) is the bottleneck, like the time spent in `read_fifo()` is for a large fraction caused by stalls when the Queen is the bottleneck. The most time consuming function of the Queen is `decode()` and `process_triangle()` is the most time consuming function of the Worker process.
The previous chapters describe the rasterization process and the need for a parallel tile-based solution together with its implementation. The implementation of the parallel tile-based rasterizer has been discussed and the method to measure performance is introduced. In this chapter we present the results of our performance investigations.

6.1 Goals

The initial goal of this work is to determine the possibility to speedup tile-based rasterization using multiple cores. A number of parameters however has to be fixed to determine the possibilities to achieve this goal. The degrees of freedom which have to be investigated cover:

- The size of the FIFOs used to send elements to the Workers (Section 6.3).
- The size of an image tile (Section 6.4).
- The number of processing tiles used and the number of Workers per processing tile (Section 6.5 and Section 6.6).
- The algorithm used to create the tilemask (Section 6.7).

It is very hard, if not impossible, to determine the optimal value of all these parameters simultaneously. During each investigation we vary one or two parameters while the other parameters stay fixed.

Before performing the experiments some initial insight in the process is needed. Therefore an initial profiling is performed. In Section 6.2 we present the results of this first experiments.

6.2 Initial profiling on a parallel system

A number of processing tile configurations is chosen to perform the initial experiments. The results give some insight in the rasterizer, needed to perform the measurements presented in the next sections. A processing tile configuration defines the way multiple Workers and one Queen are divided over the processing tiles. A notation for expressing processing tile configurations is introduced. This notation is discussed in the first part of this section. Next a number of functions is measured using different configurations. The result is evaluated.
6.2.1 Notation of processing tile configurations

A special notation is devised to express the allocation of processes on processing tiles, the processing tile configuration. The first symbol of this notation is always a Q. This is done to denote the Queen runs on the first processing tile. Since the image tiles on the synthesized platform are homogeneous it does not matter on which processing tile the Queen is initialized. The Q is followed by a number of W’s equal to the number of Workers on the same processing tile as the Queen. The other processing tiles runs zero, one or more Workers, expressed by zero, one or more W’s. The processing tiles are separated by a dash. To clarify this notation Table 6.1 presents the configurations used in the first measurement. For this work the number of Workers on one processing tile is limited to 2. Each processing tile can thus contain 0, 1 or 2 Workers. Every configuration should have at least one Worker, and since the processing tiles are homogeneous Q-W- (one Worker on the second processing tile) is the same platform as Q--W (one Worker on the third processing tile).

<table>
<thead>
<tr>
<th>Workers on tile 1</th>
<th>Workers on tile 2</th>
<th>Workers on tile 3</th>
<th>Notation</th>
<th>Workers in total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>QW--</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>QWW--</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Q-W-</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
<td>Q-WW-</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>QWW-W-</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>QWW-WW-</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Q-W-W</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>QW-W-W</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>QWW-WW-WW-</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 6.1: Notation for processing tile configuration

6.2.2 Profiling on different processing tile configurations

Using the processing tile configurations presented in Table 6.1 we rasterize the Gears-image on a resolution of $104 \times 78$ pixels. The FIFOs from the Queen to the Workers can contain only one element. With this configuration we evaluate the part of the execution time spent in the functions `runframe()`, `decode()` and `write_fifo()` on the Queen and the functions `read_fifo()` and `process_triangle()` on the Worker(s). The result is presented in Figure 6.1. The size of a section in this measurement is $104 \times 39$ pixels. This way one section fits in the memory of a single Microblaze. Because the list of configurations used in this measurements has configurations with 1 up to 5 Workers, a section has to be dividable by one ($104 \times 39$), two ($54 \times 39$), three ($104 \times 13$), four ($26 \times 39$) and five image tiles. Because both 104 and 39 can not be divided by five, the picture for the last measurement is scaled to $104 \times 80$ pixels. Since this resolution is not dividable by three we decided to use two resolutions that differ only 2.5% in area.

In Figure 6.1 we can see that on the QW-WW-WW configuration the total execution time of `read_fifo()` is multiple times larger then the total execution time of
runframe(). This can be explained by the fact that this configuration uses five Workers, and the execution times of five Workers are added in this graph. Representing the execution time for these function for each individual Workers would make the graph hard to read. It is therefore very hard to compare the values on a quantitative base. On a qualitative base however some conclusions can be drawn:

- In all the processing tile configurations decode() is only a small part of the execution time, while write_fifo() consumes a large portion of the total execution time on the Queen side. This implies that the bottleneck of the rasterizer is in the Worker. This suggests adding more processing tiles and/or Workers will further decrease the processing time.

- The function read_fifo() takes the majority of time in some processing tile configurations. This means that Workers are waiting for new elements, but the FIFO is empty. This may be due to data locality: if two consecutive elements for Worker 1 are decoded, the first will be in the process_triangle() function on Worker 1. The second will remain in the FIFO. This means that the Queen will have to wait until Worker 1 is done with the first element before it can write another element in the FIFO to Worker 1. In the meantime all other Workers might be idle.

- As the number of Workers grows the time spent in read_fifo() increases.

- Configurations with an equal number of processes on each processing tile perform well compared to other configurations with the same number of processing tiles.

The results presented above suggest the performance of the rasterizer is limited by
the capacity of the FIFO. By increasing the FIFO it might be possible to speedup the rasterization process.

### 6.3 Investigating the size of the FIFO

Because previous measurements suggested increasing the size of the FIFO makes the Workers more productive, the next measurements are conducted with different FIFO sizes. This time all the images mentioned in Chapter 5.1 are rasterized. From Figure 6.1 we conclude that processing tile configuration QW-W-W is one of the fastest configurations. We add the QWW-WW-WW configuration for this test to see whether the speedup with different configurations is comparable. The image configurations for this measurement is presented in Table 6.2.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>sections</th>
<th>resx</th>
<th>resy</th>
<th>secsizex</th>
<th>secsizey</th>
<th>dimx</th>
<th>dimy</th>
</tr>
</thead>
<tbody>
<tr>
<td>QW-W-W</td>
<td>1</td>
<td>168</td>
<td>108</td>
<td>168</td>
<td>108</td>
<td>56</td>
<td>108</td>
</tr>
<tr>
<td>QWW-WW-WW</td>
<td>1</td>
<td>168</td>
<td>108</td>
<td>168</td>
<td>108</td>
<td>56</td>
<td>54</td>
</tr>
</tbody>
</table>

Table 6.2: Image configuration for FIFO size measurements

The result is presented in Figure 6.2. The first thing that we noticed is that increasing the FIFO sizes will in all cases improve the execution time. When we look at the time needed to rasterize our four images in two configurations, we can see that by increasing the size of the FIFO from one to two elements, the total execution time is decreased.
by 4.7%. Increasing the size to three elements will increase another 2.7%, and the total decrease will be 7.3%. Gradually increasing the size to nine elements show that, though the decrease in execution time is less impressive with each increment, the total execution time decreases by 14.8%.

This decrease in execution time however is different for different images and different processing tile configurations. The biggest improvement can be seen in the rasterization time for Scube on configuration QWW-WW-WW, which decreases with 46%. The least impressive improvement is measured with Surfgrid on the QW-W-W configuration, which shows a decrease in execution time of only 3%. On average the execution time for the QW-W-W configuration decreases with 13.5%, while the decrease of execution time on the QWW-WW-WW is more impressive with an average of 23.7%. It looks like the effect of a larger buffer saturates on the long run. Although saturation is likely to occur long after a FIFO size of nine elements, due to the limitations presented in Chapter 4.3, a size of nine elements is chosen for further measurements. A larger FIFO size does not fit in the memory on some processing tile configurations.

Another observation that can be made from the current measurements is the relative speed of the processing tile configuration with six Workers (QWW-WW-WW) compared to the version with only three Workers (QW-W-W). Before attributing the increase in speed entirely to the increase in the number of Workers another observation has to be made: in Table 6.2 we see that the size of the image tiles differs between the two processing tile configurations. Therefore the influence of the image tile size on the execution time has to be investigated.

6.4 Size of the image tile

In previous measurements it was assumed that an image tile should be as large as possible in the limited amount of memory. This assumption has however not been verified. This is done in the next measurements. The chosen processing tile configuration is once again QW-W-W, the image configuration with one Worker per processing tile.

6.4.1 Variation in width of the image tile

The four images are rasterized at a resolution of $168 \times 108$ pixels. The section height and the image tile height are kept constant. The image configuration is presented in Table 6.3.

The results of these measurements are visible in Figure 6.3. Although this figure confirms that the size of an image tile should not be too small, it contradicts the assumption that a bigger image tile size will always speedup the process. Instead, there seems to be an optimum which is comparable though not equal for each image. For Gears the minimal execution time is measured at a tile image size of $42 \times 36$ pixels. Decreasing the image tile width to 14 pixels increases the execution time with 71%. Increasing the image tile width to 168 pixels increases the execution time with over 100%. With small variations the increase in execution time is less impressive. Decreasing the image tile width with 33.3% to 28 pixels increases the execution time with 11.8%. If the width of a tile is increased with 33.3% to 56 pixels, the increase in execution time is only 1.7%. 
CHAPTER 6. RESULTS

<table>
<thead>
<tr>
<th>Number of sections</th>
<th>resx</th>
<th>resy</th>
<th>secsizex</th>
<th>secsizey</th>
<th>dimx</th>
<th>dimy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>168</td>
<td>108</td>
<td>168</td>
<td>108</td>
<td>168</td>
<td>36</td>
</tr>
<tr>
<td>2</td>
<td>168</td>
<td>108</td>
<td>84</td>
<td>108</td>
<td>84</td>
<td>36</td>
</tr>
<tr>
<td>3</td>
<td>168</td>
<td>108</td>
<td>56</td>
<td>108</td>
<td>56</td>
<td>36</td>
</tr>
<tr>
<td>4</td>
<td>168</td>
<td>108</td>
<td>42</td>
<td>108</td>
<td>42</td>
<td>36</td>
</tr>
<tr>
<td>6</td>
<td>168</td>
<td>108</td>
<td>28</td>
<td>108</td>
<td>28</td>
<td>36</td>
</tr>
<tr>
<td>8</td>
<td>168</td>
<td>108</td>
<td>21</td>
<td>108</td>
<td>21</td>
<td>36</td>
</tr>
<tr>
<td>12</td>
<td>168</td>
<td>108</td>
<td>14</td>
<td>108</td>
<td>14</td>
<td>36</td>
</tr>
</tbody>
</table>

Table 6.3: Image configuration for image tile width measurements

To gain more insight in the causes for this variation in execution time we have to investigate further. In the investigation presented above only the width of the image tile is varied. The next section investigates the execution time when the height of an image tile is varied.

6.4.2 Variation in height of the image tile

The previous measurement showed there exists an optimal image tile width. By varying the height of an image tile it may follow this optimum exists for the image tile height as well. For this measurement the image configuration from Table 6.4 is used.

The result of this measurement is visible in Figure 6.4. It can be seen that indeed there exists an optimal height as well. For the Gears image the optimal image tile size is $56 \times 27$ pixels. Decreasing the width to 9 pixels increases the execution time with 93% while increasing the image tile width to 108 pixels increases the execution time with
6.4. SIZE OF THE IMAGE TILE

<table>
<thead>
<tr>
<th>Number of sections</th>
<th>resx</th>
<th>resy</th>
<th>secsizex</th>
<th>secsizey</th>
<th>dimx</th>
<th>dimy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>168</td>
<td>108</td>
<td>168</td>
<td>108</td>
<td>168</td>
<td>108</td>
</tr>
<tr>
<td>2</td>
<td>168</td>
<td>108</td>
<td>168</td>
<td>54</td>
<td>168</td>
<td>54</td>
</tr>
<tr>
<td>3</td>
<td>168</td>
<td>108</td>
<td>168</td>
<td>36</td>
<td>168</td>
<td>36</td>
</tr>
<tr>
<td>4</td>
<td>168</td>
<td>108</td>
<td>168</td>
<td>27</td>
<td>168</td>
<td>27</td>
</tr>
<tr>
<td>6</td>
<td>168</td>
<td>108</td>
<td>168</td>
<td>18</td>
<td>168</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>168</td>
<td>108</td>
<td>168</td>
<td>12</td>
<td>168</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>168</td>
<td>108</td>
<td>168</td>
<td>9</td>
<td>168</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 6.4: Image configuration for image tile height measurements

Figure 6.4: Execution time with varying image tile height

117%. Again small variations in image tile width have relatively little influence on the execution time. Decreasing the image tile width with 33.3% increases execution time with 7.6%, increasing the image tile width with 33.3% increases execution time with only 2.7%.

Both the optimal height and the optimal width seem to be around 32 pixels. There may be a correlation between the two values. Varying the width and the height at the same time might provide some insight in this matter, but is difficult to perform and to analyze.

6.4.3 Variation in width and height of the image tile

If both the height and the width of an image tile are varied an optimal value for both the width and the height might be found. Varying two values however has some drawbacks. The large number of measurements is one of these drawbacks. A more important
challenge is the right division in image tiles and sections. During the variation of the width a section was vertically divided into image tiles. A horizontal division of the section was used during the variation in height. To ease measurements while two values are varied, only one Worker is used. To optimize performance this Worker is placed on the second processing tile, thus processing tile configuration Q-W- is used. Because the image of $168 \times 108$ pixels does not fit in the memory of one processing tile, the largest tile rasterized is $84 \times 54$ pixels.

<table>
<thead>
<tr>
<th>Number of horizontal tiles</th>
<th>dimx</th>
<th>Number of vertical tiles</th>
<th>dimy</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>84</td>
<td>2</td>
<td>54</td>
</tr>
<tr>
<td>3</td>
<td>56</td>
<td>3</td>
<td>36</td>
</tr>
<tr>
<td>4</td>
<td>42</td>
<td>4</td>
<td>27</td>
</tr>
<tr>
<td>6</td>
<td>28</td>
<td>6</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td>21</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>14</td>
<td>12</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 6.5: Image configuration for image tile height and width variation

By choosing six values for the width and six values for the height we get a total of 36 measurements per image. The measurements for the Gears image are presented in several graphs. First a three-dimensional graph is created. This is shown in Figure 6.5. This image shows there is a local minimum, but it is not clear where this minimum is. Next a depth map is created, as can be seen in Figure 6.6. This shows the area with minimal execution time a little bit clearer.

Figure 6.5: Execution time rasterizing Gears with varying image tile size
6.4. SIZE OF THE IMAGE TILE

Figure 6.6: Execution time rasterizing Gears with varying image tile size

The contour lines in Figure 6.6 seem to have asymptotic behavior. This suggests the product of the width and height is of influence to the execution time of the rasterization process. To check this the measured data is presented in a third way. This time the width and height of the image tiles are multiplied to calculate the area. The resulting area size can be seen in Table 6.7. In Figure 6.7 the execution time is plotted against the area of an image tile.

<table>
<thead>
<tr>
<th>Execution time in megacycles</th>
<th>image tile width</th>
<th>image tile height</th>
<th>image tile area</th>
</tr>
</thead>
<tbody>
<tr>
<td>535</td>
<td>84</td>
<td>18</td>
<td>1512</td>
</tr>
<tr>
<td>541</td>
<td>56</td>
<td>36</td>
<td>2016</td>
</tr>
<tr>
<td>552</td>
<td>56</td>
<td>27</td>
<td>1512</td>
</tr>
<tr>
<td>552</td>
<td>42</td>
<td>36</td>
<td>1512</td>
</tr>
<tr>
<td>553</td>
<td>28</td>
<td>54</td>
<td>1512</td>
</tr>
<tr>
<td>566</td>
<td>84</td>
<td>27</td>
<td>2268</td>
</tr>
<tr>
<td>589</td>
<td>42</td>
<td>54</td>
<td>2268</td>
</tr>
</tbody>
</table>

Table 6.6: Fastest image tile sizes for Gears

Figure 6.7 shows there is a strong correlation between the area of an image tile and the execution time of the rasterization process. Again we focus on the Gears image. The minimal execution time is found on an image tile size of $84 \times 14$ pixels, or an area of 1512 pixels. In Table 6.6 the fastest seven image tile configurations are presented. We see that the fastest configurations have an area close to the optimal area of 1512 pixels. If we take the total execution time needed for rasterizing the four images we see that
for the fastest image tile configuration 1840 Megacycles are needed. The total execution time for the smallest tile size, with an area of only 126 pixels, we need almost 8864 Megacycles. This shows that in an extreme case execution time can be reduced with almost 80%. This result is investigated further in the next section.

### 6.4.4 Detailed investigation on image tile area

In the previous sections it was observed that for each image there exists an optimal image tile size. This section attempts to explain this observation. To find more information on the cause of this effect we use the tools introduced in Chapter 5 to profile the most time consuming functions of the rasterizer. In Figure 6.7 we saw the ratio between image tile width and image tile height had little influence on the execution speed. Therefore we select six image tile dimensions from Table 6.7. The selected dimensions are presented in Table 6.8. These selected dimensions are on the diagonal of Table 6.7 and represent a large range of tile area sizes with a ratio close to 1:1.
Using these image configurations we rasterize the four images at a size of $168 \times 108$ pixels again. This time we measure the execution time of the functions discussed in Chapter 5. Because we increased the size of the FIFO we had to add one function to the set of profiled functions. When the last element has been send to the Workers the Queen will have to wait until all the elements in the FIFO haven been rasterized by the Workers. When all the Workers are finished they are in sync again. Therefore we call this time “synchronization” time. Since this time is very short when the FIFO has a capacity of only one element, this function has not been profiled in Chapter 5 and Section 6.2. The results are visualized in Figure 6.8, Figure 6.9, Figure 6.10 and Figure 6.11.

![Graph of profiling functions with varying tile area using Gears](image)

**Figure 6.8: Profiling functions with a varying tile area using Gears**

For small image tiles we see that the greatest contribution to the total execution time (runframe) is \texttt{decode()}. Since the \texttt{decode()} function has to be called once per section, and small sections or image tiles imply a large number of sections, this is not unexpected. This can be solved by reintroducing the element buffer (Chapter 3.1.2.2) to store decoded elements when sufficient memory is available. For larger image tiles the
Figure 6.9: Profiling functions with a varying tile area using Highlight

Figure 6.10: Profiling functions with a varying tile area using Scube

function \texttt{process\_triangle()} becomes a great contributor to the execution time. This function is responsible for the identification (see Appendix B.2) and parametrization (see Appendix B.3) of the pixels in the image. The parallel tile-based rasterizer presented in Chapter 4 uses a triangle mask to communicate between these two processes. During
identification, this mask is filled with ones and zeros, depending on whether the pixel corresponding to the element in this mask is inside or outside the rasterized triangle. During parametrization only pixel parameters are calculated for which this triangle mask holds a one. When the image tiles are much larger than the average triangle, the triangle mask will hold a large number of zeros. Writing and reading this mask will consume more time than the parametrization of the pixels. This increases the total execution time for very large image tiles.

Other works, like [6] suggest the optimal size of an image is related to the size of the triangles. They calculate a size of $32 \times 32$ pixels to be optimal for their benchmarks. However, they explain the increasing execution time for larger images by the lower speed of larger memories. Since in this simulation the same memory is used for every tile size this can not explain why rasterization with very large tiles is slow compared to tiles of $\pm1000$ pixels. However by incorporating the triangle mask into our investigation we arrive at the same conclusion as [6], that “An optimal tile size can be determined from the average triangle size, so that most triangles can be rendered without being divided into sub-triangles.”

### 6.5 Load balancing

Now that we have fixed the size of the FIFO to nine elements, and have a rough idea of the optimal tile size, we can revisit the allocation of Workers on processing tiles.
6.5.1 Possible processing tile configurations

The maximal number of Workers on a tile for this work is fixed at two. This means that each processing tile can hold zero, one or two Workers. The total possible configurations thus becomes $3^3 = 27$. They are depicted in Figure 6.12.

One of these configurations, Q--, makes no sense, since it has no workers. Other configurations, like Q-W- and Q--W are functionally identical, since the second and third processing tile are equivalent. By removing the duplicates and the zero-tile, we have 17 configurations. They are depicted in Figure 6.13. Their characteristics are presented in Table 6.9.
For comparison we want to rasterize several images with the same dimensions on all 17 processing tile configurations. Table 6.9 shows we have configurations with 1, 2, 3, 4, 5 and 6 Workers. In 6.4 we demonstrated the size of the image tile is of influence on the execution time of the rasterization process. Therefore it is important to use the same tile-size on all configurations. We have several soft constraints for the next measurements. These constraints are:

- The rasterized image has a ratio of approximately 2:3.
- The rasterized image has a resolution \( \leq 16384 \) pixels.
Table 6.9: Processing tile configurations using three processing tiles

<table>
<thead>
<tr>
<th>Name</th>
<th>Number of tiles used</th>
<th>Number of Workers</th>
</tr>
</thead>
<tbody>
<tr>
<td>QW--</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>QWW--</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>QWW-W-</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>QW-W-</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Q-W-</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>QWW-WW-</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Q-WW-</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>QWW-W-W</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>QW-W-W</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>QWW-WW-W</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>QWW-WW-WW</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Q-W-W</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>QWW-WW-WW</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Q-WW-W</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Q-WW-WW</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

- The tile should have an area of around 1000 pixels.

And we have some hard constraints, which are as follows:

- The image size and image tile size should be equal among all measurements.

- A section should contain 1, 2, 3, 4, 5 or 6 image tiles.

- An integer number of sections should fit within the image.

After calculating the possibilities, an image size of 144 × 100 pixels is selected. 144 can be divided by 2, 3 and 4, while 100 can be divided by 2 and 5. The tile size is fixed at 12 × 20 pixels. This way 12 horizontal tiles and 10 vertical tiles fit within one image. Unfortunately this means that one soft constraint, to let the area of one tile be close to 1000, has not been met, since the area is only 12 · 20 = 240 pixels. Figure 6.7 suggests execution time can be over 100% longer at a tile area of 240 pixels.

6.5.2 Execution time on one processing tile

If just one tile is available there are two possible processing tile configurations. The measurements of both tile configurations are depicted in Figure 6.14. Using two Workers instead of only one Worker seems to boost the performance in all configurations. This can be explained by the fact that the decode() function has to be called for each section. Using two Workers instead of one decreases the total number of calls to decode() while simultaneously decreasing the percentage of execution time spent in the Queen process.
6.5. LOAD BALANCING

<table>
<thead>
<tr>
<th>Nr. of tiles per section</th>
<th>secsizex</th>
<th>secsizey</th>
<th>horizontal tiles</th>
<th>vertical tiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>20</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>24</td>
<td>20</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>36</td>
<td>20</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>48</td>
<td>20</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>100</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>72</td>
<td>20</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 6.10: Image configurations using three processing tiles

If we sum the execution time of the four images we can see that the QWW-- configuration needs 22% less cycles to rasterize all images than the configuration with two Workers on the first processing tile, configuration QW--.

6.5.3 Execution time on two processing tiles

Six different processing tile configurations can be made with two processing tiles. The result is visualized in Figure 6.15. The fastest tile configuration is Q-WW-, the version with two Workers, on a tile separate from the Queen. Adding one Worker to the first tile, as in the QW-WW- setup, decreases performance. This can be best explained by the percentage of execution time spent on the Queen. Although less calls to `decode()` have to be made when the number of tiles per section increases, the Queen will only have a share of the computation time on the first processing tile. The other part is used by the Worker on the same processing tile.

If we compare the fastest configuration, Q-WW-, with the fastest configuration with
only one processing tile in Figure 6.14 QWW--, we can see the execution time needed to rasterize the four images decreases from 5992 Megacycles to 2286 Megacycles. This calculates to a decrease of 62% by doubling the number of processing tiles.

![Configuration vs Megacycles](image)

Figure 6.15: Execution time with different worker allocations over two processing tiles

### 6.5.4 Execution time on three processing tiles

Finally the processing tile configurations with three processing tiles are measured. The result is presented in Figure 6.16. The fastest configuration in this case is Q-WW-WW. The execution time to rasterize the four images is decreased to 1275 Megacycles. This is a decrease of 44% compared to the fastest configuration with two processing cycles. Compared to the QWW-- configuration we see that tripling the number of processing tiles decreases the execution time with 79%. Since this is super linear speedup care has to be taken. It is likely that the QWW-- is a suboptimal configuration when one processing tile is available. Future investigations might show a further speedup can be achieved by adding a third or a fourth Worker to the processing tile and measure the execution time on the QWWW-- and QWWWW-- configurations. Unfortunately the used implementation is limited to two workers per processing tile.

The importance of choosing the right configuration is visible if we compare the results in Figure 6.15 and Figure 6.16. The best configuration using two tiles, configuration Q-WW-, needs 2286 cycles to rasterize the four images. The worst configuration using three tiles, configuration QWW-W-W needs 3084 cycles, almost 35% more, for the same purpose. This can be explained by the fact that QWW-W-W uses three processes on the first processing tile and only one on the second and the third. The best configuration using three tiles, configuration Q-WW-WW, saves almost 57% execution time compared
6.6 Load balancing in a decreased solution space

In the previous section the quest for an optimal processing tile configuration was initialized. We investigated the performance of seventeen different processing tile configurations. To be able to do this we had to meet a number of constraints presented in Section 6.5.1. The large number of processing tile configurations made it impossible to meet all constraints imposed by previous measurements in a sufficient way. By limiting the number of processing tile configurations we decrease our solution space and make it easier to meet all constraints.

6.6.1 Measured tile configurations

In Table 6.9 seventeen different processing tile configurations are presented. Three configurations use five or six Workers. In the previous section they did not prove to be the most efficient configurations. To ease the hard constraints mentioned in Section 6.5.1...
they are removed from the solution space. The hard constraints now become:

- The image size and tile size should be equal among all measurements
- A section should contain 1, 2, 3, or 4 image tiles
- An integer number of sections should fit within the image

To be able to compare the new measurements with the measurements in Section 6.5.1, the size of the image is fixed at 144 × 100 pixels. The new image configuration is presented in Table 6.11.

<table>
<thead>
<tr>
<th>Nr. of tiles per section</th>
<th>secsizex</th>
<th>secsizey</th>
<th>horizontal tiles</th>
<th>vertical tiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>100</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>24</td>
<td>100</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>36</td>
<td>100</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>48</td>
<td>100</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.11: Image configurations using up to 4 Workers

A huge improvement is made by increasing the tile area from 12 × 20 = 240 pixels to 12 × 100 = 1200 pixels. According to Figure 6.7, rasterizing the Gears image using an image tile area of 252 pixels takes 2167 Megacycles. When the image tile area is increased to 1134 pixels the execution time decreases with 616 Megacycles, a relative decrease of 72%.

6.6.2 Execution time on one processing tile

First the image is rasterized with only one Worker. The result is presented in Figure 6.14. Compared to Figure 6.14, three important observations can be made:

1. The fastest configuration for the Gears image is still QWW--.
2. The fastest configuration for the Surfgrid image has changed to QW--.
3. On the QWW-- configuration Gears is faster than Surfgrid.
4. The execution time to rasterize all the images has decreased with 41%.

To explain these observations we need to take a look at the images used for benchmarking. They are described in Chapter 5.1. Table 5.1 shows that the TKI-file for Gears is larger than the TKI-file for Surfgrid. The number of TKI-instructions for Gears is also larger than for Surfgrid. Decoding the instruction file will thus be a lot slower for Gears than for Surfgrid. By increasing the size of an image tile with a factor five, and thus the size of a Section, the number of times `decode()` has to be called decreases with a factor five. This is very beneficial for the Gears image. The rasterization of Surfgrid image however, will have little benefit from this. Its bottleneck will be in the Worker process. If we look at the image of Surfgrid in Chapter 5.1, we see some black in the background.
This can easily be rasterized. When a tile containing only black is rasterized in parallel with a more complex tile, the Queen will wait for the last tile to be finished. This indicates that when the triangles in an image are equally distributed the speedup gained from decreasing the image tile size is bigger.

### 6.6.3 Execution time on two processing tiles

In Figure 6.18 the results of rasterizing the four images with larger image tiles can be seen. Again the Surfgrid configuration has become slower than the Gears image in almost all configurations. The sequence of fastest configurations has changed as well. The fastest configuration is QW-WW- when the image tile area is 1200 pixels. In Section 6.5 we rasterized the same images with an image tile area of 240 pixels. In that case the Q-WW- configuration was the fastest. From Figure 6.8 we can conclude that with an image tile area of 240 pixels most of the execution time is spent in this function. Since this function is part of the Queen process, adding a Worker process to the same processing tile decreases the fraction of the processing time available for the Queen. This explains why the QW-WW- configuration performs worse than the Q-WW- configuration in Section 6.5.

If we take the sum of the execution time of all images we can compare this configurations to two different configurations. Compared to the fastest configuration with two processing tiles in Section 6.5 we see a decrease in execution time of 22%. This speedup has been gained by increasing the image tile area from 240 pixels to 1200 pixels. Compared to the QWW-- configuration in Figure 6.17 the execution time is decreased with 49% by adding a second processing tile.
6.6.4 Execution time on three processing tiles

The results of the last measurement with different configurations are presented in
Figure 6.19. Again the sequence of fastest configurations and fastest image has changed compared to the measurements in Section 6.5. Compared with the fastest configuration with two processing tiles in Section 6.5 we have a decrease in execution time of only 8.2%. From this we conclude that the decrease in execution time gained by choosing the right image tile size depends highly on the chosen processing tile configuration.

Eventually we compare the results of the fastest configuration with three processing tiles to the fastest configuration with only two processing tiles. In Figure 6.19 we see that Q-W-W is the fastest configuration using three processing tiles. This configuration needs 1170 Megacycles to rasterize the four images. From Figure 6.18 we conclude that QW-WW- is the fastest configuration with two processing tiles and has an execution time of 1787 Megacycles. By adding a third core and choosing the right configuration the execution time can thus be decreased with 35%. Compared to the fastest configuration with one processing tile in Figure 6.19 a decrease in execution time of 67% by tripling the number of processing tiles is achieved. This indicates that, under the given circumstances, a linear speedup is possible.

6.7 Linear Edge Test Algorithm

In Chapter 2.1.2 two algorithms are discussed for creating the tilemask: the simple Bounding Box-algorithm, and the more precise but complex Linear Edge Test algorithm. All previous measurements have been executed using the Bounding Box algorithm. The disadvantage of BBOX is that it can lead to false positives: some elements will incorrectly be sent to a Worker. Because communication over the FIFO is expensive this may lead to an increase in execution time. Therefore this section presents measurements with the LET-algorithm. Figure 6.20, Figure 6.21 and 6.22 present the results of these measurements, performed with one, two or three processing tiles and using the same configuration as in Section 6.6. We can conclude that changing the tilemask algorithm has negligible effect on the execution time. This suggests that either the overhead added by extra communication for the LET-algorithms equals the decreased communication overhead compared to the BBOX-algorithm, or both types of overhead have insignificant contribution to the computation times. This was already suggested in [31], where it is concluded that „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“.

6.8 Conclusion on changing parameters

From the results presented in this chapter it can be concluded that changing the configuration or the size of an image tile has a complex influence on the execution time. All the presented measurements however show that increasing the number of processing tiles can indeed speedup rasterization. Care has to be taken to choose the right configurations. It might not be trivial to choose the right parameters for a given image. We have seen that choosing the best processing tile configuration can decrease the execution time with 67% for an image tile area of 1200 pixels. For an image tile of 240 pixels we see that a speedup of 79% can be achieved, this number indicates that the processing tile configuration with
only one Worker can benefit from using more than two Workers on the same tile. The size of the image tile has an optimum value, as suggested by previous work. Furthermore
the performance difference between the LET-algorithm and the BBOX-algorithm can be neglected.
In this work we made a parallelization study on tile-based rasterization. The possibility to improve the rasterization speed of a rasterizer by incorporating multiple processors on a System on Chip has been investigated. For this investigation we developed a rasterizer that is able to make effective use of the presence of multiple processing elements, such as Multi-Processors on a System on Chip (MPSoC). This chapter presents a summary of the work and an overview of the contributions made in Section 7.1. Section 7.2 discusses the limitations of the current implementation. This section discusses some advantages that can be achieved by overcoming these limitations and how this can be achieved. Finally in Section 7.3 a number of recommendations are given to further improve multi-process tile-based rasterization.

7.1 Summary and contributions

Computer graphics require a significant part of the resources available on a computer system. While the performance of computing elements is improving over the years, energy consumption is always a bottleneck. Both the increasing processor speed and increasing memory sizes are responsible for a large amount of energy usage. With the success of smartphones the use of advanced graphics on mobile devices has been increasing lately. On these devices a trade-off has to be made between energy consumption and portability. Therefore minimizing power consumption of Computer Graphics on mobile devices, or embedded systems in general, deserves investigation.

The most resource demanding stage in computer graphics is often the last stage in the pipeline, namely rasterization. This is the process in which triangles and other primitives are converted to pixels. To decrease memory usage and thus power usage tile-based rasterization has been proposed. This technique allows rasterization to be performed with smaller memories. The image is split in a number of smaller images, called tiles or image tiles. By rasterizing one image tile at a time less memory is needed. Instead of using one large memory for all computations, a local fast memory is used for rasterization. The result of different image tiles is written to a slower memory of medium size. Together they form the entire image.

Although a lot of research on tile-based rasterization has been done, none of the existing approaches concerns the usage of multiple homogeneous processors as an underlying hardware platform. Instead they are aimed at application specific hardware, such as GPUs, or Single Instruction Multiple Data processors. These processors offer high speed concerning multimedia tasks such as rasterization, but are very limited in use for other applications.

This work investigates the possibility to use an existing homogeneous Multi-Processor System on Chip, namely the CoMPSoC platform, to speed up image rasterization in a
tile-based fashion by adding processors. To perform these investigations some new tools have been developed and existing tools have been adapted. After the development and modifications were finished, a number of experiments have been performed. From these measurements some conclusions are drawn. Our contribution to the field of computer graphics can be split in two parts. The first part covers the software and hardware implemented to perform the experiments. The second part answers the two open questions proposed in the introduction (Chapter 1). The implementation related contributions are as follows:

- A tile-based rasterizer written in C++ to run on a desktop PC is modified so that the decoding process, the Queen, and the rasterizing process, the Worker, can run in different POSIX threads. Next, this implementation is enhanced to allow multiple Worker processes to be used in a configuration with just one Queen. This enables the tile-based rasterizer to make efficient use of multiple processing elements. The new rasterizer is called a parallel tile-based rasterizer.

- The operating system running on the CoMPSoC platform, CompOSe, is written in C. Some modifications are made so that it can be used to run a program written in C++. Subsequently the parallel tile-based rasterizer is modified and compiled to run on the CoMPSoC platform. The CompOSe support for running multiple processes within one application is used to run multiple Workers along with one Queen on the same processing tile.

- The parallel tile-based rasterizer is adapted to write the rasterized image tile to a shared framebuffer, and thus make it possible to use different processing tiles for the rasterization process. Furthermore a new CoMPSoC platform prototype has been instantiated. This platform has enough local memory and enough shared memory to make it possible to do experiments with parallel tile-based rasterization.

Using the software and hardware described above, we were able to answer the questions from Chapter 1 as follows:

1. The rasterization process is split in two parts, the Worker and Queen. While the Queen runs on one embedded processor, multiple Workers can be distributed over all available processors. The communication between the Queen and the Workers is minimized while there is no communication between the Workers. The Workers use only local memory for computation intensive operations. This way the total execution time for performing tile-based rasterization can be reduced by adding more processors. Experiments have shown that by using three instead of one processor the execution time for rasterization can be reduced by 67% under realistic circumstances.

2. Several trade-offs have been investigated using the proposed platform. The most important factors influencing the rasterization speed are:

- The size of the FIFOs from the Queen to the Workers. Increasing the FIFO decreases probability of a full buffer. This reduces the time that processes are
7.1. SUMMARY AND CONTRIBUTIONS

stalled by other processes. Increasing the size of the FIFOs from one element
to two elements decreased execution time with over 4%, increasing the size
of the buffer with a third element decreased execution time with over 2%.
The drawback of increasing the size of the FIFO is the increasing memory
consumption. Both experiments and the theory indicate that for large FIFOs
increasing the size further will have limited effect on the execution speed. It
was however not possible to determine the optimal size. The memory reserved
for communication in our prototype is limited to 8KiB, while the size of one
element in the FIFO is 300 bytes. Since one processing tile can run two
Worker processes, this puts the theoretical limit to the number of elements in
a FIFO to 13. Because the communication memory is used for other forms of
communication as well this limit is even lower. A maximum of nine elements
can be stored in the memory. This led to an average decrease of 15% in
execution time.

• The size of an image tile. Previous research concluded the optimal size of an
image tile is $32 \times 32$ pixels. The experiments we conducted indicated that
the optimal image tile size for the rasterizer implemented for this work does
not differ from previous research. Small deviations from the optimal image
tile size has little influence on the execution time, but larger deviations will
have a significant influence on the efficiency of the rasterizer. For example,
one experiment showed an execution time that doubled when the width of
the image tile was increased from 36 to 168 pixels. The ratio between width
and height of an image tile proved to be of little significance. This made it
possible to experiment with image tile areas as small as 126 pixels up to an
image tile area of 4536 pixels. By choosing the right image tile size it is shown
that the execution time can be decreased by 80%.

• The number of tasks or processes per processor. The first processor runs the
Queen. Each processor, including the first one, can run zero, one or more
Workers. For this work we limited the number of Workers to a maximum of
two. Experimental results show that the most efficient number of processes per
Worker depend on the image being rasterized and the size of the image tiles.
When the Queen process is the bottleneck of the rasterization process e.g., it
is efficient to run only the Queen process and no Worker processes on the first
tile. It has been shown that proper load balancing can decrease execution
time with up to up to 57% while using the same number of processors.

While the factors summarized above have a significant influence to the execution
time of the parallel rasterizer it has been confirmed that the algorithm used to
calculate the tilemask has negligible influence on the rasterization speed.

The experimental results show that tile-based rasterization on a MPSoC prototype
promises to be a scalable solution. The used platform however had some limitations
that restricted the number of experiments that could be performed. These limitations
will be discussed in the next section.
7.2 Overcoming limitations of the current implementation

The tools used in this work have several limitations. The tile-based rasterizer that is the starting point of this work is a simple rasterizer that has only been used for academic research. Although its simplicity allows for easy modifications, some parts of the rasterizer should be optimized for better performance. Furthermore, the hardware prototype has some limitations. Because the FPGA-board used has limited resources, every addition to the platform asks for a compromise on another part. More processing tiles can be added, but at the expense of less available memory per processing tile. A larger shared memory might be beneficial for some experiments, but this would be at the cost of local memory etc. This section suggests some solutions to overcome the limitations of the current implementation.

- Use DDR memory. The used FPGA has a relatively large DDR-memory available. The main reason this memory has not been used is that it currently is not possible to combine the usage of DDR with an accurate platform monitor. Using another monitor would decrease the means for feedback to the user. If this feedback can be arranged otherwise, a lot of improvements can be made. The shared SDRAM can be removed to create space for other devices, or can be used for other purposes. The entire image that can be rasterized can be a lot larger. Larger images might have other characteristics than the images used in this work.

- Start with a current version of the MESA library. The front-end is based on an old implementation of the MESA library. This generates a TKI file with TKG operands. Both are undocumented. By updating the MESA version and documenting the instructions used to transfer commands between the back-end and the front-end further research might be easier. Moreover, the developed tools can be used in a larger context over a longer term, without becoming obsolete or hard to maintain.

- Add support for textures. The current rasterizer does not have support for textures. Textures will add the need for more communication between the Worker and the Queen. Because multiple triangles might use the same texture, the texture should be stored on the Worker as well as on the Queen. The influence of parameters on the performance might change by the use of textures.

- Merge the identification stage and parametrization stage. In Chapter 6.4.4 we investigated why increasing the size of an image tile increases the execution time of the rasterization process. It is concluded that the size of the triangle mask might be responsible for this effect. This mask is used to transfer data from the pixel identification stage to the pixel parametrization stage (see Appendix B.2 and Appendix B.3). If each pixel is parametrized immediately after identification this triangle buffer can be discarded. This will save a little bit of memory. Moreover, it might decrease the execution time for large tiles.
7.3 Topics for further research

Next to the limitations of the tools, there are still a lot of questions unanswered by this work. Further research, based on issues that are found in this work, are recommended in this section. They are as follows:

- Using an element buffer. The decode() function of the Queen performs the decoding of instructions into elements. This has to be repeated for each section. Especially for large images with small image sizes this is a time consuming process that has to be done a large number of times. Instead the decoded element could be stored in a buffer. This buffer has to be very large for complex images and might not be stored on internal memory. A trade off between communication overhead and speed can be made.

- Non-synchronous tile rasterization. The number of processing tiles on the CoMP-SoC platform is limited. Adding too many Workers will decrease the performance per Worker. Furthermore the local memory on a processing tile is limited. Finally there is an optimal size for an image tile. For large images this means the image cannot be rasterized at once. The image is therefore split into Sections which can be rasterized at once. This approach has two important disadvantages:

  1. Data locality: The size of the FIFO is limited. When the FIFO can contain \( n \) elements, and more than \( n \) consecutive elements are only intersecting with one of the image tiles being processed, the Workers processing the other image tiles will be idle.

  2. Unbalanced image tiles: Not all image tiles in one section will have the same complexity. If a complex image tile and a simple image tile are rasterized in parallel, the Worker rasterizing the simple image tile will be idle for most of the time.

A solution for the data locality-problem is to increase the size of the FIFO. This way the problem will have less influence on the throughput. However, locality of triangles may introduce the need for very large FIFO buffers.

The problem with unbalanced tiles is more prominent. If an image tile has no or a small number of triangles, but is in a section with more complex image tiles, the processor will be idle for most of the time. This can be solved with another communication strategy. Instead of pushing new elements into the buffer, the Queen should respond to requests from the Workers. Each time a Worker needs a new element the Queen can iterate through a list of elements and find elements that still need to be rasterized in the particular image tile. Once there are no elements for this image tile left, the Worker should write its local tile buffer into the global framebuffer. When this is finished it can request a new image tile to be rasterized. To use this technique a number of improvements have to be implemented:

  1. A new, request-based communication scheme is needed.
2. The Queen needs to create a list with all the elements before the Workers can start issuing image tiles and Elements. The element buffer mentioned above can be used for this functionality. This list should be saved in the memory.

3. In general, the Queen will need more processing time and more memory than in the current solution. Compared to the communication model presented in this work, this may lead to other processing tile configurations being optimal. E.g., processing tile configurations in which the Queen shares a processing tile with one or more Workers might prove inefficient.

- Change-based state communication. In the current implementation each element, thus effectively each triangle, has its own state. Each state has a number of state variables that can be changed independently. In practice not all state variables will differ between consecutive triangles. Even more, consecutive triangles might have exactly the same state. Instead of sending the entire state over the network with each triangle, it might be advantageous to only send the changed state variables, or only send a new state when the current state has changed. This requires a new way of communication, e.g., a second FIFO from the Queen to each Worker might be used. This new scheme might only improve the performance on scenes with a small number of state changes.

The number of state changes can further be reduced by sorting the elements before sending them over the FIFO. Once again, decoding and storing all the elements before starting to send them over the FIFO is needed for this improvement.
OpenGL is a heavily used Graphic Library. It is used for rendering 3D objects, or converting 3D scenes into bitmap images. In the first section of this appendix the base elements of pictures are discussed. Next Section A.2 discusses a general 3D graphics pipeline on a conceptual level. The pipeline is divided into blocks or stage, where the output of each pipeline stage is the input of the next stage. In the following sections OpenGL is discussed. OpenGL is a more specific implementation of a general 3D graphics pipeline. First OpenGL is discussed as a State Machine. Finally the OpenGL pipeline itself is dealt with.

This appendix does not aim to give a precise implementation of a graphics pipeline, but it provides the reader with a general idea about the stages that are dealt with before rasterization takes place. The rasterization stage is discussed in the next Appendix.

Figure A.1: Pixels and vectors
A.1 Pixels and triangles

For the understanding of OpenGL it is important to understand the basic concept of images. Today digital images can be built up from two different base elements. On one hand are the images built up from pixels, the others are vector based images.

A.1.1 Pixel based images

The category “pixel based images” includes, among others, photos made with digital cameras. Each light sensitive element on the sensor of a digital camera results in one pixel. If a photo is enlarged, there will come a point in which only rectangular or square elements of the same size are visible, these elements are called pixel elements or pixels. In Figure A.1(a) an example of a picture build from pixels is shown. A detail of this image in enlarged in Figure A.1(b). The base elements, little squares called pixels, are visible in this enlargement.

A.1.2 Vector based images

Pictures can also be generated with vectors. Those pictures consist of points which are connected by lines. Multiple lines together can form areas. If straight lines without curves are used, these areas are called polygons. A map or a logo can be presented as a vector image. When a vector based image is enlarged polygons of variable size will become visible. This is shown in Figure A.1(d), which shows an enlarged detail of Figure A.1(d).

A.1.3 Three-dimensional images

![Figure A.2: A sphere build up from a different number of triangles](image)

Three-dimensional images created with, e.g., OpenGL often aim for a photo-realistic effect. The base elements of an OpenGL image, however, are triangles with three dimensions. Those triangles are not created with (two-dimensional) points, but three-dimensional vertices instead. To view them on pixel based devices like monitors or printers, they need to be converted to two dimensional pixel based images. In Figure A.2 spheres are shown, build up with a different number of triangles. While the leftmost sphere looks like a perfect sphere, it is obvious the rightmost sphere consists of triangles.
A.2 A 3D Graphics Rendering Pipeline

This section discusses the functionality of a general 3D graphics pipeline. Although this work uses the OpenGL pipeline specifically, the concepts of a general 3D graphics pipeline apply to a multitude of implementations. After discussing the general 3D graphics rendering pipeline the OpenGL pipeline is discussed in more detail.

![Diagram of the three stages of a general graphics pipeline]

A General 3D graphics pipeline consists of three main stages (see Figure A.3). These stages are:

1. The application stage. In this stage an application creates the three-dimensional “world” or scene. It consists of defining objects in a three-dimensional space.

2. The geometry stage. In this stage the three-dimensional objects are converted to primitives, e.g., two-dimensional triangles. Transformation and lighting calculations are performed to achieve this.

3. The rasterization stage. In this stage the primitives, that is points, lines and triangles, are converted to pixels that can be displayed on the screen.

The following subsections elaborate on these stages.

A.2.1 Application stage

The application stage is usually implemented by a program or application running on a computer or device that generates the image to be displayed. This application can be a game, a CAD-program and nowadays even a browser (WebGL) or desktop manager. The application handles a large number of actions, among which the interaction with the user. In e.g., a game the user can use the mouse and/or keyboard to manipulate the image. These manipulations can involve the movement of objects (e.g. actors that move), but they can also change the position of the light source(s) or the viewing point.

![Diagram of a simple projection in the application stage]

In Figure A.4, an impression is given to visualize an image in the application stage. The image consists of points on a three-dimensional grid. These points form lines, which
form surfaces, which form objects. It is important to realize that in this stage the viewpoint, viewing frustum or camera, is also positioned on the three-dimensional grid. The same goes for the light(s). In Figure A.4 the object is a simple house. The light is printed in yellow, while the viewing frustum is purple. The viewing frustum encapsulates the part of the image that is displayed in the final result. The frustum in Figure A.4 is only used to denote the shape and direction of the frustum. Thus the image is only limited in two directions, or by four planes. If a minimum and a maximum distance are set all six planes of the frustum will limit the image.

Note that objects that are not inside the viewing frustum (or: outside the camera view) can also be present in the application stage.

The application stage is relatively flexible. The second and third phases can be regarded as stages that are more straightforward and only concern the final image. The application stage however can be interactive. Not only can it respond to input given by the user, the output of this stage might also depend on other variables, like input from an external source (network), time of the day (displaying a clock) or even itself. Collision detection involves the procedures needed to determine whether objects touch each other and should react upon this, e.g., by bouncing back. The output generated by the application stage serves as an input for the next stage: the geometry stage.

### A.2.2 Geometry stage

The geometry stage can be regarded as a pipeline with five stages, as displayed in Figure A.5. The majority of the operations involving polygons or vertices are performed in this stage. Both the input and the output of the geometry stage are three dimensional images. However, the coordinates of the vertices differ. The input of the stage uses x-, y- and z-coordinates to refer to a certain point in Cartesian space. The x-, y- and z-value can range from $-\infty$ to $+\infty$. The output on the other hand uses x and y to refer to the point relative to the output image, e.g., to the computer screen. The coordinates are mapped to a limited range, e.g. (0,0,0) as the lower left corner of the screen and (1,0,1) as the upper right corner. The z-value is also limited to a certain range, but can often be negative. It is used to tell the last stage, the rasterization stage, which object is in front of another object. E.g. when the range is set from $-1.0$ to $+1.0$, a point at $Z = 0.3$ is behind a point at $Z = 0.2$. Note that the difference between z-values does not give any information about the distance between vertices in the original three-dimensional space. This information is no longer needed and thus disregarded by the geometry stage.

The stages of the geometry stage is discussed in this subsection. Mind that the sequence is one of the many possible sequences. Also the edge between two stages might not be as sharp as suggested by the division: multiple stages might be handled in one action.
A.2. A 3D GRAPHICS RENDERING PIPELINE

### A.2.2.1 Model & View Transform

Vertices, lines and polygons are presented in three dimensional space. Using transformation matrices a multitude of transformations can be performed to change the position or rotation of an object. The position of each vertex can be given by just three parameters, the x-coordinate, the y-coordinate and the z-coordinate. These vertices can be transformed using a transformation matrix. Using a $3 \times 3$ matrix one can perform operations like:

- Scaling
- Rotation
- Mirroring

For example, the matrix $T_r$, used to rotate a point around the z-axis with angle $\phi$, is given by [31] and shown in Equation A.1.

$$T_r = \begin{bmatrix} \cos(\phi) & -\sin(\phi) & 0 \\ \sin(\phi) & \cos(\phi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (A.1)$$

If we take a point represented by vector $P$ and rotate it around the x-axis with angle $\phi$ we find the rotated version of $P$, called $P_r$ (see Equation A.2).

$$T_r \times P = \begin{bmatrix} \cos(\phi) & -\sin(\phi) & 0 \\ \sin(\phi) & \cos(\phi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x \cdot \cos(\phi) - y \cdot \sin(\phi) \\ x \cdot \sin(\phi) + y \cdot \cos(\phi) \\ z \end{bmatrix} = P_r \quad (A.2)$$

However for other transformations, like translation and perspective transformations, a $3 \times 3$ matrix does not suffice. Transformations can be performed using an addition, e.g., moving the point given by $P$ over the x, y and z-axis using a displacement vector called $D$ can be done as show in Equation A.3.

$$P + D = \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} x + u \\ y + v \\ z + w \end{bmatrix} = P_t \quad (A.3)$$

Addition instead of multiplication has an important disadvantage. If an image has to be scaled, rotated and mirrored using transformation matrices $T_s$, $T_r$ and $T_m$, each point $P_i$ should go through the transformations individually. Thus the computations in Equation A.4 should be performed for each point.

$$P_i' = T_s \times P_i \quad P_i'' = T_t \times P_i' \quad P_i''' = T_m \times P_i'' \quad (A.4)$$

However, since matrix multiplication is associative, multiple transformation matrices can be multiplied to form a new transformation matrix, as shown in Equation A.5.

$$P_i''' = T_m \times (T_t \times (T_s \times P_i))) = (T_m \times T_t \times T_s) \times P_i = T_{mts} \times P_i. \quad (A.5)$$
This means that instead of performing three transformations on each point, the transformation matrices can be multiplied to create a combined transformation matrix. Thus each point has to be multiplied with only one matrix. Note that matrix multiplication is not commutative. Therefore it is important that the order of the transformations is preserved.

Since translating a 3-dimensional object can not be done using a 3×3 matrix, a fourth dimension is introduced, thus creating a homogeneous coordinate system. Next to the real coordinates x, y and z, we introduce a fourth coordinate, called W. Though this value can be constant, often one, the transformation matrix now becomes a 4×4 matrix, which enables transformations like perspective transformation and translation. An example of a translation done using a 4 transformation matrix is shown in Equation A.6. This equation can be compared to Equation A.3.

\[
P \times T_t = \begin{bmatrix} 1 & 0 & 0 & u \\ 0 & 1 & 0 & v \\ 0 & 0 & 1 & w \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + u \\ y + v \\ z + w \\ 1 \end{bmatrix} = P_t \tag{A.6}
\]

The objects at the input of this stage are located relative to a World Coordinate System. After the View Transformation the position of the objects are relative to the viewing frustum.

A.2.2.2 Lighting and Shading

In this stage the lighting properties of primitives are calculated. In Figure A.4 a yellow light source can be seen. Multiple light sources can be present in a scene. Depending on the properties of the light source and the distance relative to this light source the lighting properties of primitives are calculated. These properties can be different per triangle or even per vertex.

The actual calculations of light properties are difficult. For photorealistic effects a technique called ray tracing is used. This technique simulates the path of a light ray or photon. This way transparency and reflection can be calculated with a high level of realism. Unfortunately these calculations are too complex to be used in real time systems. Therefore most 3D pipelines use several approximations, called shading models.

A.2.2.3 Projection

After the Lighting and Shading Phase each pixel still has three coordinates. The final image however needs to be 2-dimensional. Since the image is already transformed to the viewing frustum this could be done by discarding the z-value. This would however create an image without any perspective: two triangles with the same size will always be projected the same size, independent of their distance to the viewing point. In Figure A.6(a) the process of mapping the scene to a viewport using an orthogonal projection is visualized. The resulting projection can be seen in Figure A.7(a). Parallel lines in the original image stay parallel in this projection.

Perspective projection on the other hand is a more complex projection. Objects look smaller when they are further away and parallel lines that are not parallel to the viewing
A.2. A 3D GRAPHICS RENDERING PIPELINE

Figure A.6: Projecting a scene seen from the viewing frustum

plane are not projected in parallel. This process is visualized in Figure A.6(b). The result is shown in Figure A.7(b).

Figure A.7: Projected scene seen from the viewing frustum

A.2.2.4 Clipping

As mentioned before not all primitives will be visible on the final image. Some primitives are blocked by other primitives that are closer to the cameras. Those primitives will be handled during the rasterization stage. Some primitives might fall outside of the viewing frustum. Those primitives are discarded in the clipping stage. Primitives that are not entirely covered by the viewing volume, the frustum, are replaced by primitives that are.
This might result in one triangle that is partly outside of the viewing windows being replaced by several smaller triangles that are entirely covered.

A.2.2.5 Screen Mapping

In the last stage of the geometry pipeline the image is mapped to the screen space or viewport. The objects visible to the viewing frustum are scaled and translated to obtain window coordinates. Often after this stage the x- and y-coordinates will fall between 0.0 and 1.0. The depth coordinate is also saved, and often called the z-value.

A.2.2.6 Geometry stage implementation

The implementation of the geometry pipeline can be done both in hardware and in software. The stage has a well defined functionality and thus does not need the flexibility required by the application stage. On the other hand, while the transformations need computation intensive matrix multiplications, the number of computations is relatively low. This is due to the fact that the number of primitives is relatively low compared to the number of pixels in an image.

A.2.3 Rasterization stage

![Image after rasterization](image.png)

Figure A.8: Image after rasterization

The last stage of a 3D graphic rendering pipeline is the rasterization stage. In this stage the primitives, vertices, edges and triangles, are converted to pixels. Because this stage is computation intensive, it is often realized in hardware. In this work however the rasterization is handled by software, because the development of specialized hardware is time and resource consuming. Because this work concentrates on the rasterization stage, this stage is discussed in [B]. The result of the rasterization stage is visible in Figure A.8.
A.3 The OpenGL Pipeline

In Section A.2 a general 3D pipeline is discussed. The OpenGL implementation of this pipeline is discussed in short, because this is the pipeline that is used by the MESA-library. OpenGL can be considered to be a state machine. The machine has a large number of state variables, like the color, position and characteristics of light etc. Each object that is drawn inherits these properties. When a state variable changes, this has consequences for all the new elements.

The OpenGL pipeline consists of a number of stages \[26\], which is treated in short in this section. The relationship between the stages is depicted in Figure A.9. The most important step, the rasterizing stage, is explained more elaborately. Note however, that not all stages in the official OpenGL pipeline can be mapped to a single block or a single operation. In some implementations of the pipeline multiple stages can be collapsed into a single operation. On the other hand, multiple operations might be needed to perform only a single stage in the pipeline.

Input to the OpenGL pipeline is pixel data, like textures, and vertex data, like geometrical objects. They are created by an application.

A.3.1 Display Lists

All data needed for creating an image, including both geometry data and pixel data, can be saved in a display list. When a display list is “executed” this data is used for creating an image. It is also possible to create images without using a display list. Instead of
saving data in a list, in that case the data is send immediately to the OpenGL pipeline, therefore this mode is called “immediate mode”.

A.3.2 Evaluators

The next stage of the pipeline, the “per-vertex operations stage”, operates on vertices. Curves and surfaces from the display list however may also be described as basic functions, such as control points and polynomial functions. This stage converts these functions to surface normals, texture coordinates, colors, spatial coordinates and other parameters the Per-Vertex Operations stage can handle.

A.3.3 Per-Vertex Operations

This stage converts vertices in the 3D world into primitives that are relative to the screen. Geometrical transformations take place in this stage. When textures and lighting are used, material properties and light source positions are used to calculate the color values.

A.3.4 Primitive Assembly

Primitive assembly (in Figure A.9 displayed in one box with Per-Vertex Operations) covers clipping. If the perspective projection is activated this is also handled in this stage, along with viewport and depth operations. The results of this stage are complete geometric primitives.

A.3.5 Pixel Operations

Pixel data propagates through another path in the pipeline. The pixel data is read from memory, unpacked from the format it is in, scaled, biased and processed. The result is written into texture memory or sent to the rasterization stage directly.

A.3.6 Texture Assembly

The color of a pixel can be interpolated from the color of each corner of the triangle. However, for realism, often textures are used. These textures are provided as pixel-based images. Because the size of the texture on screen will seldom be equal to the native size of the texture image, multiple texture pixels might be mapped on one pixel on the screen or vice versa. Therefore the texture data will be visited often, thus should be stored on a fast accessible memory \[22\].

A.3.7 Rasterization

As mentioned before, the rasterization process, as the most important step of the OpenGL pipeline concerning this work, is discussed in Appendix \[13\].
A.3.8 Fragment Operations

Just before sending images to the framebuffer, in an OpenGL pipeline a series of operations might be performed on the fragments (pixels). These operations might include applying textures, combining primary and secondary colors, the scissor test, alpha test and stencil test. Only some of these operations are supported in this work.
In Appendix A, a general OpenGL pipeline is discussed. The last stage of this pipeline is the rasterizing stage. The input of a rasterizer consists of primitives. Those primitives are pixels, lines, triangles and other primitive shapes. This chapter focuses on triangles. Polygons like squares are often divided into triangles to ease the rasterization process.

The rasterizer performs one of the most computation intensive calculations in the OpenGL pipeline. During the rasterizing stage primitives, e.g., triangles are converted into picture elements, or pixels. Those pixels are saved in a framebuffer which can be displayed on a screen or sent to a printer.

### B.1 Primitives

For simplicity the basic element of the rasterization stage discussed in this appendix is the triangle. A triangle has the minimal number of coordinates needed for creating a two dimensional figure, namely three. A more general primitive would be a polygon. A polygon is a plane figure with at least three straight sides and angles, and typically five or more \[12\]. A polygon can be convex or non-convex. A convex polygon has no inner edges larger than 180°. An example of a non-convex polygon is given in Figure 3.2(a), for convenience repeated in Figure B.1(a). Figure B.1(b) shows how this polygon can be divided in three triangles. A triangle is always convex.

![Figure B.1: A polygon can be divided into triangles](image)

A triangle can be presented by three corners, called vertices. Each vertex has a number of properties, represented in Table B.1. The coordinates of x and y are mapped to the screen coordinates in the geometry stage of the OpenGL pipeline (Appendix A). The texture coordinates are used for mapping a rasterized image or pattern to a triangle, to create a more realistic or complex result. The 1/w parameter is introduced in
Appendix [A.2.2.1] and used for perspective correct hyperbolic interpolation of colors and textures.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x, y</td>
<td>screen coordinates</td>
</tr>
<tr>
<td>z</td>
<td>depth coordinates</td>
</tr>
<tr>
<td>r, g, b, a</td>
<td>color and alpha values</td>
</tr>
<tr>
<td>s, t</td>
<td>texture coordinates</td>
</tr>
<tr>
<td>1/w</td>
<td>homogeneous coordinate</td>
</tr>
</tbody>
</table>

Table B.1: Vertex parameters

B.2 Identification of pixels inside the primitives

In order to display the triangle on a 2D-screen with a finite resolution, the pixels need to be sampled. This process is called rasterization in computer graphics, because the pixels are fixed in a regular raster. Of all pixels in this raster, it has to be determined whether this pixel is within the primitive or outside the primitive. The two algorithms used for this process are Scanline Conversion and Linear Edge Testing [31]. In this Appendix we discuss the scanline conversion-algorithm.

In scanline conversion the pixel grid or raster is divided in lines, in this example horizontal lines. These lines are called scanlines. In Figure B.2 a grid of 15 × 6 pixels is shown with six horizontal scanlines. The first part of scanline conversions involves in selecting the scanlines that intersect with the primitive. Assume the primitive is a triangle defined by the vertices A, B and C or the coordinates x_A, y_A, x_B, y_B, x_C and y_C. The horizontal scanline is identified by its y-coordinate. An image of a resolution n × m has m horizontal scanlines. Let 0 < i ≤ m, then scanline i has a vertical coordinate y_i. If the triangle edge is extended to a line (that is, its direction is preserved but its start point and end point are discarded), each non-horizontal line intersects with each scanline at exactly one x-coordinate. The line AB is the line that intersects with the coordinates (x_A, y_A) and (x_B, y_B). To calculate the x-coordinate of the position where
B.3. PARAMETRIZATION OF THE PIXELS

The previous section explained how pixels inside a triangle can be identified. The next step is to parametrize each pixel. That is, the values of several parameters, including the
RGBA-values and the z-value have to be determined. These parameters are calculated based on the values of the parameters of the vertices defining the triangle. First, we define the set of parameters as \( P = (p_0, p_1, p_2, p_3, \ldots, p_n) \). Next, for each parameter \( p_i \) we want to calculate the value depending on the \( x \) and \( y \)-coordinates of the current pixel.

This can be done in a linear way, as demonstrated in Equation \( \text{B.2} \)

\[
p_i(x, y) = c_0 \cdot x + c_1 \cdot y + c_2
\]

Equation \( \text{B.2} \) needs the constants \( c_0, c_1 \) and \( c_2 \) to be known. These can be calculated from the parameters of the corners of the triangle. By replacing Equation \( \text{B.2} \) by its incremental form we find Equation \( \text{B.3} \)

\[
p_i(x, y) = p_{i,\text{init}} + \frac{dp_i}{dx} \cdot \Delta x + \frac{dp_i}{dy} \cdot \Delta y
\]

If \( p_{i,\text{init}} \) is known (or calculated using Equation \( \text{B.2} \)) calculating the parameters for the next pixel can be done with simple calculations. Of course the calculation of \( \frac{dp_i}{dx} \) and \( \frac{dp_i}{dy} \) involve more complex computations, but have to be done only once per triangle.

Interpolation can also be done using hyperbolic interpolation. This is useful because the \( x \) value in screen space does not vary linear with the distance from the viewport to the scene. Therefore instead of calculating \( p_i \) we calculate \( \frac{p_i}{w} \), where \( w \) is the homogeneous coordinate. Equation \( \text{B.2} \) now becomes Equation \( \text{B.4} \). The value for \( p_i \) can be found by \( p_i = \frac{p_i}{w} \cdot w \).

\[
\frac{p_i}{w}(x, y) = c_0 \cdot x + c_1 \cdot y + c_2
\]

The values for \( c_0, c_1 \) and \( c_2 \) are calculated by solving the linear equations \( \text{B.5} \), \( \text{B.6} \) and \( \text{B.7} \)

\[
c_0 \cdot x_A + c_1 \cdot y_A + c_2 = \frac{p_i A}{w}
\]

\[
c_0 \cdot x_B + c_1 \cdot y_B + c_2 = \frac{p_i B}{w}
\]

\[
c_0 \cdot x_C + c_1 \cdot y_C + c_2 = \frac{p_i C}{w}
\]

### B.4 Hidden Surface Removal

Some of the parameters calculated in the previous sections, especially the RGB-values, will eventually have to be stored in the framebuffer, if they are not obscured by other objects. Triangles in OpenGL might overlap. Only the triangle that is the closest to the eyepoint will be visible. Therefore the depth buffer or Z-buffer is used. One of the parameters mentioned above is the z-value. This value represents the relative depth of the fragment, or the distance of this fragment to the eyepoint.

Before rasterizing a new image the buffers are cleared. The RGB-buffers are usually set to black. The Z-buffer is set to the highest value. When the parameters of a fragment
are calculated its z-value is compared to the value of the corresponding fragment in the Z-buffer. When the z-value of the current fragment is below the saved value, this can be interpreted as being in front of the saved fragment. The new fragment parameters overwrites the saved parameters. However, when the z-value in the new fragment is higher than the saved value in the Z-buffer, the saved fragment is obscuring the new fragment. In that case the new fragment is discarded. This way only the pixels that are the closest to the eyepoint are saved. Therefore this technique is called Hidden Surface Removal [26].


[32] ZDNet.co.uk. Intel apologises to Apple, ARM for iPhone attack. 
http://www.zdnet.co.uk/2008/10/24/.
Index

Application stage, 87
Arithmetic Operations, 39
ARM Holdings, 15

BBOX, 11
Bounding Box test, 11
Buffers, 10

C-buffer, 10
C-HEAP, 33
C-heap, 28
Clipping, 91
CompOSe, 32
Configuration, 21

Display Lists, 93
Division, 39

Element, 23
Endianess, 23
Evaluators, 94
Execute, 25

FIFO, 28, 33
Floating point, 15
FPGA, 15
Fragment Operations, 95
Framebuffer, 10

geometry stage, 87
Graal, 18
GraalBench, 17
Hyperbolic interpolation, 100
Identification, 64
Intel, 15
Larrabee, 19
LET, 11
Linear Edge function Test, 11, 12
Linear interpolation, 100
Memory, 10
Mesa, 17
Microblaze, 15
OpenGL Pipeline, 93
Packed Element, 28
Parametrization, 64
Per-Vertex Operations, 94
Perspective, 90
pipeline, 93
Pixel Operations, 94
Polygons, 24
POSIX Thread, 27
Primitive Assembly, 94
primitives, 87
Profiler, 48
Projection, 90
Pthread, 27
Queen, 26
Rasterization stage, 87
RGBA, 24
S-buffer, 10
Scanline Conversion, 98
Section, 36
State, 24
Stream Processors, 16
Texture Assembly, 94
Tiles, 10
TKI, 18
TKI-file, 47
TKI-instructions, 21
Transformation matrix, 89
triangle mask, 64
Worker, 26
x86, 15
Xilinx, 15