Suitability of Tile-Based Rendering for Low-Power 3D Graphics Accelerators

Suitability of Tile-Based Rendering for Low-Power 3D Graphics Accelerators

---

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.dr.ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen

op maandag 29 oktober 2007 om 12:30 uur

door

Iosif ANTOCHI

inginer
Universitatea Politehnica Bucureşti
geboren te Boekarest, Roemenie

Dit proefschrift is goedgekeurd door de promotoren:

Prof.dr. S. Vassiliadis†
Prof.dr. K.G.W. Goossens

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter	Technische Universiteit Delft
Prof. dr. S. Vassiliadis†, promotor	Technische Universiteit Delft
Prof. dr. K.G.W. Goossens, promotor	Technische Universiteit Delft
Dr. B.H.H. Juurlink	Technische Universiteit Delft
Prof. dr. L.K. Nanver	Technische Universiteit Delft
Prof. dr. H.A.G. Wijshoff	Universiteit Leiden
Prof. dr. J. Takala	Tampere University of Technology
Dr. A. Pimentel	Universiteit van Amsterdam
Dr. K. Pulli	Nokia Research Center, Palo Alto

Dr. B.H.H. Juurlink heeft als begeleider in belangrijke mate aan de totstandkoming van het proefschrift bijgedragen.

ISBN: 978-90-807957-6-1

Keywords: 3D Graphics Accelerators, Tile-based Rendering, Low-Power Graphics Architectures

Printed in the Netherlands

This dissertation is dedicated to Claudia
and my family,
for all their understanding and support over the years.

Suitability of Tile-Based Rendering for Low-Power 3D Graphics Accelerators

Iosif ANTOCHI

Abstract

In this dissertation, we address low-power high performance 3D graphics accelerator architectures. The purpose of these accelerators is to relieve the burden of graphical computations from the main processor and also to achieve a better energy efficiency than can be achieved by executing these computations on the main processor. Since external data traffic is a major source of power consumption and because usually the rasterization stage of the 3D graphics pipeline requires the highest amount of data traffic, in this dissertation we especially focus on this stage of the graphics pipeline. Another reason for focusing on the rasterization stage is that it requires more processing power than the other stages because the operations are pixel-based. A promising technique to reduce the external data traffic in the rasterization stage of the graphics pipeline is tile-based rendering. This technique decomposes a scene into tiles and renders the tiles one by one. This allows the color components and z values of one tile to be stored in small, on-chip buffers, so that only the pixels visible in the final scene need to be stored in the external framebuffer. Tile-based accelerators, however, require large scene buffers to store the primitives to be rendered. While there have been studies related to the tile-based rendering paradigm for high performance systems, we are specifically discussing the suitability of tile-based 3D graphics accelerators for low-power devices. In order to evaluate various low-power 3D graphics architectures we first present GraalBench, a set of 3D graphics workloads representative for contemporary and emerging mobile devices. Furthermore, we propose several scene and state management algorithms for tile-based renderers. Thereafter, we analyze the performance of tile-based renderers compared to that of traditional renderers and we also determine the influence of the tile size on the amount of the data-traffic required for the rasterization stage of a tile-based renderer. In order to reduce even more the data traffic between the main memory and graphics accelerators, and to exploit the high temporal and spatial locality of texture accesses, we have also investigated several cache structures. Our results show that the proposed algorithms for tile-based renderers can effectively decrease the data traffic and computational requirements for the rasterization stage of the 3D graphics pipeline.

Acknowledgments

During the time that I was performing the research described in this dissertation, I came across many people who have supported and assisted me without whom it would have been much harder to produce this dissertation.

First of all, I would like to thank my supervisor Ben Juurlink and my promotors Stamatis Vassiliadis and Kees Goossens for their endless support and guidance. They succeeded to provide me not only research knowledge, but also a better understanding of real life. Although Stamatis is no longer among us, his presence lives on through each CE member.

Furthermore, I would like to thank my officemates, Dan and Pepijn for our inspiring technical and also less technical discussions. I would like to thank Elena for her encouragement and positive thinking that helped me over the years.

I had also found it very enjoyable to work with and to talk to every member of the Computer Engineering Laboratory. I am in debt to the "older" generation (Pyrrhos, Casper, Stephan) for introducing me into the geeky spirit of Computer Engineering, and also to the newer generation which kept me up to date with various interesting topics.

I am also indebted to the small Romanian community from the Netherlands that helped me overcome my homesickness and also for being with me when I stumbled across various problems.

Finally, special thanks go to Claudia for her understanding and support over the years.

I. Antochi Delft, The Netherlands, 2007

Contents

Chapter 1

Introduction

In recent years, mobile computing devices have been used for a broader spectrum of applications than mobile telephony or personal digital assistance. Several companies [4, 3] expect that 3D graphics applications will become an important workload of wireless devices. For example, according to [11], the number of users of interactive 3D graphics applications (in particular games) is expected to increase drastically in the future: it is predicted that the global wireless games market will grow to 4 billion dollars in 2006. Because current wireless devices do not have sufficient computational power to support 3D graphics in real time and because present accelerators consume too much power, companies [4, 5, 1, 3, 2] and universities [7, 6] have started to develop low-power 3D graphics accelerators.

The main goal of this dissertation is the design of a low-power 2D and 3D graphics accelerator for mobile terminals equipped with an ARM CPU core (ARM architecture variant v5T). The purpose of this accelerator is to relieve the burden of graphical computations from the ARM CPU core. The accelerator concerns only the back-end stage of the graphics pipeline, more specifically, the rasterization stage.

Tile-based rendering (also called chunk rendering or bucket rendering) is a promising technique for low-power, 3D graphics platforms. This technique decomposes a scene into smaller regions called tiles and renders the tiles independently. In a high-performance graphics accelerator these tiles can be rendered in parallel, but in a low-power accelerator we can render the tiles one by one. The advantage of this scheme from a power perspective is that each tile can be rendered without accessing the off-chip memory except when the tile has been completely rendered. Since off-chip memory accesses are a

major source of power consumption and often dissipate more energy than the datapaths and the control units [16, 31], reducing them decreases the power consumption. Tile-based accelerators, however, require large *scene buffers* to store the primitives to be rendered. Therefore, this dissertation investigates the suitability of tile-based rendering for low-power 3D graphics accelerators.

This chapter is organized as follows. In Section 1.1 we give a short overview of 3D graphics accelerators concepts, discuss the organization of a conventional and a tile-based renderer, and describe design limitations for current low power graphics accelerators. In Section 1.2 related work regarding traditional and tile-based rendering methods is described. Section 1.3 presents the open questions and our contributions. Finally, Section 1.4 completes the chapter with an overview of the dissertation.

1.1 Background

This section starts with a brief overview of the terms used in computer graphics. Further on, a description of conventional and tile-based rendering architectures is provided. Finally, factors limiting the design space for low-power graphics architectures are discussed.

1.1.1 The 3D Graphics Pipeline

In this section we provide the background information necessary to understand the remaining sections of this chapter. More details are given in Chapter 2. **3D graphics** refers to systems and methods used to create and manipulate a modeled "world" or scene on a computer and then to display the world on the 2D computer screen in a realistic way. The world is typically constructed from objects made up from meshes of adjoining triangles or polygons each defined by its vertices. Each vertex has a number of properties including its position in 3D space (x, y, z) and color. Each polygon additionally has some global properties such as texture (image pattern filling the polygon).

Figure 1.1(a) depicts a typical 3D game scene as it is presented to the user. Figure 1.1(b) shows how the scene is made from meshes of polygons. For instance, the statistics located in the lower part of the image are rendered as meshes of triangles; each individual digit being represented as a group of two textured triangles.

The **3D graphics pipeline** consists of three main stages: the application stage,

(a) Scene displayed on-screen (b) Scene components

Figure 1.1: A scene from a typical 3D game (Quake3 [43])

the geometry stage, and the rasterization stage. In the application stage a high-level description of the world is generated. All interaction between the user and the computer is implemented in this stage.

The geometry stage is responsible for the majority of the operations concerning polygons and vertices. It is also organized as pipeline and consists of the following stages: model and view transform, lighting and shading, projection, clipping, and screen mapping. Model transform consists of transformations such as scaling and rotation. View transform is the transformation from the world coordinate system to a coordinate system relative to the location from which the scene is observed (the *camera* or *eye* space). In the lighting and shading stage calculations are performed that simulate the effect of different lights on objects in the scene. In the projection stage the coordinates of the primitives are transformed into a canonical view volume. In the clipping stage the primitives that are partially inside the view volume are clipped so that only the part that is inside the view volume is sent to the following stages. Finally, in the screen mapping stage the graphics primitives are scaled to obtain the window (screen space) coordinates.

The most time-consuming and power-dissipating stage is the rasterization or rendering stage. In this stage the primitives (commonly triangles) are first scan converted to fragments (series of 2D screen space coordinate, and other attributes such as color and depth). Thereafter, 2D images are mapped onto fragments to create the impression of real-life 3D objects. This technique is called *texture mapping* and is one of the stages that requires the most memory bandwidth. The simplest technique, called point sampling, uses a single texture element (texel) from the texture map. More advanced techniques, such as trilinear MIP-mapping, interpolate among many texels to more accurately

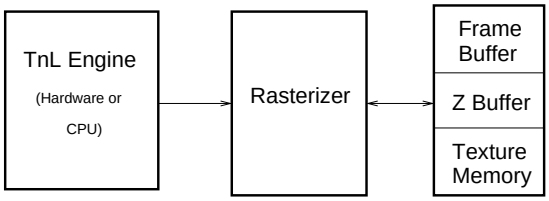


Figure 1.2: Overview of a traditional renderer.

estimate the texture for the fragment. After texturing, several operations are performed on each fragment to eliminate the fragments that are not visible but were generated by previous stages. One of these operations is the depth test which determines if a fragment is obscured by other fragments. Commonly employed is the z buffer algorithm, which stores the depth (z) value of each fragment in a buffer.

1.1.2 Tile-Based Versus Conventional Rendering

The organization of a conventional rasterizer is depicted in Figure 1.2. The *Transform and Lighting* (*TnL*) *Engine* corresponds to the previously described geometry stage. On modern PC-class graphics accelerators the TnL Engine is implemented on-chip, but for low-power, low-cost accelerators for mobile phones the TnL computations can also be performed by the host CPU. After the vertices have been processed by the TnL Engine, they are sent to the rasterizer.

The rasterizer writes rendered pixels to the framebuffer and stores the depth values of the fragments closest to the viewpoint in the z buffer. Because these buffers are large, they can in general not be placed in the on-chip memory of a low-cost accelerator. Furthermore, textures also need to be stored in the off-chip memory.

In a tile-based renderer, each scene is decomposed into regions called *tiles* which can be rendered independently. This allows the z values and color components of the fragments corresponding to one tile to be stored in small, on-chip buffers, whose size (in pixels) is equal to the tile size. This implies that only the pixels visible in the final scene need to be written to the external framebuffer. However, as previously mentioned, tile-based rendering requires that the triangles are first sorted into bins that correspond to the tiles. Moreover, since a triangle might overlap more than one tile, it may have to be sent to the rasterizer several times.

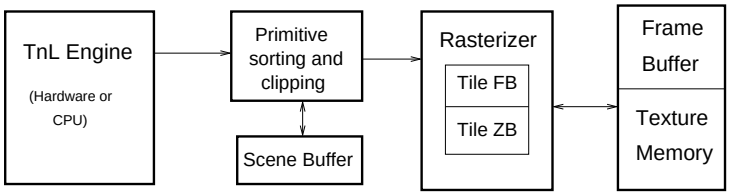


Figure 1.3: Overview of a tile-based renderer.

The organization of a basic tile-based renderer is depicted in Figure 1.3. First, the primitives are sorted according to the tile partitioning and stored in a so-called scene buffer. This may be performed by the CPU or the rasterizer. After the primitives have been sorted, the tiles are rendered one by one. For each tile, the primitives that overlap the current tile are sent to the rasterizer. First, if the application does not clear the frame or z buffers when it starts to render a new frame, the data associated with the current tile is fetched from the corresponding external frame or z buffers to the local tile buffers (Tile FB and Tile ZB). Thereafter, the rasterizer starts rendering the primitives into the tile buffers. After all primitives that overlap the current tile have been rendered, the content of the tile buffers is written to the external framebuffer.

1.1.3 Design Limitations and Performance Requirements

While graphics accelerators are intended to improve performance by off-loading graphics processing from the CPU, the resources available to implement such accelerators on mobile devices are rather limited compared to other systems. This section describes the most important design limiting factors and also expected performance from a graphics accelerator targeted at current mobile devices.

Due to the fact that mobile devices are powered by batteries, an important limitation for a mobile device accelerator is power consumption. For current mobile accelerator devices, power consumption levels around 0.4-3mW/MHz [11, 27] are considered acceptable. In some cases, due to the small size of mobile devices, additional restrictions related to heat dissipation can limit the power budget for an accelerator design.

The reduced size of mobile devices and the particular semiconductor technology used to implement the chip also dictates the number of gates available to implement the accelerator. Current gate budgets available for graphics accelerators for mobile devices are around 200k to 500k gates [11, 27].

Another set of factors that can limit the accelerator design is given based on the mobile system architecture. Low-power CPUs are often coupled with narrow buses and a reduced amount of memory. To make the overall picture even worse, such CPUs might not implement floating-point instructions or multimedia instruction set extensions. While the external buses on some systems might theoretically provide sufficient bandwidth for a graphics accelerator, they might be shared with other functional units which decreases the overall cost but also limits the bandwidth available to the accelerator. Moreover, since the data throughput of various functional units coupled on a shared bus might be unpredictable, a graphics accelerator should be able to cope with the remaining bandwidth while still providing acceptable frame rates.

To further reduce power consumption, most mobile devices are equipped with small (e.g., 3 in.), low resolution (e.g., 320 $\times$ 240 pixels) screens. Due to the small size, such screens are usually used at a shorter distance to the viewer than normal size displays. By also considering the low resolution, the projected pixel size on the viewer's eye is larger for mobile devices than for conventional PC. This implies that a mobile graphics accelerator should render each pixel with higher quality levels than a normal PC, while the reduced size of the memory limits even further the number of colors that can be stored for each pixel.

Last but not least, the system cost and the expected market size can be a significantly restrictive design factor.

1.2 Related Work

In this section we discuss related work and highlight the differences with our work.

Tile-based architectures were initially proposed for high-performance, parallel renderers [32, 52, 40]. Since tiles cover non-overlapping parts of a scene, the triangles that intersect with these tiles can be rendered in parallel. In such architectures it is important to balance the load on the parallel renderers [53]. However, we are focusing on low-power architectures for which the tiles are usually rendered sequentially one by one.

While not designed for low-power, the PixelPlanes5 multicomputer described by Fuchs et at. [32] has pioneered the area of advanced parallel rendering by using simple per pixel rendering processors. Each array of 128×128 pixel processors was grouped as a rendering unit, and the entire system contained

a number of 16 rendering units. PixelPlane5 also included 32 graphics processors (GP) that could be dynamically assigned to any 128×128 rendering region. Each GP processes the data for a 128×128 region by placing render commands for each graphics primitive overlapping the region into a bin associated with each region. The render commands are used by the rendering units to compute intermediate results. When a final rendering request is received by a rendering unit, final pixel values are computed and stored to the frame buffer. Although, such massively parallel systems are impractical for current mobile devices, they opened the opportunity of using tile-based designs for low-power sequential rendering.

The Kyro [71] accelerator, developed by PowerVR, is one of the first low-power, tile-based graphics accelerators intended for PC markets. To achieve low-power, Kyro uses sequential tile-based rendering to render tiles one-by-one using small on-chip memories. Since traffic to small on-chip memories requires far less power than traffic to large off-chip memories, this technique shows significant potential for low power designs. In Kyro, graphics primitives sorting into bins according to rendering regions (tiles) is performed on the CPU and further sent for processing to the rendering core. For efficiency, since each tile is rendered sequentially, the geometry send by the CPU needs to be stored in a potentially large scene buffer and processed in a tile-based order. There is no available information on the efficiency of the algorithms used to implement primitive sorting. Based on a similar design, the MBX [72] core was further developed and targeted towards low-power mobile devices.

Torborg and Kajiya [73] described a 3D graphics hardware architecture named Talisman. There are four major concepts utilized in Talisman: composited image layers, image compression, chunking, and multi-pass rendering. Generally, there is an independent image layer for each non-interpenetrating object in the scene. This allows to update each object independently and, therefore, to optimize object update rates based on scene properties. Compression is applied to images and textures, using an algorithm similar to JPEG referred to as TREC. The decompressor, however, requires approximately 26% of the total area. Chunking is identical to tiling. Finally, multi-pass rendering is used to render reflections and shadows. Talisman is, however, targeted at PCs and it is unclear if it can be used for mobile terminals.

Because only 20% to 83% of the triangles of the original 3D object models are visible in the final scene, Liang and Jen [49] proposed to employ two parallel pipelines so that only visible triangles are lighted, shaded, and textured. Their proposal, however, requires additional, off-chip I and Tdbs buffers, which

might increase the external memory bandwidth requirements. Moreover, the I-Buffer holds only one index for each pixel, which would require special handling of (semi)transparent objects blending.

Kameyama et al. [44] described a 3D graphics LSI core for mobile phones named Z3D. The geometry engine, the rendering engine, and the pixel engine are all implemented on the Z3D core. Furthermore, the display list buffer, the texture memory, and the frame and z buffers are also implemented on this core. Since tiling is not used in Z3D, this limits the applications to character animations and simple games. To reduce power consumption, Z3D uses clock gating and a so-called step mode. Clock gating is used to turn off clock supply to logic blocks that do not need to work. For example, when the 2D engine is working, the clock supply to the 3D pipeline is turned off. In the step mode, the geometry engine, the set up engine, and a third block that includes the raster, texture and pixel engines run in sequence in a round robin fashion. First the geometry engine is supplied with a clock and the clock supply to the set up engine and the third block is turned off. Then, the set up engine is the only block supplied with a clock and thereafter the third block. This process is repeated until all vertex data has been processed. The step mode, however, requires intermediate buffering.

McCormack et al. [50] presented a single-chip 3D graphics accelerator named Neon. In order to improve the memory bandwidth and to reduce power on the low pin count memory interface, the design of Neon started from the memory system upwards. Despite the fact that there was no special memory used, eight 32-bit independent memory controllers were implemented to provide sufficient bandwidth for the memory request queues. Neon also featured a unified memory system for storing textures, and color, depth, and other off-screen buffers. This design allows for better dynamic reallocation of memory bandwidth. Furthermore, due to low cost and gate budget limitations, integer based operations with similar implementations shared hardware logic while floating point operations remained to be performed on the CPU. With the added cost of three additional 600-bit save states (later redesigns proved that only one save state would be sufficient), Neon also implements texture chunking and interleaving to better exploit frame buffer and texture access patterns.

Recently, Akenine-Möller and Ström [9] also proposed a hardware rasterization architecture for mobile phones. They also focus on saving memory bandwidth and propose a system which features an inexpensive multisampling scheme, a texture minification system, and a simple, scanline-based culling scheme. Compared to nearest-neighbor sampling, their system requires

slightly more memory bandwidth but achieves a better quality. Compared to trilinear MIP-mapping, their system reduces the required memory bandwidth by 53% on average. They also briefly mention tile-based rendering but remark that it requires a significant amount of memory bandwidth because geometry sorting is needed before rendering can start. However, no experimental data is provided.

1.3 Open Questions and Contributions

The following open questions can be posed with respect to low-power mobile 3D graphics accelerators in general and tile-based rendering architectures in particular.

- **What 3D graphics applications are relevant for low-power mobile devices?**

  In order to determine the performance and energy improvement that can be achieved by introducing a new architecture or algorithm, representative workloads are needed. Although there are several benchmarks for desktop 3D graphics accelerators, these are usually not suitable for the low-power 3D graphics application area. Moreover, 3D graphic applications that are expected to be used on mobile devices (e.g., 3D Games) have an interactive behavior. They are therefore unreproducible from one run to another, which implies that they are not directly suitable to be used as benchmarks. As shown in Chapter 3, only a few available applications can be considered representative for mobile devices.

- **How much external data traffic is generated by a tile-based renderer compared to a traditional renderer?**

  Tile-based renderers, as indicated in Section 1.1.2, can generate less traffic to external color buffer and texture memories than traditional renderers since they render the scene on-chip and only transfer the final result to external memories. However, when compared to traditional renderers, tile-based renderers also require data traffic from the scene buffer to the renderer core. As shown in Chapter 4, for workloads with a high overlap factor and low overdraw, traditional rendering requires less data traffic than tile-based rendering, while for workloads with a low overlap factor and high overdraw, tile-based rendering is more efficient than traditional rendering.

- **What algorithms can be used to sort primitives to tiles for tile-based rendering architectures?**

  In a sequential tile-based rendering approach, a tile can be completely rendered only when all primitives required to render the entire scene are available, as any primitive can potentially overlap/affect any tile. If there is not sufficient memory to store all the primitives for one scene, then the scene must be rendered in multiple iterations per tile. The computational complexity of sorting the primitives according to their overlap with the tiles, as shown in Chapter 5, is influenced by the amount of available memory. In general, the more memory is available, the less computations have to be performed in order to sort the primitives. As shown in this dissertation, a trade-off between computational complexity and the size of the required memory can be used also.

- **What performance gain can be expected from primitive sorting algorithms by exploiting tile characteristics (e.g., position)?**

  Since a tile-based rendering system requires sorting of primitives to tiles, and a tile might overlap several tiles, there are significant chances that computations used to determine if a primitive overlaps a tile might be (partially) reused to determine if the same primitive overlaps other tiles. Moreover, since the size, shape, and position of the tiles can be predetermined, additional optimizations might be performed.

  Consider for example, the tile located in the upper left corner of a scene. Based on the position of the tile, it is expected that most primitives will be situated to the right or below the current tile. Thus, by checking if a primitive is located below or to the right of the tile instead of checking the left and upward directions, can result in a significant reduction of the number of computations required to determine if the tile and the primitive overlap as shown in Chapter 5.

- **How much can the state change information that is sent to the tile-based 3D graphics accelerator be reduced?**

  While in traditional architectures state change information (e.g., enable or disable depth testing) is usually negligible, in tile-based architectures the state change information can potentially require duplication for each tile and as a consequence, the state change information traffic can significantly increase. Thus, for tile-based architectures state change data traffic might not be negligible.

Moreover, since primitives not overlapping a tile are not usually sent to a tile, the state change data traffic that would have been generated to prepare rendering for such primitives might be eliminated. However, compacting and eliminating redundant state, often requires additional CPU processing power and/or memory to keep track of state changes. On the other hand, memory can be saved by not storing the redundant state information, and moreover, the external state change data traffic can be reduced. In this dissertation we show what state change reduction can we expect and how to overcome several tile-based inherent limitations.

- **What other techniques can be used to further reduce external data traffic (e.g., texture caches) and what improvements can be expected?**

  Since texture accesses patterns are more regular than general memory accesses, texture caches could be employed to reduce external data traffic. However, due to restricted on-chip memory size limitations, only a relatively small texture cache would be affordable. Moreover, large caches could potentially consume significantly more energy while not significantly reducing the data traffic as compared to smaller caches.

  As shown in Chapter 4, due to high spatial and temporal locality of texture accesses patterns even small caches can reduce the external data traffic significantly.

1.4 Thesis Overview

This dissertation is organized as follows. In Chapter 2 we provide a survey of the 3D Graphics Pipeline with a particular focus on texture mapping. In order to understand the possible performance and quality trade-offs, several texture mapping algorithms are described and analyzed.

For instance, using only the linear filter without MIP-mapping can significantly reduce the quality of filtering when the ratio between covered texels and corresponding pixels is larger than one. Therefore, MIP-mapping is necessary in these situations. Moreover, when the shape of the projection in texture space of a screen region is not square then anisotropic filtering should be used. However, anisotropic filtering is significantly slower than MIP-mapping. The choice of which method to implement in hardware depends on many parameters such as rendering quality, rendering speed, power consumption, and number of gates available to implement a specific feature. Readers familiar

with the 3D graphics pipeline can skip this chapter.

In Chapter 3 we provide evidence indicating that 3D benchmarks employed for desktop computers are not suitable for mobile environments. Consequently, we present GraalBench, a set of 3D graphics workloads representative for contemporary and emerging mobile devices. In addition, we present detailed simulation results for a typical rasterization pipeline. The results show that the proposed benchmarks use only a part of the resources offered by current 3D graphics libraries. In this chapter we also discuss the architectural implications of the obtained results for hardware implementations.

In Chapter 4 a comparison of the total amount of external data traffic required by traditional and tile-based renderers is presented. A tile size yielding the best trade-off between the amount of on-chip memory and the amount of external data traffic is determined. We also show that tile-based rendering reduces the total amount of external traffic due to the considerable data traffic reduction between the accelerator and the off-chip memory while maintaining an acceptable increase in data traffic between the CPU and the renderer. We show that for workloads with a high overlap factor and low overdraw, the traditional rendering can still outperform the tile-based rendering, while for workloads with a low overlap factor and high overdraw, the tile-based rendering is more suitable than traditional rendering.

In Chapter 5 several algorithms are presented for sorting the primitives into bins and evaluate their computational complexity and memory requirements. In addition, we describe and evaluate several tests for determining if a triangle and a tile overlap. While the number of primitives sent to the accelerator depends only on the triangle to tile overlap test used, the memory required to sort and store the primitives before being sent to the accelerator as well as the required computational power depends largely on the algorithm employed.

Also in Chapter 5 a dynamic bounding box test is presented. We show that the efficiency of the bounding box test can be improved significantly by adaptively varying the order in which the comparisons are performed depending on the position of the current tile. Experimental results obtained using the GraalBench workloads show that the dynamic bounding box test reduces the average number of comparisons per primitive by 26% on average compared to the best performing static version in which the order of the comparisons is fixed.

In Chapter 6 several state management algorithms for tile-based renderers are presented. While in traditional (non tile-based) rendering the state information traffic is negligible compared to the traffic generated by the primitives, in tile-based rendering architectures, the required processing power and gener-

ated traffic can increase significantly since the state information might need to be duplicated in multiple streams. In this chapter we analyze how much the amount of state change traffic to the accelerator can be decreased by sending an optimized state change stream to the accelerator.

In Chapter 7 we propose and evaluate using a small filter cache between the graphics accelerator and the texture memory in portable graphics devices which cannot afford large caches. In this chapter we determine the cache and block sizes which are most efficient for texture caching w.r.t. the Energy-Delay metric. A comparison of the performance of the proposed caches and a conventional cache is also presented. We also present evidence that a widely-used benchmark exhibits anomalous behavior.

Finally, Chapter 8 concludes the dissertation by summarizing our investigations, discussing our main contributions, and proposing further research areas.

Chapter 2

Overview of A 3D Graphics Rendering Pipeline

To illustrate a general but not necessarily unique method to generate 3D worlds on 2D devices, we will describe one of the most accepted models to transform 3D objects and render them using the pipeline concept which is very well suited for our purpose. Implementing graphics operations as a pipeline has the benefit that the border between software and hardware implementation is not fixed before hand, so depending on the technology available and the production costs, it can be decided which pipeline stage should be implemented in hardware and which in software.

This chapter is organized as follows. Section 2.1 describes a graphics rendering pipeline main stages. In Section 2.2 we briefly describe the application stage of a graphics pipeline. A description of the geometry stage of the pipeline is given in Section 2.3. A detailed description of the rasterization stage of the pipeline is presented in Section 2.4. Finally, conclusions regarding operations suitable for hardware acceleration are given in Section 2.5.

2.1 The Graphics Rendering Pipeline

The main function of a graphics pipeline is to generate (render) three dimensional objects on a two dimensional device while taking into consideration factors such as light sources, lighting models, textures, and other information that would have an impact on rendering quality.

The rendering pipeline can be seen as the underlying tool for achieving real-

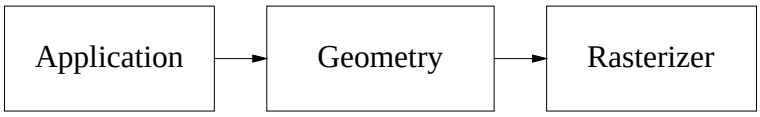


Figure 2.1: The basic construction of a graphics rendering pipeline.

time rendering. As any pipeline the rendering pipeline has a number of stages that will pass information from one to the other. The link between each stage and its predecessor or successor stage is very well defined and each stage could be seen at first as a "black box" that will be defined later. A model of a basic 3D graphics rendering pipeline is depicted in Figure 2.1.

The main conceptual stages of the graphics pipeline are:

- The application stage - In this stage, a high level description of a 3D world is generated.
- The geometry stage - In this stage, the transform and lighting computations are performed.
- The rasterizer stage - This stage transforms the rendering primitives (points, lines, and triangles) into pixels.

In the following sections, we will briefly describe each of these stages.

2.2 The Application Stage

In the application stage, the user has full control over the resources, meaning that this level is mostly implemented in software and user can change the implementation in order to change the performance. As an example: the user can choose how smooth a sphere would be rendered with respect to the number of triangles generated for that sphere.

Usually in this stage is implemented almost all the user-computer interaction such as keyboard or other input device interaction that can modify the rendered scene. Also in this level, are received and processed events that can affect the objects in the virtual world or even only the viewing parameters of the rendered world.

Figure 2.2: The geometry stage subdivided into a pipeline of functional stages.

Another important task at this level is collision detection. After a collision is detected between some objects certain treatment should occur in order to reflect the new state of the objects including eventually a response to the user (force feedback devices).

Other tasks that could help the next stages are also implemented here. Some precomputations could be performed before hand if we know, for instance, that we use a particular feature and perform the calculus for the particular model instead of performing similar computations later but relaying on a more general model provided by the following stages. For example, some transformations such as scaling, and rotation could be implemented as 3×3 matrix multiplications. In order to support translation or perspective transformations, 4×4 matrix multiplications are required. Hardware implementations of the geometry level, usually provide a unified architecture for the transformation operations, thus implementing only 4×4 matrix multiplications. However, if for instance, a projection operation is not required while transforming a scene from its original coordinate space to the display coordinate space, then some computations can be saved by precomputing the transformation matrices using only 3×3 matrix multiplications.

The information that is generated by this stage consists of rendering primitives such as points, lines, and triangles. Not necessarily all the primitives generated from this stage will belong to the final rendered image. Some of them might be out of the viewing frustum or obscured by other objects.

2.3 The Geometry Stage

The geometry stage is responsible for the majority of the operations concerning polygons (per-polygon operations) or vertices (per-vertex operations). The geometry stage itself can be treated as a pipeline (see Figure 2.2).

The order of operations varies across implementations and the delimitation between the operations is not sharp and it depends upon the control sequence (algorithms) used. For instance a clipping operation can be performed before

or after a projection operation, albeit using different coordinate spaces.

We will briefly focus on each of the blocks that form this internal pipeline.

Model and View Transform

At this level each of the primitives generated from the previous stage is transformed according to different coordinate spaces.

At first, upon each vertex of a primitive can be applied transformations in its own coordinate space which is called *model space*. Transformations that can be applied here are:

- Scaling - The size of the object is modified.
- Rotations - The orientation of the object is affected.
- Translations - The position of the object is changed.
- Mirroring effects - The object is mirrored against a reference plane, in a 3D space, or against a line in 2D space, or against a point in 1D space.

For some operations, the order in which they are applied is important. For instance, applying first a rotation and afterwards a translation on a vertex can yield a different result than the result obtained if the order of the operations is changed.

After an object has been transformed in its own model space, it can be placed in the general scene *world* space. By looking at objects from an object oriented perspective, each copy of an object that is placed in the general scene is called an instance of the respective object. Thus, actually, the graphics pipeline is not processing objects, but instances of objects. The idea behind this naming convention is that instances of objects contain general characteristics of their corresponding objects, but also some specific, per instance, details.

So far, the instances of the objects, represented by their primitives, were transformed in their model space and placed in the world space. Hereafter, another set of transformations similar with the ones described above is applied, but the difference is that each transformation is applied for the whole object instance (on all its primitives). For instance, a cat will not have only its tail bent (local transformation), instead the whole cat is positioned upside down, or moved to another location.

The final operation of this level is to apply a *view transform*. That is, a transformation from the world coordinate system to a coordinate system relative to

the location from which the scene is observed. The latter coordinate system is also referred to as *camera* space or *eye* space. This space is usually defined by the position (coordinates) where the viewer is located, the direction where the viewer is looking at, and a viewing frustum.

Since not all the primitives will be inside the viewing frustum some of them will be effectively removed or have parts removed. Actually this step is not performed here but we prepare for the clipping step by applying the view transform which transforms the objects coordinates in the eye space.

Most of the calculations performed at this level can be described using matrix operations. To perform scaling and rotation operations in a 3D coordinate system 3×3 matrices are sufficient. However, to integrate also translation and other operations, homogeneous coordinates need be used, and the transformation matrices sizes increases to 4×4.

In a homogeneous system, a vertex $V(x, y, z)$ is represented as $V(X, Y, Z, w)$ for any scale factor $w \neq 0$. The three dimensional Cartesian coordinate representation is:

$$x = \frac{X}{w} \tag{2.1}$$

$$y = \frac{Y}{w} \tag{2.2}$$

$$z = \frac{Z}{w} \tag{2.3}$$

The homogeneous representation allows a more efficient solution of applying several geometry transformations by first multiplying all the transformation matrices and storing the result into a single matrix and then for each vertex to perform only the multiplication with the final matrix.

Lighting and Shading

Lighting is the term designated to specify the interaction between material and light sources, but not only the material is important, also the geometry of the object plays an important role.

In this stage the lighting information belonging to each primitive is taken into account. For the purpose of realism different types of lights can be added to the virtual world depending of the medium intended to be created or simulated.

Since usually computing accurate lighting equations requires significant computational power, several approximations (shading models) can be computed

instead. The shading models can be applied to each primitive such as a triangle or to each vertex or even to each point that belongs to the respective primitive (triangle). Wide spread shading models are:

- Flat shading - The whole surface of a triangle (polygon) has the same color so the color is calculated only in one point for the whole triangle.
- Gouraud shading - This model calculates the color in each vertex of the polygon by averaging the normals of the triangles (polygons) connected to each vertex, and applying the lighting equations on this vertices and then interpolation among this values is performed.
- Phong shading - Interpolates the normals vectors and applies the lighting equation at each pixel.

Projection

At this stage the coordinates of the primitives are transformed into a canonical view volume. There are two widely used methods for this transformation.

- Orthographic projection - Consists of translation and scaling. In this case parallel lines remain parallel after projection.
- Perspective projection - A more complex projection that mimics the way we see things. The farther is the object from the observer, the smaller it appears after projection.

Both projection models can be computed either using general 4x4 matrices or optimized computations.

Clipping

In order to draw only the primitives that are contained fully or partially inside the view volume clipping should be performed. The primitives that are partially inside the view volume are clipped so that only the part that is inside the view volume remains to be sent to the following stages.

The clipping process can generate additional primitives for each clipped primitive. For instance, the remaining part of a clipped triangle, might only be represented by several triangles instead of one triangle.

Viewport (Screen Space)

At this level primitives are translated and scaled to obtain the window (screen space) coordinates. Although, the screen space is usually a two dimensional space, for each transformed vertex of a primitive, a depth coordinate (also denoted as z in some references) is required to be kept. The depth coordinate represents the distance from the viewer to the vertex and it is used to determine which primitive is the closest (visible) to the viewer for a given screen space position.

The viewport stage is the last stage of the geometry pipeline. Following the geometry pipeline is the rasterization stage which is described in the following section.

2.4 The Rasterization Stage

The rasterization stage is responsible with mapping each primitive such as a triangle into components that can be individually represented in a bi-dimensional discrete screen space. Such components are also named pixels (picture elements). Each pixel corresponds to a location in the screen space and has associated a color that will be visible on the screen. The process of mapping a primitive into pixel elements is also called rasterization.

It can be also noted that, the rasterization process depends not only on the primitive itself, but also on some additional information, named state information. The state information is sent independently from the primitives. For instance, a triangle can be rasterized as a set of three points, a set of three edges (only the outline of the triangle), or as a surface (filled triangle).

The rasterization stage is usually also organized as a pipeline. Various implementations of a rasterizer pipeline can have hardware optimized stages. In order to present a general and widely used model, we opted to describe an OpenGL [64] rasterization pipeline. A functional organization of a OpenGL rasterization pipeline in presented in Figure 2.3.

Conceptually, the rasterization pipeline can be divided into three stages. The first stage consists of scan-converting the primitives into pixels. This stage comprises the *Triangle Setup*, *Edge Walk*, and *Span Interpolation*. The second stage consists of the *Texture Mapping* unit responsible to fetch required texels from texture images. The third stage, contains the per fragment operations and consists of the remaining units of the rasterization pipeline. Detailed

description of each block is provided in the following sections.

2.4.1 Triangle Setup

Considering a complete software implementation of a graphics library as a reference, we can distinguish increasing degrees of acceleration of the graphics computations by moving their execution from the host processor to a dedicated graphics processing device. For 3D graphics, the majority of graphics architectures include a rasterizer to which the 3D vertex coordinates in image space, associated color values, and for some architectures also texture coordinates, are sent.

In the following sections we describe several differences that can be encountered while rasterizing various primitive types such as points, lines, and triangles.

Point Rasterization

Any point that has to be rasterized requires the following information: its screen coordinates, a depth value relative to the camera, and color information or texture coordinates. A point can have a size by specifying the width or the diameter. The effective mapping of a point to pixels depends not only on the characteristics of the point but also by the enhancements applied on the resulting pixels. For instance, if anti-aliasing is enabled it is possible that a larger number of pixels is affected by transforming the same point compared to the normal rendered point.

Line Rasterization

Line rasterization can be considered as an extension of the point rasterization case. A line has 2 sets of information, one for each vertex, and it can also have a width associated or a pattern to be filled with. Anti-aliasing can be applied to enhance the visual result on a raster device. To actually draw the line Bresenham's algorithm [30][29] or similar variations based upon it can be used.

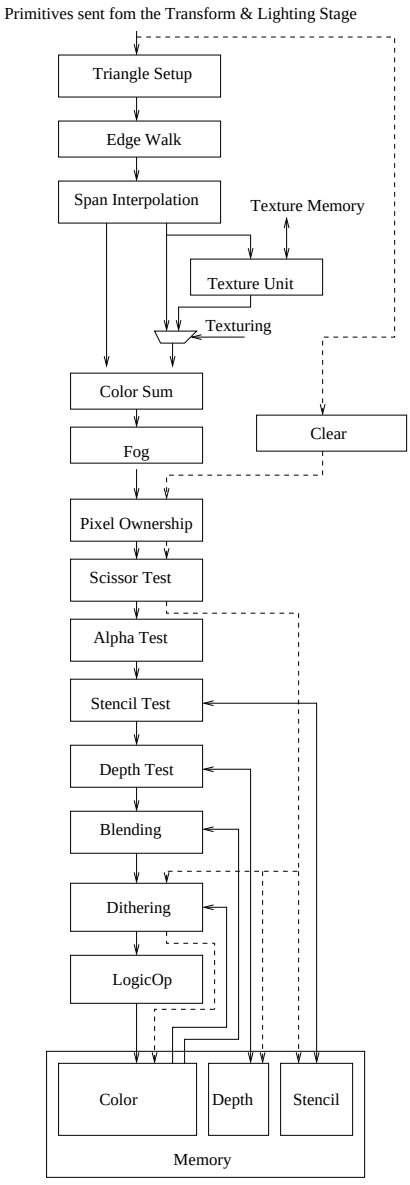


Figure 2.3: Graphics pipeline for rasterization.

Triangle Rasterization

In this section are described the operations required to rasterize a triangle. In particular, an efficient triangle rasterization method is described. This method is based on rasterizing the edges of the triangle and by generating horizontal spans of pixels.

The color information for each pixel of a horizontal span of pixels can be linearly (or in same cases, for perspective correction, hyperbolically) interpolated from information available at the edges of each span. The hyperbolic interpolation is used when a perspective transformation was employed in the geometry stage in order to obtain a perspective compression effect. More detailed information on perspective corrected interpolation is available in Section 2.4.2.

Figure 2.4 depicts the basic model for triangle rasterization using linear interpolation to implement the Gouraud shading model.

Triangles can be rasterized by rasterizing their edges and by generating horizontal spans of pixels corresponding to the surface covered by a triangle across a horizontal raster line (scanline). Since the gradients used to generate the edges of each horizontal span can change when an intersection of two edges is encountered, some implementations are decomposing each triangle into 2 horizontal aligned triangles. This, way there are no gradient changes while rendering each horizontal aligned triangle. This step is usually integrated in a group of operations called triangle setup. As it can be seen in Figure 2.4 a point I_{13} has been added and two horizontal aligned triangles were obtained.

Although colors are usually linearly interpolated as we do here, better results can be achieved by using hyperbolic interpolation. For the Gouraud shading the difference is small, but for texture mapping process (described in Section 2.4.3) the difference is significant.

For rasterization it is common to use triangles or triangle strips as basic drawing primitives. The rasterizer interpolates the depth and color values for all the pixels bounded by the edges which define the triangles. Since triangles are planar shapes, a first idea to draw a triangle would be to linearly interpolate the vertex parameters along the edges and then to linearly interpolate the edge values along scan lines. The problem is that linear interpolation is correct only when all three vertices have the same depth (z distance) from the observer. Otherwise, a perspective correction is needed since the human perception of depth is not linear, but hyperbolic.

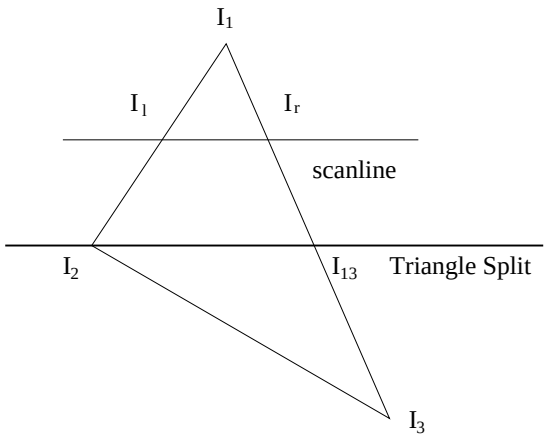


Figure 2.4: Triangle scan conversion.

By moving the slope and setup calculations for triangles in the rasterizer, the host processor is off-loaded from intensive calculations and this can significantly increase 3D system performance. Besides freeing the main processing unit, computing triangle setup data on the hardware rasterizer has the additional advantage of freeing the data bus. The usual amount of geometry information required to draw a triangle is shown in Table 2.1. If the triangle setup stage is implemented in hardware, the *Setup parameters* group will be computed by the triangle setup unit. Due to this, the required bandwidth between the CPU and the hardware accelerator to transfer the same number of triangles is reduced at by least 50%.

2.4.2 Span Generator

The purpose of this unit is to linearly interpolate a set of parameters passed from the *Triangle Setup* unit. To really understand what the problems are at this unit we recall that the last coordinate system used before device coordinates are the eye coordinates. We need to linearly interpolate all the parameters relative to window coordinates, but this will not always correspond to a linear interpolation of these parameters in eye space. The only case when the linear interpolation in window coordinates corresponds to a linear interpolation in eye space is when all the vertices of the primitive have the same depth, namely z, value in the eye coordinates.

To obtain a correct rendered primitive, theoretically, a division for each pa-

	Parameters
Triangle data	
Vertices	$x_0,\ y_0,\ z_0,\ x_1,\ y_1,\ z_1,\ x_2,\ y_2,\ z_2$
Colors	$r_0,\ g_0,\ b_0,\ \alpha_0,\ r_1,\ g_1,\ b_1,\ \alpha_1,$ $r_2,\ g_2,\ b_2,\ \alpha_2$
Texture Coordinates	$u_0,\ v_0,\ u_1,\ v_1,\ u_2,\ v_2$
Setup parameters	
Edge interpolation increments	$\Delta x/\Delta y$ (for 3 edges)
Color increments	$\delta r/\delta x,\ \delta r/\delta y,\ \delta g/\delta x, \delta g/\delta y,$
	$\delta b/\delta x,\ \delta b/\delta y,\ \delta \alpha/\delta x, \delta \alpha\ \delta y,$
Depth increments	$\delta z/\delta x,\ \delta z/\delta y$
Texture increments	$\delta u/\delta x,\ \delta u/\delta y,\ \delta v/\delta x,\ \delta v/\delta y$

Table 2.1: Triangle setup data.

rameter to be interpolated should be performed at each interpolation point. If we consider that a vertex in the eye coordinates has the following homogeneous representation: $P_e = [x_e, y_e, z_e, w_e]$, to obtain the window coordinates we perform a perspective projection and the respective homogeneous coordinates are: $P_w = [x_w, y_w, z_w, w_w]$. To obtain the normalized window coordinates we have to perform a division by the w_w component. This way the window coordinates of the P vertex are $P'_w = P_w/w_w = [x'_w, y'_w, z'_w, 1] = [x_w/w_w, y_w/w_w, z_w/w_w, w_w/w_w]$. A similar process should be applied to the parameters for each vertex (colors, texture coordinates) [37]. Therefore, the following steps have to be followed to perform correct primitive rendering using a linear interpolation method:

1. Construct a vector of values for each vertex of the triangle $V = [x_w, y_w, z_w, w_w, p_1, p_2, p_3, ..., p_n, 1]$, where n is the number of parameters that have to be interpolated for each vertex.

2. Divide V by w_w, the new vector $V' = [x'_w, y'_w, z'_w, 1, p'_1, p'_2, p'_3, ..., p'_n, 1']$, where $1' = 1/w_w$.

3. Linearly interpolate all the values of V' along polygon edges and across scan lines inside triangle using a formula similar with:

$$V_{current} = V_{initial} + f * (V_{final} - V_{initial}), \quad f \in [0, 1]$$

4. At each pixel divide the p' parameters by the corresponding $1'$ value to get the proper perspectively projected p values.

We remark that we are guaranteed that the $1'$ value is $\neq 0$, because after clipping all w_w are positive. Furthermore, instead of computing n divisions per pixel, we can compute one division and then n multiplications.

All the parameters computed at this unit are sent to the following units on a "per pixel" base. That is from now on we work at the pixel level instead of the primitive level.

2.4.3 Texture Mapping

In the quest for realism in 3D applications, it is desirable to render real-life alike images. One technique to achieve this goal is to scan 2D images and then to map these images to 3D surfaces to create the impression of 3D objects. Thereafter, these 3D objects are projected to a bi-dimensional surface that is usually a discrete space. This procedure is called "texture mapping" and is widely used in real-time 3D systems. Actually this approach can create a very realistic rendered images, but there are some trade-offs that can be made to trade speed for quality or vice versa. A very important problem in the texture mapping process is the quality of the mapping process and the artifacts that can arise due to aliasing problems. Another factor that affects the image quality is the discrete representation of the textures as arrays of elements due to limited storage space. Also, computing an exact projection of a pixel into the texture space, in real-time systems might be too slow. Consequently, approximation methods for performing texture mapping are used, but some rendering quality degradation may appear. In the following sections, relevant topics involved in the texture mapping process are described.

Direct Mapping and Inverse Mapping

There are two main methods to perform texture mapping: direct (forward) mapping and inverse mapping. Some authors [38] refer to these methods as texture order and screen order. In the case of direct mapping, as can be seen in Figure 2.5, for each texture element (texel) a correspondence (location) in the object space is found, and afterwards, by employing a projection operation, the correspondence in the screen space is determined. This can be written in pseudo-code as:

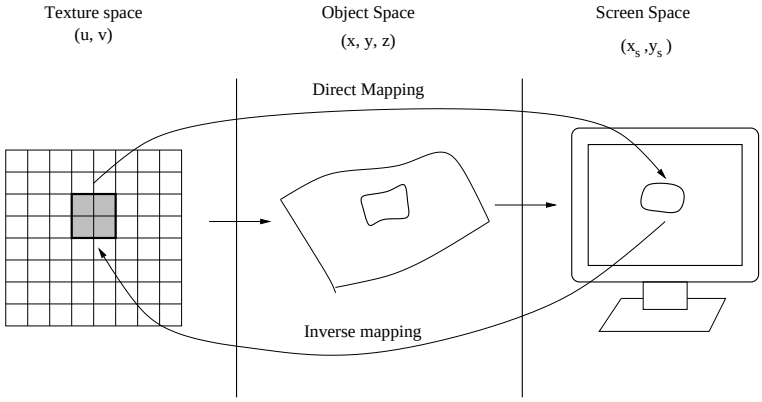


Figure 2.5: Direct and inverse mapping.

```
for each u
 for each v
     begin
       xs=x(u, v)
       ys=y(u, v)
       screen[xs, ys] = texture[u, v]
     end
```

where `x` and `y` are projection functions, `screen` is a buffer (array) containing the values of each pixel of the displayed image, and `texture` is an array that holds the values of the image we want to map to a surface. While this method appears straightforward, preventing holes or overlaps in the screen space is not a trivial process.

The other type of mapping, in screen space, consists of "walking" into screen space and for each pixel to compute its corresponding texels in texture space. By considering a pixel a square in the screen space, its correspondence in the texture space can be a curvilinear quadrilateral and usually a filtering process is needed to obtain a single value representative for the covered texels. In pseudo-code this type of mapping can be written as:

```
for each x
 for each y
     begin
       u=ut(x, y)
       v=vt(x, y)
       screen[x, y] = filt_texture(u, v)
     end
```

where `ut` and `vt` are functions used to map pixels to texels, `screen` is again a buffer (array) containing values of each pixel from the displayed image, and `filt_texture` is a filter function that returns a value corresponding to the group of texels closest to the (`u`,`v`) coordinates.

In the next sections we describe some of the possible filter functions used in current graphic accelerators.

Texture Mapping Filter Functions

The problem of finding the most accurate value (color) corresponding to a texel (*u*,*v*) where the coordinates *u*,*v* are real numbers, can be solved in many ways depending on the desired computational speed and accuracy. Since we are interested in performing a real-time operation, finding a very accurate value might not be the intention, but mostly to find an acceptable accurate value using a function within our speed and power consumption limits. The following sections will briefly describe some of the most often implemented filter functions in current rasterizers.

Nearest Filter

This filter is the fastest filter since the only operation it does is to return the value of the closest texel on the texel grid. In pseudo-code this filter can be written as:

```
ureal=ut(x, y)
vreal=vt(x, y)
uint=integer(ureal + 0.5)
vint=integer(vreal + 0.5)
return texel[uint, vint]
```

where `integer` is a function that returns the integer part of a real number. Since the nearest filter returns only one texel, it is the fastest filter and is suitable even for software renderers but its filtering quality is lower when compared to the following filters. Using this filter serious aliasing effects can occur due to the discrete nature of texture space.

Linear Filter

A more suitable filter is a linear interpolation filter. For each set of coordinates (u_r, v_r), as depicted in Figure 2.6, we linearly interpolate the closest four texels values. This filter can be written as:

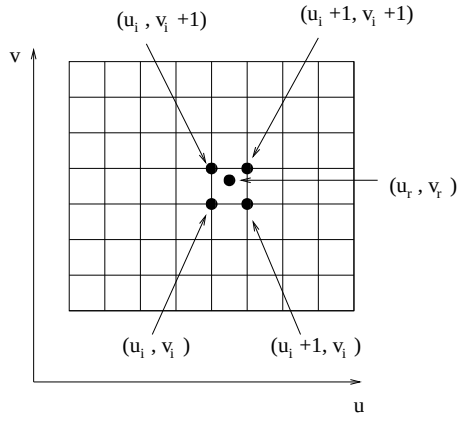


Figure 2.6: Linear interpolation control points.

```
ureal=ut(x, y)
vreal=vt(x, y)
uint=integer(ureal)
vint=integer(vreal)
s=ureal-uint
t=vreal-vint
return  (1-s)*(1-t)*texel[uint, vint]+
        (1-s)*t*texel[uint, vint+1]+
        s*(1-t)*texel[uint+1, vint]+
        s*t*texel[uint+1, vint+1]
```

We remark that this type of filter should be called bilinear filter since we have to interpolate twice, once for the *u* coordinate and once for the *v* coordinate. Since many 3D graphics card producers refer to this filter as linear and, the term bilinear is used for another filter that will be presented later, we will use the most encountered term. Thus, we will refer to it as linear.

This filter produces more accurate results than the "Nearest" filter, since it performs a certain degree of anti-aliasing. However, the mapped image has the tendency to become blurred compared to the previous filter which produces a sharper image but with more visible aliasing effects.

Mip-Mapping

To reduce aliasing effects that are still visible even when a linear filter is used, a better filter is needed. Considering that our focus is to produce real-time graphics, it is worth considering filters that precompute as much information as possible. One idea is to prefilter textures by storing a series of texture images of decreasing resolution, compared to the original image, which attempt to contain the same information in a smaller space. This technique is called MIP-map [74], where MIP stands for "*multum in parvo*" (much in a small place). It consists of building smaller versions of a texture from the original, each new texture being half the size, on each dimension, of the previous one. Thus, for instance for an image that has the initial dimensions of 512×512 we can build, using an anti-aliasing filter, a series of images having the dimensions 256×256, 128×128, ... , 2×2, 1×1. The reason for building these images can be explained as follows. If we want to map a 256×265 image to a four-pixel area on the screen, it would be very inefficient to apply an anti-aliasing filter that takes into consideration all the texels in the texture image in real-time. Instead, for such a case, a smaller texture which is already prefiltered can be used, and much of initial required computational power saved. The only remaining problem is how to select the best texture image plane(s) that we want to map to the respective area on the screen. To solve this problem, we have to compute a *Level of Detail* (LOD) factor, which will position us on the best texture image plane or between the two best texture image planes. A solution to this problem is given in [26].

In Figure 2.7 a MIP-map image series modeled as a pyramid is depicted. At the base of the pyramid is the initial image, which is the base level of a MIP-map series, and then as we go to the top of the pyramid, we find the lower resolution copies of the same image

We have to separate between two distinct situations.

1. When we have to map a number texels to a smaller number of pixels, it is called a MINIFICATION.

2. When we have to map a number of texels to a larger number of pixels

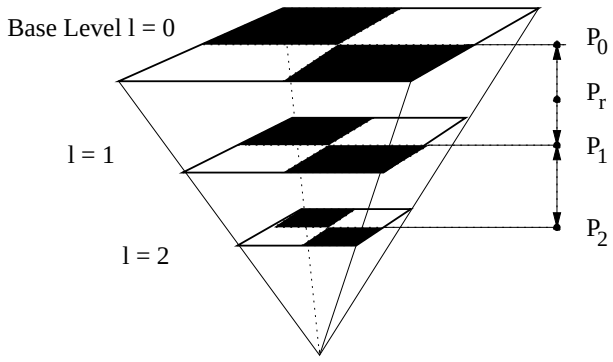


Figure 2.7: MIP-map image pyramid.

then we have a MAGNIFICATION situation.

The MIP-mapping technique is most useful for minification cases. In the magnification case, each texel corresponds to several pixels. For this case the initial image is always used since it is the most detailed image available.

Minification Filters There are basically four types of filters in this situation, and we will describe each of them briefly. Assume that the LOD was computed and its value is somewhere between the base level (0), and the first level of MIP-map for which l equals 1. As shown in Figure 2.7, P_0 is the value corresponding to the LOD for the base level and P_1 is the value of the LOD corresponding to the first level, and P_r is the value corresponding to the actual LOD. The most used minification filters are:

- **Nearest Mip-map Nearest** - In this filter only the nearest (closest) image plane (with respect to LOD) is chosen. Consequently, the nearest filter described in Section 2.4.3 is applied to this image.
- **Linear Mip-map Nearest** - In this case the nearest image plane is also chosen as in the previous case, but this time a linear filter (see Section 2.4.3) is applied on the resulted image to obtain a better quality. This filter it is also known as “Bilinear” or more correctly MIP-map bilinear since the linear filter mentioned previously is also bilinear but we refer to it as linear to avoid confusion.

- **Nearest Mip-map Linear** - In this case we do not choose only one image plane but two, the ones that are closest to the LOD. In each of the selected image plane a nearest filter is applied and the result is obtained by a linear interpolation between the values computed at each image plane.
- **Linear Mip-map Linear (Trilinear)** - This filter is the most computationally intensive, compared to the previous ones, since it takes into consideration the closest two image planes and on each plane a linear interpolation is performed. After that, a linear interpolation is performed between the two values found for each plane. This filter is also known as *trilinear*.

Magnification Filters As we have previously mentioned in the case of magnification only the base image is used. Therefore, there are only two filtering alternatives:

- Nearest - the same as the nearest filter described in Section 2.4.3.
- Linear - performs bilinear interpolation in the base image plane being equivalent to the linear filter described in Section 2.4.3.

Actually in this case, of magnification, the MIP-mapping process is not used since there are no image planes constructed with more details than the base level.

While describing the texturing process some texturing mechanism details were omitted for clarity. In the following sections we briefly describe the following details:

1. Mip-map level selection when using mip-maps.
2. What happens if the texture coordinates are bigger than the texture size?

Mip-map Level Selection

One reason for using mip-mapping is to increase the texture filtering quality by using prefiltered (smaller) copies (texture planes) of the original textures. One problem with Mipp-mapping is that additional computations for selecting the *best* texture plane(s) from which to sample the textures are required. As

a starting point, the best texture plane is a plane for which the ratio texels per pixel is the closest to 1.

In order to find the best texture plane(s) we can use the following parameters [65]:

Let $\rho(x, y)$ be a scale factor and the level of detail parameter $\lambda(x, y)$ defined as:

$$\lambda'(x, y) = \log_2(\rho(x, y)) \tag{2.4}$$

$$\lambda(x, y) = \begin{cases} Max\_LOD, & \text{if } \lambda'(x, y) > Max\_LOD \\ \lambda', & Min\_LOD \leq \lambda' \leq Max\_LOD \\ Min\_LOD, & \lambda' < Min\_LOD \\ undefined & Min\_LOD > Max\_LOD \end{cases} \tag{2.5}$$

where *Max_LOD* and *Min_LOD* are constants corresponding to the minimum and the maximum level of detail. If $\lambda(x, y)$ is less than or equal to a constant c the texture is magnified; if it is greater, the texture is minified. The c constant, the minification vs. magnification switch-over point, depends on the minification and the magnification filters. If the magnification filter is given by LINEAR and the minification filter is given by NEAREST_MIPMAP_NEAREST or NEAREST_MIPMAP_LINEAR, then c = 0.5. This is done to ensure that a minified texture does not appear sharper than a magnified texture. Otherwise c = 0.

Considering that the u, v coordinates are functions of x and y, that is $u = u(x, y)$, $v = v(x, y)$ then the ρ function is defined as:

$$\rho = max\left\{\sqrt{\left(\frac{\delta u}{\delta x}\right)^2 + \left(\frac{\delta v}{\delta x}\right)^2}, \sqrt{\left(\frac{\delta u}{\delta y}\right)^2 + \left(\frac{\delta v}{\delta y}\right)^2}\right\} \tag{2.6}$$

While Equation (2.6) gives the best result when texturing, it is often impractical to implement. Therefore, an implementation may approximate the ideal ρ with a function $f(x, y)$ subject to the conditions presented in [65].

Texture Wrap Modes

For practical reasons, a texture can be considered bigger than its real size. The texture in this case is considered as having a symmetrical border on each dimension such that if b_s is the size of the border, then for instance the new width of a texture becomes: $w_{new} = w_{real} + 2 * b_s$. When we described the texture

filters (in Section 2.4.3) we used the u and v parameters to indicate the texture coordinates. These u and v parameters are integer parameters, where $u \in [0..texture_{rows}]$, $v \in [0..texture_{columns}]$, $texture_{rows}$ and $texture_{columns}$ are powers of 2. A set of corresponding real parameters s, t can be defined that are normalized with respect to texture size such that:

$$s = \frac{u}{texture_{columns}} \tag{2.7}$$

$$t = \frac{v}{texture_{rows}} \tag{2.8}$$

Supposing that we receive the s, t texture coordinates as coordinates for the texture map, but instead of being defined on [0,1], they can also have an integer part. In this case a value larger than 1 for s or t would result in a translated physical texture coordinate that is higher than the corresponding size of the texture. There are three methods that can be used to transform the s, t values that $\notin [0, 1]$ to values $\in [0, 1]$ that will have a valid physical correspondent in texture space.

1. Repeat - in this case the integer part of the s or t coordinates is ignored, only the fractional part being used. For a number f, the fractional part is $f - \lfloor f \rfloor$, regardless of the sign of f; recall that the floor function truncates towards $-\inf$.
2. Clamp - in this case the s or t coordinates are clamped to [0,1].
3. Clamp to edge - clamps texture coordinates in all mipmap levels such that the texture filter never samples a border texel.

The color returned when clamping to edge is derived only from texels at the edge of the texture image. Texture coordinates are clamped to the range [min; max]. The minimum value is defined as $min = \frac{1}{2N}$ where N is the size of the one or two-dimensional texture image in the direction of clamping. The maximum value is defined as $max = 1 - min$ so that clamping is always symmetric about the [0, 1] mapped range of a texture coordinate.

More detailed information regarding the texturing process and its associated computational power can be found in [65].

Anisotropic Filtering

The MIP-mapping technique is not the best texture mapping technique. For practical reasons, it was an inexpensive solution, with respect to computational

speed, to achieve fast texture mapping with a certain degree of anti-aliasing. MIP-mapping described above is based on an isotropic square filter shape. This filter is variant in size, but not in shape. This means that if we look straight to a wall, for example, the mapping will be correct, but if we have a different angle, then the rendered image will not be accurate since pixels are no longer projected into square texels. That is the projection on the u and v axes are not equal so the screen to texture space scaling is anisotropic. One method to implement anisotropic filtering is still to preserve the trilinear technique described above, but to perform multiple trilinear filter operations along the direction of the anisotropy [26] [61] [13].

2.4.4 Per Fragment Operations

This section gives a brief description of the operations performed at a pixel granularity and depicted in Figure 2.3. Although, in the OpenGL specification [64] the Clear, Fog and Color Sum components are not considered as per-fragment operations, they can be conceptually included in the Per Fragment Operations group.

Clear Unit

The clear unit is used to fill the Depth buffer and/or the Color buffer with a default value. The clear operation can be either implemented as a separate 2D operation, or by rendering a large primitive that would cover the entire rendering region.

Fog Unit

This unit is responsible for generating the fog effect. A selection of linear, exponential or squared exponential fog propagation models can be selected to combine the incoming color with fog color selected by the application.

The Fog unit affects only the RGB color components while leaving the fragment's alpha component unchanged.

Color Sum

Depending on the lighting parameters, a fragment has a primary RGBA color which is the color we normally refer as fragment's color, but it might also have

a secondary color, that is also named specular color. The purpose of this block is to effectively sum the primary and the specular colors for each fragment.

According to the OpenGL graphics system diagram [63], the specular color comes directly from the lighting stage (unit), but practically, this is not true since the lighting stage generates colors at a primitive (vertex) level, for instance for each vertex of a triangle, while at the color sum unit, the colors at the pixel level should be added. This means that they should pass through the color interpolation stage (span generator). Thus, the specular color should also be interpolated using the span generation unit, and it can not come directly from the lighting stage.

Pixel Ownership

This test determines if the pixel at location (x, y) in the framebuffer is currently owned by the GL window. If it is not, the window system decides the status of the incoming fragment. Usually, in this case, the fragment is discarded or a subset of the subsequent per-fragment operations are applied to the fragment. This test allows the window system to control the GL's behavior, for instance, when a GL window is obscured by other windows.

We remark that there is no indication in the OpenGL specification about how this test should be implemented. This allows for a broad spectrum of solutions.

One method of implementing this test is to allocate an ID to each window, and each primitive, consequently each pixel of the primitive, will have assigned the ID corresponding of its window. Also a buffer to store the ID for each pixel on the output device is needed. The ID of the incoming pixel is compared with the ID from the ID buffer at the pixel's position. If the ID's are different then the new pixel is discarded.

Scissor Test Unit

This unit determines if a pixel's coordinates (x, y) are inside of a rectangular portion of a window $(left, bottom, left + width, bottom + height)$.

If the fragment is inside the scissor window, it is passed to the next unit otherwise it is discarded. The test is performed only when the unit is enabled.

Alpha Test Unit

The alpha test discards a fragment conditional on the outcome of a comparison between the incoming fragment's alpha value and a constant reference alpha value. If the comparison succeeds, the fragment is passed to the next unit, otherwise it is discarded. The comparison is performed only when the unit is enabled.

Stencil Test Unit

The stencil test conditionally discards a fragment based on the outcome of a comparison between the value in the stencil buffer at location (x, y) and a reference value (refval). Both the reference value and the value from the stencil buffer are AND-ed with a stencil mask, that can be set by the user or GL driver, before being compared. When disabled, the stencil test and associated modifications are not made, and the fragment is always passed. This unit and the depth test unit (described in Section 2.4.4) are tightly coupled meaning for instance that a fragment's corresponding stencil value (in the stencil buffer) cannot be updated, if it passes the stencil test, until the result of the depth test comparison is known. Only when the stencil test fails then we now for sure that we can modify the corresponding stencil value from the stencil buffer according to a function selected by the application.

Depth Test Unit

This test (unit) compares the z value of a fragment with a value stored at fragment's (x,y) coordinates in the depth buffer

If there is no depth buffer, it is as if the depth buffer test always passes.

If the depth test fails, the incoming fragment is discarded and the stencil value at (x,y) is updated according to the function selected for such case. If the depth test passes then the fragment is passed to the next unit and the stencil buffer value at (x,y) is updated accordingly to the function specified for this case.

Blending

Blending combines the incoming fragment's R, G, B, and A values with the R, G, B, and A values stored in the framebuffer at the incoming fragment's (x,y) location. As an extension, a separate blending color (constant color) can

be used in the combine process. If an implementation is allowing the use of a separate blending color, then an entire package of features, as described in [65], and called *imaging subset features* must be supported. Since our intention is to provide a minimum OpenGL implementation, the imaging subset features might not be implemented so the constant color also can be eliminated. Also the blending equation described in [65] must have multiple forms in case of supporting imaging subset features. From now on we consider we do not support the imaging subset extension, so we will consider only the mandatory features. The blending process is dependent on the incoming fragment's alpha value and that of the corresponding currently stored pixel. If the currently stored pixel has no alpha value, then it is considered that it has an alpha value of 1.0 (maximum value for a [0,1] floating point).

Dithering

Dithering selects between two color values in order to create patterns that would give the illusion of a higher resolution. Many dithering algorithms are possible, but a dithered value produced by any algorithm must depend only the incoming value and the fragment's x and y window coordinates. If dithering is disabled, then each color component is truncated to a fixed-point value with as many bits as there are in the corresponding component in the framebuffer.

Logic Op

A logical operation such as AND, OR, or XOR, can be applied between the incoming fragment's color and the color values stored at the corresponding location in the framebuffer. The result replaces the values in the framebuffer at the fragment's (x, y) coordinates. Logical operations are performed independently for each red, green, blue, and alpha value of each color buffer that is selected for writing. Actually, since a logical operation is computed bit-wise its only restriction is that the operands to be integer values. Whenever the logical operations are enabled the blending operations are automatically ignored. Therefore, graphical application designers, depending on the intended effects, have the option to enable either the blending or the logical operation unit.

Masking

Each logical framebuffer that we use, according to the OpenGL specification, should have means of enabling or disabling writing to some of its fields.

2.4.5 Buffers Used by The Rasterization Pipeline

Depending on the complexity of the implementation and the purpose of the graphic hardware, the rasterizer stage can be responsible besides generating the color for each pixel, to also generate other information that can be stored in additional locations (buffers) and can be employed to increase the quality (realism level) of the generated image. For instance, a rasterization device can contain a sum of the following buffers: z (or depth) buffers, alpha buffers, stencil buffers, and accumulation buffers.

- Z buffer - As described in Chapter 1, this buffer holds information about the distance to the closest visible pixel relative to the viewer.
- Alpha buffer - This buffer can hold an opacity value very useful in blending objects or transparency effects. This buffer is usually part of the color buffer.
- Stencil buffer - This buffer holds information that can be used to create special effects such as object masking or to simulate shadow casted by objects. For example in a flight simulator generated image, a *0* value in the stencil buffer can be associated to pixels covered by the cockpit, and an *1* value for the pixels that should be rendered by the rasterizer. By using this representation, a rasterizer can test the stencil values (bits) and write in the framebuffer only the pixels that have an *1* value in the stencil buffer.
- Accumulation buffer - This buffer can be useful in a full scene antialiasing process by accumulating more images and combining them as requested by an algorithm. Also this buffer can be used for motion blur effects.

2.5 Concluding Remarks

In this chapter we described the main components of a 3D graphics pipeline. We started by providing a short overview of a generic 3D graphics pipeline and further-on we focused on the rasterization stage of the graphics pipeline. A particularity of the rasterization stage is given by the explosion of data traffic and computational power required to implement primitive conversion to fragments.

Since an important part of any current rasterization stage is represented by the texture mapping process, in this chapter we have emphasized the description of the texture mapping process components. As we have seen there are many ways to perform texture mapping. In this chapter we have described only the methods that are currently implemented in hardware accelerators. Other methods such as *Summed area tables* [22] are not used in hardware due to the high storage requirements. As a concluding remark for texturing, the *nearest* filter is the fastest one but it generates the worst mapping quality. The *linear* filter is smoother than the *nearest* one, but using only the *linear* filter without MIP-mapping in situations where the ratio between covered texels and corresponding pixels is larger than one, can significantly reduce the quality of filtering. For such situations it is recommended to use MIP-mapping. Moreover, when the shape of the projection in texture space of a screen region is not square then anisotropic filtering, which is significantly slower, should be used.

While texturing requires significant processing power, the other rasterization stage operations, organized as a pipeline, are also suitable for hardware implementation. Thus, the focus in designing a 3D graphics accelerator is not only on the texturing process but on the entire rasterization stage.

Chapter 3

GraalBench

In this chapter we argue that 3D benchmarks employed for desktop computers are not suitable for mobile environments. Also, to the best of our knowledge, there is no portable, publicly available benchmark suite that can be used to guide the architectural exploration of low-power mobile devices.

Therefore, in this chapter we present GraalBench, a 3D graphics benchmark suite suitable for evaluating 3D graphics on low-power, mobile systems, in particular mobile phones. These benchmarks were collected to facilitate the work presented in this thesis. It includes several games as well as virtual reality applications such as 3D museum guides. Applications were selected on the basis of several criteria. For example, CAD/CAM applications, such as contained in the Viewperf package [69], were excluded because it is unlikely that they will be offered on mobile devices. Other characteristics we considered are resolution and polygon count.

A second goal of this chapter is to provide a detailed quantitative workload characterization of the collected benchmarks. For each rasterization unit, we determine if it is used by the benchmark, and collect several statistics such as the number of fragments that bypass the unit, fragments that are processed by the unit and pass the test, and fragments that are processed but fail the test. Such statistics can be used to guide the development of mobile 3D graphics architectures. For example, a unit that is rarely used might not be supported by a low-power accelerator or it might be implemented using less resources. Furthermore, if many fragments are discarded before the final tests the pixel pipeline of the last stages might be narrower than the width of earlier stages.

This chapter is organized as follows. Previous work on 3D graphics bench-

marking is described in Section 3.1. In this section we also give reasons why current 3D graphics benchmarks are not appropriate for mobile environments. Section 3.2 describes the components of the proposed benchmark suite. Section 3.3 describes our tracing environment and explains how the benchmarks were obtained. Section 3.4 provides a workload characterization of the benchmarks and discusses architectural implications. Conclusions are given in Section 3.5.

3.1 Related Work

To the best of our knowledge, portable 3D graphics benchmarks specifically targeted at low-power architectures have not been proposed. Furthermore, existing benchmarks cannot be considered to be suited for embedded 3D graphics architectures. For example, consider SPEC's Viewperf [69], a well-known benchmark suite used to evaluate 3D graphics accelerator cards employed in desktop computers. These benchmarks are unsuitable for low-power graphics because of the following:

- The Viewperf benchmarks are designed for high-resolution output devices, but the displays of current wireless systems have a limited resolution. Specifically, by default the Viewperf package is running at resolutions above SVGA (800x600 pixels), while common display resolutions for mobile phones are QCIF (176x144) and QVGA (320x240).
- The Viewperf benchmarks use a large number of polygons in order to obtain high picture quality (most benchmarks have more than 20,000 triangles per frame [51]). Translated to a mobile platform, most rendered polygons will be smaller than one pixel so their contribution to the generated images will be small or even invisible. Specifically, the polygon count of the Viewperf benchmarks DRV, DX, ProCDRS, and MedMCAD is too high for mobile devices.
- Some benchmarks of Viewperf are CAD/CAM applications and use wire-frame rendering modes. It is unlikely that such applications will be offered on mobile platforms.

Except Viewperf, there are no publicly-available, portable 3D graphics benchmark suites. Although there are several benchmarking suites [18, 19] based on the DirectX API, they are not suitable for our study since DirectX implementations are available only on Windows systems. After we proposed GraalBench,

other benchmarking suites were proposed. For instance, the SPMark04 [20] benchmarking suite, although more suitable for portable graphics than previously mentioned suites, is only available as a binary for several platforms.

There have been several studies related to 3D graphics workload characterization (e.g., [51, 23]). Most related to our investigation is the study of Mitra and Chiueh [51], since they also considered dynamic, polygonal 3D graphics workloads. Dynamic means that the workloads consist of several consecutive image frames rather than individual images, which allows to study techniques that exploit the coherence between consecutive frames. Polygonal means that the basic primitives are polygons, which are supported by all existing 3D chips. The main differences between that study and our workload characterization are that Mitra and Chiueh considered high-end applications (Viewperf, among others) and measured different statistics.

Recently, a number of mobile 3D graphics accelerators [9, 67] have been presented. In both works particular benchmarks were employed to evaluate the accelerators. However, little information is provided about the benchmarks and they have not been made publicly available.

Another reason for the limited availability of mobile 3D graphics benchmarks is that until recently there was no generally accepted API for 3D graphics on mobile phones. Recently, due to high interest in embedded 3D graphics, APIs suitable for mobile 3D graphics such as OpenGL ES [35], Java mobile 3D Graphics API (JSR-184) [34], and Mobile GL [12] have appeared. Currently, however, there are no 3D benchmarks written using these APIs. So, we have used OpenGL applications. Furthermore, our benchmarks use only a part of the OpenGL functionality which is also supported by OpenGL ES.

3.2 The GraalBench Benchmark Set

In this section we describe the components of our benchmark set and we also present general characteristics of the workloads.

The proposed benchmark suite consists of the following components:

Q3L and Q3H Quake III [43] or Q3, for short, is a popular interactive 3D game belonging to the shooter games category. A screenshot of this game is depicted in Figure 3.1(a). Even though it can be considered outdated for contemporary PC-class graphics accelerators, it is an appropriate and demanding application for low-power devices. Q3 has a

(a) Q3 (b) Tux (c) AW

(d) ANL (e) GRA (f) DIN

Figure 3.1: Screenshots of the GraalBench workloads

flexible design and permits many settings to be changed such as image size and depth, texture quality, geometry detail, types of texture filtering, etc. We used two profiles for this workload in order to determine the implications of different image sizes and object complexity. The first profile, which will be referred to as Q3H, uses a relatively high image resolution and objects detail. The second profile, Q3L, employs a low resolution and objects detail. Q3 makes extensive use of blending operations in order to implement multiple texture passes.

Tux Racer (*Tux*) [70] This is a freely available game that runs on Linux. The goal of this game is to drive a penguin down a mountain terrain as quickly as possible, while collecting herring. The image quality is higher than that of Q3. Tux makes extensive use of automatic texture coordinate generation functions. A screenshot can be seen in Figure 3.1(b).

AWadvs-04 (*AW*) [69] This test is part of the Viewperf 6.1.2 package. In this test a fully textured human model is viewed from different angles and distances. As remarked before, the other test in the Viewperf package are not suitable for low-power accelerators, because they represent high-end applications or are from an application domain not likely to be offered on mobile platforms. A screenshot of AW is depicted in Figure 3.1(c).

ANL, GRA, and DIN These three VRML scenes were chosen based on their

diversity and complexity. ANL is a virtual model of Austrian National Library and consists of 10292 polygons, GRA is a model of Graz University of Technology, Austria and consists of 8859 polygons, and Dino (DIN) is a model of a dinosaur consisting of 4300 polygons. In order to obtain a workload similar to one that might be generated by a typical user, we created "fly-by" scenes. Initially, we used VRWeb [55] to navigate through the scenes, but we found that the VRMLView [42] navigator produces less texture traffic because it uses the `glBindTexture` mechanism. Screenshots of ANL, GRA, and DIN are depicted in Figures 3.1(d),(e), and (f).

GraalBench is the result of extensive searching on the World Wide Web. Although there are more OpenGL applications available, most represent high-end applications and thus are not suited to evaluate low-power 3D graphics architectures.

Recently, several links to 3D games were provided on Mesa's website [58]. However, these games such as Doom, Heretic, and Quake II belong to the same category as Quake III and Tux Racer, and therefore do not represent benchmarks with substantially different characteristics. We, therefore, decided not to include them.

Applications using the latest technologies (Vertex and Pixel Shaders) available on desktop 3D graphics accelerators were also not included since these technologies are not supported by the embedded 3D graphics APIs mentioned in Section 3.1. We expect that more 3D graphics applications for low-power mobile devices will appear when accelerators for these platforms will be introduced.

3.3 Tracing Environment

In this section we describe the environment we used to create the benchmarks. One of the required tools for our project is an OpenGL compliant tracer, that can log the calls made by an application to an OpenGL library. Such a tool can indicate which OpenGL functions are frequently used by applications. Thus it can provide hints about what functionality should be implemented in hardware in an OpenGL accelerator. Because some of the OpenGL functions depend on implicit state variables (Section 3.3.6), those functions must be explicitly expanded in self-contained functions in order to make a reproducible trace.

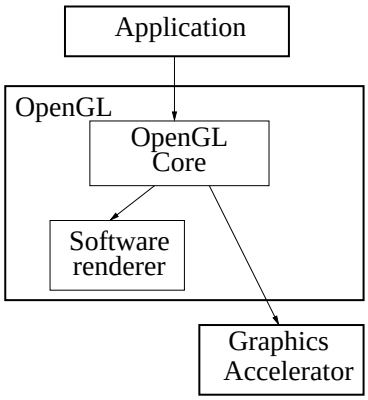


Figure 3.2: OpenGL scenario.

3.3.1 OpenGL Environment

A typical OpenGL environment is depicted in Figure 3.2. Applications call OpenGL functions in order to perform 3D rendering. The OpenGL library core determines if there is hardware support for the application call or it must execute in software the required function. The modality in which application is bound (linked) to the OpenGL library is important for the tracing mechanism to work properly. On our targeted systems (Linux and Windows) there are two possible linking methods:

- Static Linking

  In this case the required libraries are encapsulated in the executable image of the application. The advantage of this type of linking is that the application does not need other external libraries for its execution but the disadvantage is that this scheme is usually not flexible enough for OpenGL applications. If a new version of the library is developed the application cannot use the new version. Our tracing mechanism cannot be used if this type of linking is chosen.

- Dynamic Linking

  In this linking scheme the executable holds only links to required functions from external libraries. At run-time a dynamic linker or even the application can obtain links to instances of the required functions from external libraries. This linking scheme is more flexible and by comparison to static linking it has the advantage that it reduces the size of the

executable since each application will access a common library instead of encapsulating it. Also, if a new version of the library is developed all the applications can take advantage of the new library.

The dynamic linking scheme is useful for tracing the behavior of applications since libraries can be cascaded so that a tracing library can be placed between the application and the real library. Even though under different operating systems they have different extensions (under Windows they have a .dll extension and under Linux they have a .so extension) the dynamic linkable libraries (DLL's) have the same behavior.

3.3.2 OpenGL Tracers

Different OpenGL tracing solutions are currently available. One solution is provided for instance by **ZAPdb** OpenGL debugger, from IBM, that is closely linked with a particular OpenGL implementation. Even if it appears that it provides extensive support for OpenGL, we could not trace Quake III [43] under Windows with it. Another OpenGL (actually IRIX GL) debugger is **gldebug** from SGI. This debugger was not available neither on Windows nor on Linux systems and it was not tested.

Recently the **Mesa** library incorporates tracing support, but it is not completely implemented. Another disadvantage of the Mesa tracer is that it can be used only with the Mesa library. Real OpenGL independent approach (standalone) to tracing OpenGL calls was taken by the **gltrace** library from Stanford. This tracer is available for Linux and also for Windows systems. It can generate complete traces of execution for the Quake III game but only if OpenGL extensions are turned off. Even though it can generate even binary traces (for increased performance), it does not support OpenGL multitexturing extension. Also no source code is available.

Another OpenGL tracing library is **spyGLass** [68]. This tracer automatically generates the library code based on SGI specification for each OpenGL function. This library is not yet completely usable, and moreover since it generates the tracing code automatically it cannot generate complete traces.

Finally, the **GLtrace** [66] library available from Hawksoft was evaluated. For this library there is also source code available. This library is one of the most complete OpenGL trace library, but it does not log completely reproducible OpenGL calls. It has also the disadvantage that the logging process is performed only in text mode (slow).

3.3.3 Grtrace

This sections describes our OpenGL tracer (Grtrace), used to trace relevant applications to our project. Grtrace is an improved version of the GLtrace version 2.3 OpenGL tracer. The main improvements of Grtrace over Gltrace are:

- The code was rearranged into a more conventional coding style. OpenGL extensions were placed in a separate ext.c file instead of the ext.h header file. In the header files mostly external declarations of common variables and functions remained.
- Some minor errors were eliminated. For instance GlTrace logged glTranslatef function as glTranslated.
- A GLX function GLXChooseVisual, when logged verbosely, was crashing the library if it could not find a suitable context, due to passing a NULL pointer to a XVisualInfo structure to the logger.
- Initial support for the following extension functions was added: glTexImage3D, glCompressedTexImage2DARB, glCompressedTexSubImage2DARB, glFlushVertexArrayRangeNV, glVertexArrayRangeNV, glCombinerParameterfNV, glCombinerParameteriNV, glCombinerInputNV, glCombinerOutputNV, glFinalCombinerInputNV and glColorTableEXT.
- On Windows platforms the default behavior when an extension function is requested by an application from the OpenGL library was modified. If the application requests a pointer to a function that is not implemented in the Grtrace library, but it is implemented in the target OpenGL implementation, the library will return to the application a pointer to the real function and will also put in the trace log a warning message that the function is not traceable instead of just stopping the application. This behavior has the advantage that applications can be executed even if there are new extension functions that are not implemented in Grtrace but will be implemented in future OpenGL implementations.
- **Added support for binary logging**. Even though the support for binary logging is not yet complete, the tracer can be used to trace several “demanding” applications in binary mode. As explained in Section 3.3.5 this improves tracing performance.

3.3.4 Generating Portable Traces

One design issue about OpenGL is that the initialization of the graphics windows and context is window manager dependent. Since our goal is to log and reproduce OpenGL calls on two target systems (Windows and Linux) we need a flexible mechanism to transform window dependend calls to independent generic calls. While an application uses GLX calls to initialize OpenGL rendering under Linux, or WGL calls under Windows, the initialization sequence is similar and the main operations are as follows:

- Look for an appropriate OpenGL implementation
- Create an OpenGL context
- Create a window
- Assign the OpenGL context to the window
- Display window

These operations can be synthesized in a portable way using two functions:

- CreateContext - This function creates a rendering context with the required OpenGL features (if required also a window).
- MakeCurrent - This function assigns an existing rendering context to the current rendering window.

Even if the text mode based log of Grtrace logs the calls in a windows dependent manner (one to one function mapping), in the binary log mode the calls are logged in a portable way such that they can be reproduced on either Linux or Windows.

The above mentioned function are not the only system dependent functions. For example, the function that swaps rendering buffers (e.g., glXSwapBuffers) is also platform specific but it is also logged into a portable manner.

3.3.5 Improving Tracing Performance

Since the tracing operation can slow down the running application considerably as well as modify the behavior of interactive applications, it is worth

considering techniques to improve the tracing speed. If the tracing performance is slow, then the collected calls can be be significantly different from the ones normally used since the application might skip rendering some of the frames. Initially Grtrace was generating text logs, but this approach is quite slow since for each logged function formatting of the parameters must be done in real-time and this can induce large latencies in rendering. One optimization is to perform binary logging. In this approach the overhead for logging calls is reduced considerably. An even faster method is to use a buffering system. Considering the binary format chosen for logging, a significant number of requests can be used to log only small chunks of data. For instance, in the fragment code:

```
void GLAPIENTRY glAlphaFunc (GLenum func, GLclampf ref)
{

    STARTBIN(TKG_ALPHAFUNC); //writes 2 bytes
    CMDLEN(8); //command body len. writes 2 or 4 bytes
    print_value_bin(_GLenum, &func); //write 4 bytes
    print_value_bin(_GLclampf, &ref); //write 4 bytes
    ENDBIN;

    //call to the target glAlphaFunc
    GLV.glAlphaFunc (func, ref);

}
```

each call to the *print_value_bin* function writes 4 bytes. First an integer is written and secondly a float. If the *print_value_bin* function is implemented directly using *fwrite* calls then a significant overhead occurs for only 8 bytes. By using a buffering scheme and calling the *fwrite* for larger chunks of data increases performance significantly.

3.3.6 Reproducing OpenGL Calls Made by Applications

One of the important difference between Grtrace and a conventional tracer that just logs OpenGL calls is that it performs complete logs of calls. For example if an application calls a function that has a parameter that points to a data structure (e.g., texture), a conventional logger would only log the pointer as an address without any information about the contents of the data structure. But if it is desired to reproduce the same call later on, a complete knowledge of the data the pointer refers to is required.

Problematic OpenGL Calls

In order to increase the rendering performance of hardware OpenGL implementations, starting with version 1.1 of the OpenGL specifications, new functions were added. Flexible OpenGL implementations have a client-server architecture. The client part of the OpenGL is responsible for receiving commands from applications, while the server part handles the execution of the commands received by the client side. The server part of the library can be executed on a different machine than the client part using for instance, a computer network protocol to communicate (e.g., the GLX protocol). Even if the client and the server components are running on the same machine they do not access the graphics hardware directly. While this approach is very flexible it has a major limitation: all data must be copied from the client to the server. This process can slow the rendering process considerably. Since all the commands are executed indirectly this rendering method is also called *indirect rendering*. In order to overcome this problem, flexibility was traded for speed and most hardware vendors are using a *direct rendering* method. In this case the OpenGL implementation accesses the graphics hardware directly, so actually the client and the server components are located on the same machine and they share the same memory space.

One of the improvements to increase the rendering speed, introduced in OpenGL 1.1, was support for arrays. Typical 3D applications are rendering hierarchical scenes based on objects. Each object is decomposed into triangles (primitives) in order to be rendered. Most of the time objects are sharing vertices. Instead of sending the information related to each vertex multiple times, arrays of properties (e.g., vertex coordinates, colors, texture coordinates) can be formed and sent from the application to the OpenGL implementation. The vertex arrays can be stored, for instance, in a local fast memory on the graphics accelerator. Each time the information for a vertex is required only an array index is sent instead of resending all the information related to the respective vertex. In the OpenGL standard the functions related to arrays are implemented using a call-by-reference mechanism. For each property that an application creates an array of, it must call the corresponding OpenGL function. For example, if an application creates an array with the coordinates of some vertices, it uses the *glVertexPointer* function declared as:

```
void glVertexPointer( int size, enum type,
                      sizei stride, void *pointer);
```

where

size	specifies the number of coordinates per vertex,
type	specifies the data type of each coordinate,
stride	specifies the byte offset between consecutive vertices,
pointer	is the pointer to the beginning of the array.

After specifying the arrays, an application might refer to any array element by sending, to the OpenGL library, the index of the element using the *glArrayElement* function. By sending only the array indices instead of the content of the array, applications can save substantial data traffic amounts.

However, since there is **no parameter** for the number of vertices in the array, the size of the array is not known and the array cannot be logged. One solution to this problem was to introduce extension functions that can be used by applications to specify the size of the array or the range of the elements that need to be rendered. However, applications are not required to use these extensions. Therefore, OpenGL tracers must expand each *glArrayElement* call into basic operations. Furthermore, since our tracer cannot log arrays without knowing their lengths, in order to make a complete trace, all the functions related to arrays have to be decomposed.

Some of the relevant functions that need expanding in order to be logged properly are *glArrayElement*, *glDrawArrays*, and *glDrawElements*. These functions can be expanded as described in the OpenGL Specification [65].

In addition to the tracer, we developed a trace player that plays the obtained traces. It can play recorded frames as fast as the OpenGL implementation allows. It does not skip any frame so the workload generated is always the same. The workload statistics were collected using our own OpenGL simulator based on Mesa [58], which is a public-domain implementation of OpenGL.

3.4 Workload Characterization

This section starts by providing general characteristics of the proposed components followed by the detailed analysis of results we obtained by running the proposed benchmark set. For each unit of a typical rasterization pipeline we present the relevant characteristics followed by the architectural implications.

Bench.	Resolution	Frames	Textures (MB)	Received Triangles		Processed Triangles		Area	
				Avg.	Max.	Avg.	Max.	Avg.	Max.
Q3L	320x240	1,379	12.84	4.5k	9.7k	3.25k	6.8k	422k	1,327k
Q3H	640x480	1,379	12.84	4.6k	9.8k	3.36k	6.97k	1,678k	5,284k
Tux	640x480	1,363	11.71	3k	4.8k	1.8k	2.97k	760k	1,224k
AW	640x480	603	3.25	23k	25.7k	10.55k	13.94k	63k	307k
ANL	640x480	600	1.8	4.45k	14.2k	4.45k	14.2k	776k	1,242k
GRA	640x480	599	2.1	4.9k	10.8k	3.6k	6.9k	245k	325k
DIN	640x480	600	1.7	4.15k	4.3k	4.15k	4.3k	153k	259k

Table 3.1: General statistics of the benchmarks

3.4.1 General Characteristics

Table 3.1 present some general statistics of the workloads. The characteristics and statistics presented in this table are:

Image resolution Currently, low-power accelerator should be able to handle scenes with a typical resolution of 320x240 pixels. Since in the near future the typical resolution is expected to increase we decided to use a resolution of 640x480. The Q3L benchmark uses a lower resolution (320x240) in order to study the impact of changing the resolution.

Frames The total number of frames in each test.

Avg. triangles The average number of triangles sent to the rasterizer per frame.

Avg. processed triangles The average number of triangles per frame that remained after backface culling, i.e., the triangles that remained after eliminating the triangles that are invisible because they are facing backwards from the observer's viewpoint.

Avg. area The average number, per frame, of fragments/pixels after scan conversion.

Texture size The total size of all textures per workload. This quantity gives an indication of the amount of texture memory required.

Maximum triangles The maximum number of triangles that were sent for one frame. Because most 3D graphics accelerators implement only rasterization, this statistic is an approximation of the bandwidth required

for geometry information, since triangles need to be transferred from the CPU to the accelerator via a system bus. We assume that triangles are represented individually. Sharing vertices between adjacent triangles allows to reduce the bus bandwidth required. This quantity also determines the throughput required in order to achieve real-time frame rates. We remark that the maximum number rather than the average number of triangles per frame determines the required bandwidth and throughput.

Maximum processed triangles per frame The maximum number of triangles that remained after backface culling over all frames.

Maximum area per frame The maximum number of fragments after scan conversion, over all frames.

Several observations can be drawn from Table 3.1. First, it can be observed from the columns labeled "Received triangles" that the scenes generated by Tux and Dino have a relatively low complexity, that Q3, ANL, and Graz consist of medium complexity scenes, and that AW produces the most complex scenes by far. Second, backface culling is effective in eliminating invisible triangles. It eliminates approximately 30% of all triangles in the Q3 benchmarks, 24% in Graz, and more than half (55%) of all triangles in AW. Backface culling is not enabled in the ANL and Dino workloads. If we consider the largest number of triangles remaining after backface culling (14236 for ANL) and assume that each triangle is represented individually and requires 28 bytes (xyz coordinates, 4 bytes each, rgb for color and alpha for transparency, 1 byte each, and uvw texture coordinates, 4 bytes each) for each of its vertices, the required bus bandwidth is approximately 1.2MB/frame or 35.9MB/s to render 30 frames per second. Finally, we remark that the largest amount of texture memory is required by the Q3 and Tux benchmarks, and that the other benchmarks require a relatively small amount of texture memory.

3.4.2 Detailed Workload Statistics

An important aspect for 3D graphics benchmarking is to determine possible bottlenecks in a 3D graphics environment since the 3D graphics environment has pipeline structure and different parts of the pipeline can be implemented on separate computing resources such as general purpose processors or graphics accelerators. Balancing the load on the resources is an important decision. Bottlenecks in the TnL part of the pipeline can be generated by applications that have a large number of primitives, i.e. substantial geometry computation

load, where each primitive has a small size, i.e. reduced impact on the rasterization part of the pipeline, while bottlenecks in the rasterization part are usually generated by *fill intensive* applications that are using a small number of primitives where each primitive covers a substantial part of the scene. An easy way to determine if an application is for instance rasterization intensive is to remove the rasterization part from the graphics pipeline and determine the speed up.

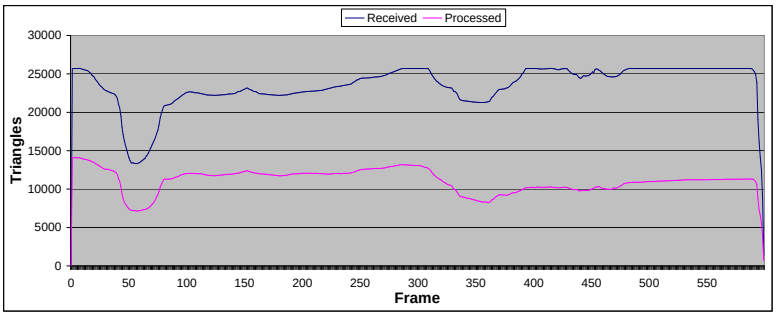
The components of the GraalBench were also chosen to stress various parts of the pipeline. For instance the *AW* and *DIN* components generate an almost constant number of primitives while the generated area varies substantially across frames, thus in these scenarios the T&L part of the pipeline has a virtually constant load while the rasterization part has a variable load. This behavior, depicted in Figures 3.3 and 3.4, can emphasize the role of the rasterization part of the pipeline. The number of triangles **received** gives an indication of the triangles that have to be transformed and lit, while the number of triangles **processed** gives an indication of the triangles that were sent to the rasterization stage after clipping and culling. Other components, e.g., Tux and GRA, as depicted in Figures 3.5 and 3.6, generate a variable number of triangles while the generated area is almost constant for long frame sequences, thus they can be used to profile bottlenecks in the T&L part of the graphics pipeline. The remaining components *Q3* and *ANL*, as depicted in Figures 3.7 and 3.8 provide a balanced load between the T&L and rasterization stages.

Another important aspect beside the variation of the workload for a certain pipeline stage is also the stress strength of the various workload. For convenience, in Table 3.2 is also presented a rough view of the stress variation and stress strength along the 3D graphics pipeline. The stress variation represents how much varies a workload from one frame to another, while the stress strength represents the load generated by each workload.

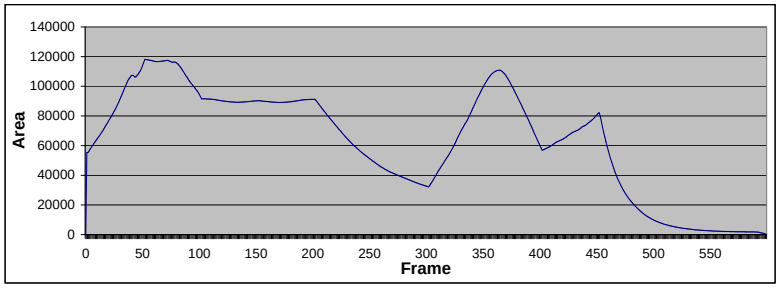
On the proposed benchmarks we determined that, for a software implementation, the most computationally intensive part of the graphics pipeline is the rasterization part. This is the reason for which only the rasterizer stage was additionally studied.

The units for which we further gathered results were described in Chapter 2 and the relation among various units is depicted in Figure 2.3 on page 23.

Triangle Setup, EdgeWalk, and Span Interpolation: We used the same algorithm as employed in the Mesa library rasterizer. We remark that the number of processed triangles in Table 3.3 is smaller than the number of processed



(a) AW Triangles

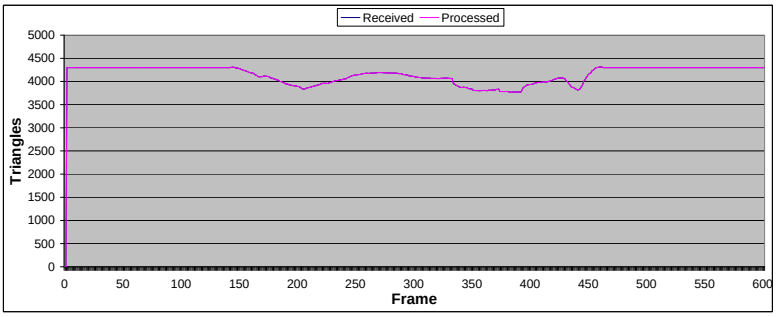


(b) AW Area

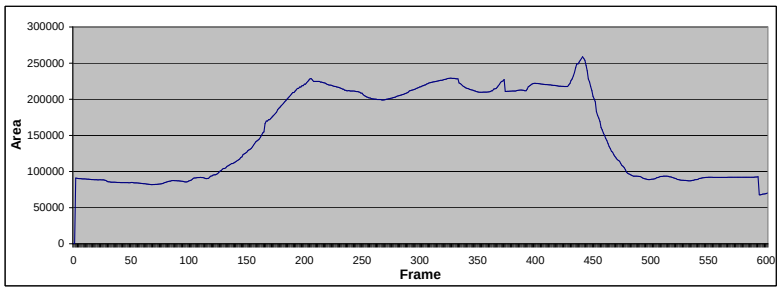
Figure 3.3: Triangle and area statistics for the AW component

Bench.	**T&L**		**Rasterization**	
	Var.	Str.	Var.	Str.
Q3L	med	med	med	med
Q3H	med	med	med	high
Tux	med	low	low	med
AW	low	high	high	low
ANL	high	med	med	med
GRA	high	med	med	low
DIN	low	med	med	low

Table 3.2: Stress Variation and Stress Strength on various stages of the 3D Graphics Pipeline



(a) DIN Triangles

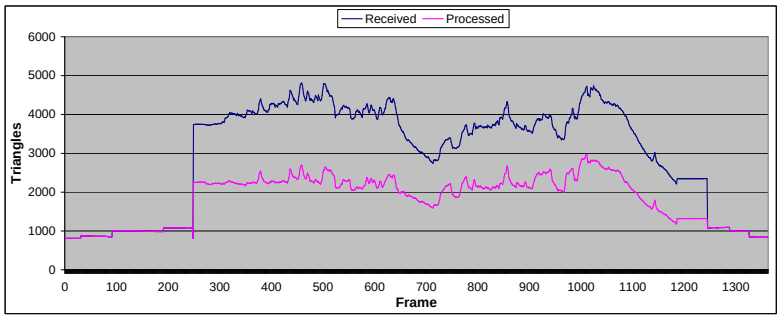


(b) DIN Area

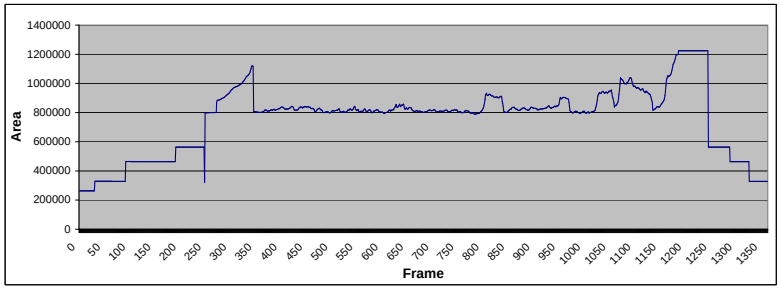
Figure 3.4: Triangle and area statistics for the DIN component

triangles in Table 3.1 because some triangles were small and discarded using supplementary tests, such as a small triangle filter. The average number of processed triangles is the lowest for Tux, medium for Q3 and VRML components, and substantially higher for AW considering that the number of triangles for the AW and VRML components were generated in approximatively half as many frames as the number of frames in Q3 and Tux. The numbers of generated spans and fragments also give an indication of the processing power required at the EdgeWalk and Span Interpolation units. The AW benchmark generates on average only 4 spans per triangle and approximately 2 fragments per span. These results show that the benchmark that could create a pipeline bottleneck in these units is the AW benchmark since it has small triangles (small impact on the rest of the pipeline) and it has the largest number of triangles (that are processed at the Triangle Setup).

Clear Unit: As can be seen in Table 3.4, the Q3 benchmark uses only depth buffer clearing, except for one initial color buffer clear. Q3 exploits the observation that all pixels from a scene are going to be filled at least once so there is



(a) Tux Triangles



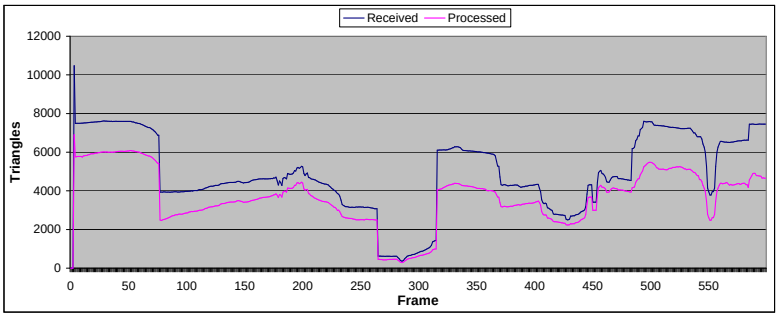
(b) Tux Area

Figure 3.5: Triangle and area statistics for the Tux components

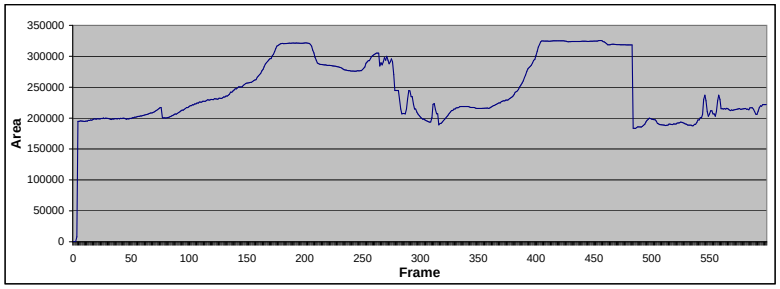
	Q3L	**Q3H**	**Tux**	**AW**
Triangles processed	4,147k	4,430k	2,425k	5,537k
Generated spans	58,837k	117,253k	27,928k	20,288k
Generated fragments (frags.)	581,887k	2,306,487k	1,037,222k	38,044k

	ANL	**GRA**	**DIN**
Triangles processed	2,528k	1,992k	2,487k
Generated spans	66,806k	14,419k	23,901k
Generated fragments (frags.)	466,344k	146,604k	91,824k

Table 3.3: Triangle Setup, Edge Walk, and Span Interpolation statistics



(a) GRA Triangles



(b) GRA Area

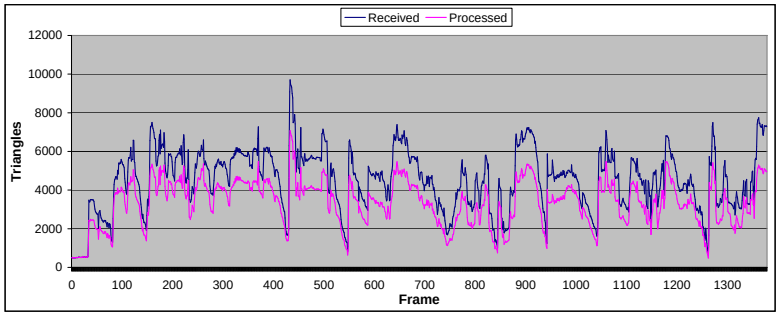
Figure 3.6: Triangle and area statistics for the GRA component

no need to clear the color buffer. The other benchmarks have an equal number of depth and color buffer clears. Although the clear function is called a relatively small number of times, the number of cleared pixels can be as high as 20% of the pixels generated by the rasterizer. This implies that this unit should be optimized for long write burst to the graphics memory.

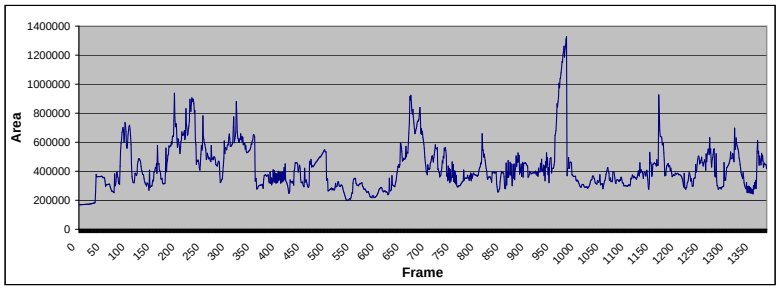
Texture Unit: Depending on the texturing filter chosen, the texture color is obtained by a direct look-up in a texture map or by a linear interpolation between several colors (up to 8 colors in the case of trilinear interpolation).

The results obtained for the texture unit are depicted in Figure 3.9. The Q3 and VV benchmarks used the texture unit for all fragments, while the Tux and AW benchmarks used texturing for 75% and 90% of the fragments respectively.

This unit is the most computationally intensive unit of a rasterizer and can easily become a pipeline bottleneck, thus it should be highly optimized for speed. Beside requiring high computational power, this unit also requires a large number of accesses to the texture memory . However, due to high spatial



(a) Q3L Triangles



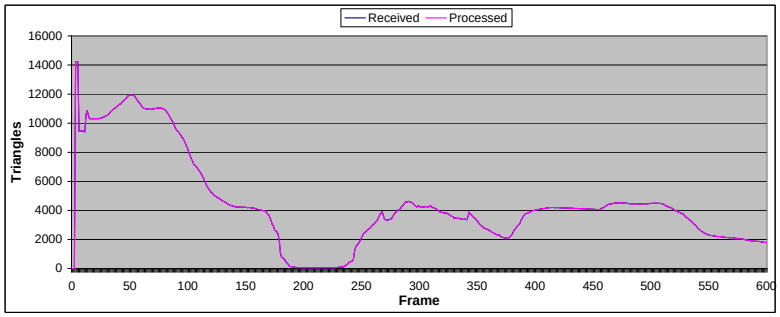
(b) Q3L Area

Figure 3.7: Triangle and area statistics for the Q3L component

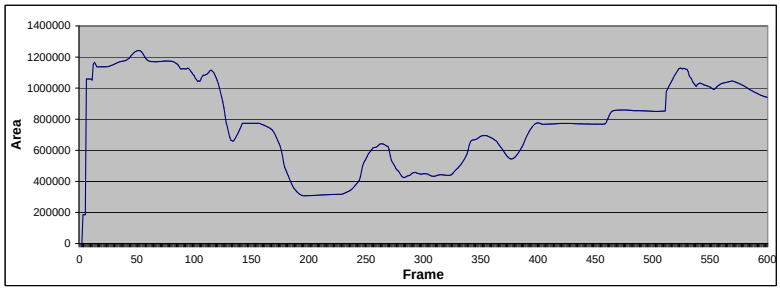
	Q3L	Q3H	Tux	AW
Clear depth calls	5,470	5,470	1,363	603
Clear color calls	1	1	1,363	604
Clear depth pixels	105,821k	423,284k	418,714k	185,242k
Clear color pixels	76,800	307,200	418,714k	185,549k

	ANL	GRA	DIN
Clear depth calls	601	600	602
Clear color calls	601	600	602
Clear depth pixels	184,320k	184,013k	189,934k
Clear color pixels	184,320k	184,013k	189,934k

Table 3.4: Clear statistics



(a) ANL Triangles



(b) ANL Area

Figure 3.8: Triangle and area statistics for the ANL component

locality, even by adding a small texture cache, the traffic from the off-chip texture memory to the texture unit can be substantially reduced[10].

Fog Unit: Only the Tux benchmark uses the fog unit and it was enabled for 84% of the fragments. From the three types of fog (linear, exponential, and squared exponential) only linear was used. The results suggest that for these low-end applications the fog unit is seldomly used and that it might be implemented using slower components or that this unit can be implemented off the critical path.

Scissor Unit: Only Q3 employed scissoring. All incoming fragments were processed and passed the test, so even in this case the test is redundant since no fragments were rejected. Normally, the scissor unit is used to restrict the drawing process to a certain rectangular region of the image space. In Q3 this unit, besides being always enabled for the whole size of the image space (to clip primitives outside it), in some cases it is also used to clear the depth

component of a specific region of the image so that certain objects (interface objects) will always be in front of other objects (normal scene). Even though it might be used intensively by some applications, this unit performs simple computations, such as comparisons, and performs no memory accesses so it does not require substantial computational power.

Alpha Unit: This unit was used only in Q3 and Tux. Furthermore, Q3 used the alpha unit only for a very small number of fragments (0.03%). The only comparison function used was *Greater or Equal*. However, this is not a significant property since the other comparison functions (modes) do not require a substantial amount of extra hardware to be implemented. The number of passed fragments could not be determined since the texturing unit of our graphics simulator is not yet complete, and the alpha test depends on the alpha component that can be modified by the texture processing unit. However, this unit is used significantly only for the Tux benchmark so this is the only benchmark that could have produced different results. Furthermore, the propagated error for the results we obtained can be at most 7.8% since 92.2% of the fragments generated by Tux bypassed this unit. We, therefore, assumed that all fragments passed the alpha test. This corresponds to the worst case. Since this unit is seldomly used, it could be implemented using a more conservative strategy toward allocated resources.

Depth Unit: This unit was used intensively by all benchmarks as can be seen in Table 3.5. While the Tux, AW, and VRML benchmarks write almost all fragments that passed the depth test to the depth buffer, the Q3 benchmark writes to the depth buffer only 36% of the fragments that passed the test. This is expected since Q3 uses multiple steps to apply textures to primitives and so it does not need to write to the depth buffer at each step. This unit should definitely be implemented in an aggressive manner with respect to throughput (processing power) and latency, since for instance the depth buffer read/write operations used at this unit are quite expensive.

Blending Unit: As depicted in Figure 3.9, this unit is used only by the Q3 and Tux benchmarks. The AW and VRML benchmarks do not use this unit since they use only single textured primitives and all blending operations are performed at the texturing stage. Q3, on the other hand, uses a variety of blending modes, while Tux employs only a very common blending mode (source = incoming pixel alpha and dest = 1 - incoming pixel alpha). An explanation

	Q3L	Q3H	Tux	AW
Incoming frags.	581,887k	2,306,487k	1,037,222k	38,044k
Processed frags.	578,345k	2,292,357k	512,618k	38,044k
Passed frags.	461,045k	1,822,735k	473,738k	35,037k
Frags. written to the depth buffer	166,624k	666,633k	462,520k	35,037k

	ANL	GRA	DIN
Incoming frags.	466,344k	146,604k	91,824k
Processed frags.	466,344k	146,604k	91,824k
Passed frags.	281,684k	137,109k	73,268k
Frags. written to the depth buffer	281,684k	137,109k	73,268k

Table 3.5: Depth statistics

why Tux manages to use only this mode is that Tux uses the alpha test instead of multiple blending modes. Alpha tests are supposed to be less computationally intensive than blending operations since there is only one comparison per fragment, while the blending unit performs up to 8 multiplications and 4 additions per fragment. Based on its usage and computational power required, the implementation of this unit should be tuned toward performance.

Unused Units: The LogicOp, Stencil and Color Sum units are not used by any benchmark. The dithering unit is used only by the AW benchmark (for all fragments that passed the blending stage). Since these units are expected to be hardly used their implementation could be tuned toward low-power efficiency.

3.4.3 Architectural Implications Based on Unit Usage

In this section the usage of each unit for the selected benchmarks is presented. The statistics are gathered separately for each benchmark. Figure 3.9 breaks down the number of fragments received by each unit into fragments that bypassed the unit, fragments that were processed by the unit and passed the test, and fragments that failed the test. All values are normalized to the number of fragments generated by the Span Interpolation unit.

From Figure 3.9 it can be seen that the Q3 benchmark is quite scalable and the results obtained for the low resolution profile (Q3L) are similar with the

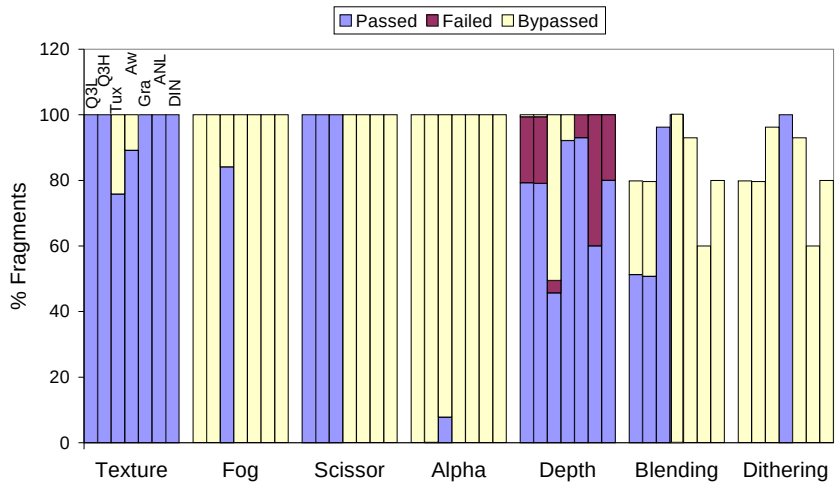


Figure 3.9: Unit usage

results obtained for the high resolution profile (Q3H). The Q3 benchmark can be characterized as an application that uses textures for most of its primitives. The Tux component is also using textures for more than 70% of its primitives, and it also uses the fog unit. The AW component does not use the scissor test as the others are doing and also it has no pixels rejected at the depth test. Another difference from the previous components is that AW is also using the dithering mechanism in order to improve the image quality on displays with a low color depth. Some architectural implications based on the units usage are: Some of the units such as Color sum, LogicOp and Stencil were not used, so they might not be implemented in hardware. Some units such as Fog and Alpha were less used and they can be also be implemented outside the critical path. The Depth and Blending units should be hardwired units and tuned toward performance. The texture unit should be definitely focused upon for a high performance implementation since, due to the processing power required, it can easily become a bottleneck for the graphics pipeline.

3.5 Conclusions

Although high-end 3D graphics benchmarks have been available for some time, there are no benchmark suites dedicated to embedded 3D graphics accelerators. In this chapter we have described a set of relevant applications

for embedded 3D graphics accelerators performance evaluation. Also one of the objectives of this chapter was to determine what features of 3D graphics implementations are used in relevant 3D graphics applications. We have also identified a number of units from the 3D graphics pipeline which are intensively used such as the texture and the depth units, while for instance, stencil, fog, and dithering units being rarely used.

In Chapter 4 we analyze the bandwidth requirements of a conventional and a tile-based renderer using the benchmarks described in this chapter.

Chapter 4

Memory Bandwidth Requirements of Tile-Based Rendering

External memory accesses are a major source of power consumption [16, 31]. They often dissipate more energy than the datapaths and the control units. Moreover, accesses to external memories can be several times slower than accesses to on-chip memories and they can significantly limit the performance of graphics accelerators. Tile-based rendering reduces the amount of external data traffic and, hence, the power consumption by using small, on-chip buffers, so that only the pixels visible in the final scene need to be stored in the external framebuffer. On the other hand, however, a tile-based renderer may require that each triangle is sent to the graphics accelerator more than once, since a triangle might overlap more than one tile. Moreover, given that the amount of chip area is limited, very little area can be devoted to local memory. So, we are faced with two opposite goals: reduce the amount of external memory traffic as much as possible while, at the same time, using as little internal memory as possible.

In this chapter we examine the memory bandwidth requirements for tile-based 3D graphics accelerators. First, we examine how the amount of external data traffic varies with the tile size. The results show that a tile size of 32×32 pixels yields the best trade-off between the data traffic volume and the amount of area dedicated to on-chip buffers. By increasing the tile size beyond 32×32 pixels, the amount of external data traffic is only marginally reduced. Also, decreasing the tile size under 32×32 pixels, increases the external data traffic substantially. Second, we measure how much external data traffic is saved by a tile-based renderer compared to a traditional renderer. The results show that the tile-based architecture reduces the total amount of external data traffic by

a factor of 1.96 compared to the traditional architecture.

This chapter is organized as follows. After discussing related work in Section 4.1, we describe in more detail the data traffic components for conventional and tile-based rendering models in Section 4.2. In Section 4.3, the results of our experiments are presented. First, we determine the influence of the tile size on the amount of external data traffic. Second, we measure how much external data traffic is saved by employing a tile-based architecture instead of a conventional architecture. Conclusions and given in Section 4.4.

4.1 Related Work

Low-power tile-based architectures were proposed in [57, 39], but no measurements of the total required bandwidth were presented. For instance, in [57] it is only shown that the tile-based approach reduces the traffic between the renderer and the external memory for various scene sizes. However, there is no indication of the amount of the data traffic increase between the CPU and the renderer for tile-based architectures as compared with a traditional architecture.

Other papers discussing tile rendering (e.g., [21]) are mainly concerned by the *overlap* (the number of tiles that a primitive covers) of triangles with respect to tile size. Only the traffic between the CPU or main memory to the accelerator was considered. We consider the overall data traffic, i.e., not only the traffic from the CPU or main memory to the accelerator but also the traffic between the accelerator and the framebuffer.

4.2 Data Traffic Components

In this section we analyze in more detail the organization of a traditional and a tile-based renderers initially described in Chapter 1 and briefly discuss the factors that contribute to the amount of external data traffic for each rendering method.

Figures 4.1 and 4.2 depict the basic organization of a conventional and a tile-based renderer. These figures have been show before in Section 1.1.2 but now the edges are labeled with external data traffic components.

We now briefly describe the amount of external data traffic generated by each method. As depicted in these figures, the data traffic between the rasterizer ac-

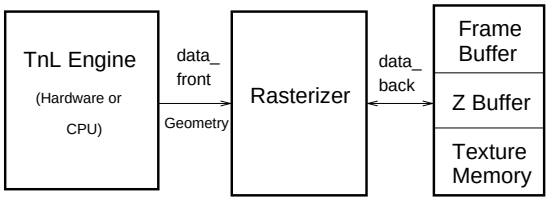


Figure 4.1: Organization of a traditional renderer.

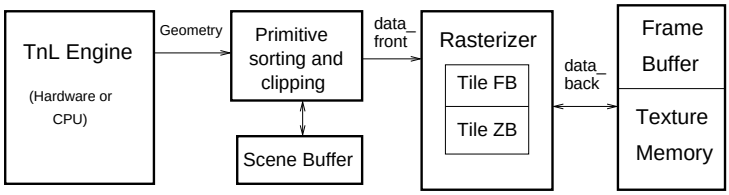


Figure 4.2: Organization of a tile-based renderer.

celerator and other components of the system such as the CPU, main memory, and the framebuffer can be divided into two categories:

1. The data sent from the host CPU (or main memory) to the accelerator. It consists of geometrical data needed to describe the primitives, texture data, and changes to the state of the rasterizer such as enable/disable depth test, change texture wrapping mode, etc. We refer to this component as *data_front*.

2. The data transferred between the accelerator and dedicated graphics memory or memories. Since the framebuffer, z buffer, and texture memory are too large to be placed on-chip, they must be allocated off-chip. This component will be referred to as *data_back*.

The *data_front* term is usually dominated by the amount of geometrical data needed to describe the primitives. The amount of texture data is, in the long run, negligible. We remark that with texture data we mean here the traffic needed to copy the texture images to the dedicated texture memory. This is necessary because the application may reuse the texture space after it has passed a pointer to this space to the rasterizer. It does not include the traffic needed to perform texturing. This component is included in the *data_back* term.

In a traditional renderer, the amount of geometrical data is proportional to the

number of primitives. In a tile-based renderer, however, each primitive might have to be sent to the rasterizer several times. In particular, if a primitive overlaps n tiles, it needs to be transmitted to the rasterizer n times. Thus, in a tile-based renderer, the *data_front* component is affected significantly by the amount of ***overlap***, which is the average number of tiles covered by each primitive.

Therefore, a tile-based renderer actually increases the *data_front* component compared to a conventional renderer. The *data_back* term, however, is significantly reduced by a tile-based renderer. Because the color components and the z values of the fragments belonging to a particular tile can be kept in small, on-chip buffers, only pixels visible in the final image have to be written to the external framebuffer. Furthermore, provided that the application clears the z buffer when it starts to render a new frame, the traffic between the rasterizer and the external z buffer is eliminated completely. In a traditional renderer, on the other hand, many fragments might be written to the external framebuffer which are not visible in the final image because they are obscured by other pixels. Thus, in a conventional renderer, the *data_back* component is affected significantly by the amount of ***overdraw***, which can be defined as the number of fragments written to an external buffer divided by the image size.

Concluding, while a tile-based renderer produces more external traffic for geometrical data than a traditional renderer (depending on the amount of overlap), it generates less traffic between the rasterizer and the off-chip frame and z buffers (depending on the amount of overdraw).

4.3 Experimental Results

In this section the experimental results are presented. First, in Section 4.3.1, the benchmarks, tools, and some simulation parameters are described. Thereafter, in Section 4.3.2 we study how the amount of external data traffic varies with the tile size. Finally, in Section 4.3.3, we compare the total amount of off-chip memory traffic produced by a tile-based renderer to the amount of traffic generated by a conventional renderer.

4.3.1 Experimental Setup

In order to compare the traditional and tile-based architectures we used 6 of the 7 components of the GraalBench benchmark suite described in Chapter 3: Q3H, Tux, Aw, ANL, GRA, and DIN. The Q3L profile from the GraalBench

benchmark suite has a different window size than the other components, and it was not used.

The statistics were gathered using several tools. First, we used our OpenGL tracer described in Section 3.3.3 to generate the benchmarks traces which were fed to the Mesa library. The Mesa library performed primitive back-face culling and generated lists of remaining primitives that were sent to our accelerator simulator. For the tile-based architecture we used tile sizes of $\{16, 32, 64\} \times \{16, 32, 64\}$ pixels, and the window size was 640×480 pixels for all benchmark suite components.

As will be shown in Chapter 7, because texturing exhibits high data locality and because small, direct-mapped caches do not require a large amount of area nor consume a significant amount of power, we have employed a tiny (256-byte) direct-mapped, on-chip texture cache. We have used our own trace-driven cache simulator to measure the miss ratio of the texture cache.

In order to simulate the 9 possible tile size configurations for all workloads on our rasterizer simulator in an acceptable time interval (several weeks), we have rendered approximatively 60 frames from each workload evenly distributed across the workload. By using this method, we ensured that for each workload, possible workload variations were taken into account. Moreover, since we employed a small texture cache, the effects of inter-frame locality were considered negligible. For the skipped frames, however, we have not skipped the state change information so that the appropriate state information was committed to the renderer before each frame was rendered.

4.3.2 Tile Size Versus External Data Traffic

In this section we determine how the amount of external data traffic varies with the tile size. Since the tile size determines the size of the local frame and z buffers, and because the amount of chip area is severely limited, we want to employ the smallest tile size possible while, at the same time, reducing the amount of off-chip data traffic as much as possible.

As a first indication of an appropriate tile size, Figure 4.3 depicts the cumulative percentage of triangles up to a certain size. It can be seen that by far the most (93%) triangles are smaller than 1024 pixels. Very few triangles (7%) are larger than 1024 pixels. This indicates that more than 93% triangles might fit in a tile size of 1024 (e.g., 32×32) pixels. However, even if most triangles are smaller than say, 32×32 they still can overlap multiple tiles if the tile size is 32×32. Therefore, a better indication of an appropriate tile size is the number

of triangles sent to the rasterizer.

Table 4.1 depicts the number of triangles transferred to the rasterizer for various tile sizes. As explained in Section 4.2, this data usually dominates the *data_front* component of the total external data. The last row shows the number of triangles transferred if the tile size is equal to the window size. The overlap factor for a specific tile size can, therefore, be obtained by dividing the number of triangles transferred for that tile size by the number given in the last row.

Obviously, if the tile size increases, the number of triangles transferred to the rasterizer decreases, since there is less overlap. However, as remarked before, it is important to use as little internal memory as possible and, therefore, a trade-off needs to be made. It can be seen that using a tile size of less than 32×32 can increase the number of triangles transferred significantly. For example, if we employ a tile size of 16×16 instead of 32×32, the amount of geometrical data sent to the rasterizer increases by a factor of 2.02 for the Q3H benchmark and by a factor of 1.97 for the Tux benchmark. On average, using the geometric mean, a tile size of 16×16 increases the number of triangles sent by a factor of 1.62 compared to 32×32 tiles. On the other hand, employing tiles larger than 32×32 reduces the amount of geometrical data only marginally. For example, the geometric mean of the reduction resulting from employing 64×64 tiles instead of 32×32 tiles is 1.35. This indicates that a tile size of 32×32 is the best trade-off between the number of triangles sent to the rasterizer and the size of the internal buffers. Moreover, the number of gates required to implement on-chip tile buffers lager than 32×32 can easily exceed gates budgets for current low-power devices. For example, for a 32×32 elements tile, where each element has 32 bits allocated for RGBA color, 24 bits allocated for depth, and 8 bits for stencil, the number of bits required to implement the tile is 64Kbits. In a SRAM implementation (6 gates per bit), this represents 384k gates.

4.3.3 Tile-Based Versus Conventional Rendering

In this section we measure the total amount of external data traffic produced by a tile-based renderer for a tile size of 32×32 and compare this to the amount of off-chip memory traffic generated by a conventional renderer.

Figure 4.4(a) presents the amount of data traffic sent from the CPU to the rasterizer (the *data_front* component of the total traffic) for the tile-based as well as the conventional renderer. It also breaks down the *data_front* term into state change data and geometrical data. As expected, the tile-based architecture

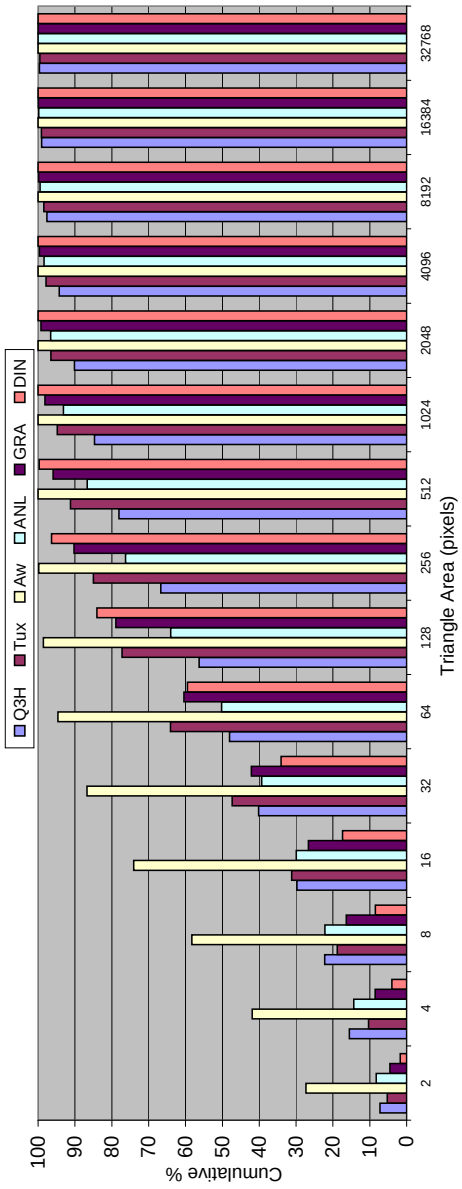


Figure 4.3: Triangles histogram. This figure shows the cumulative percentage of triangles having an area lower than a defined size.

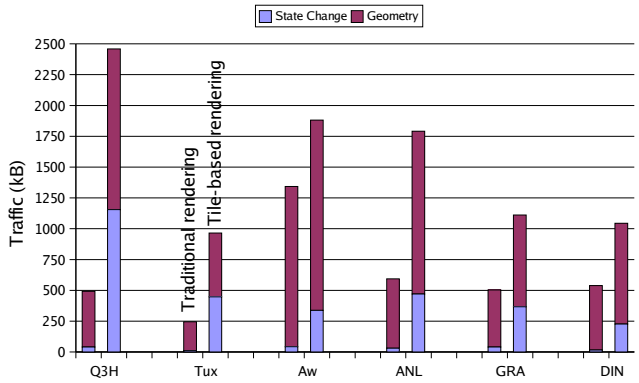
Table 4.1: Number of triangles transferred as a function of the tile size for each benchmark.

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

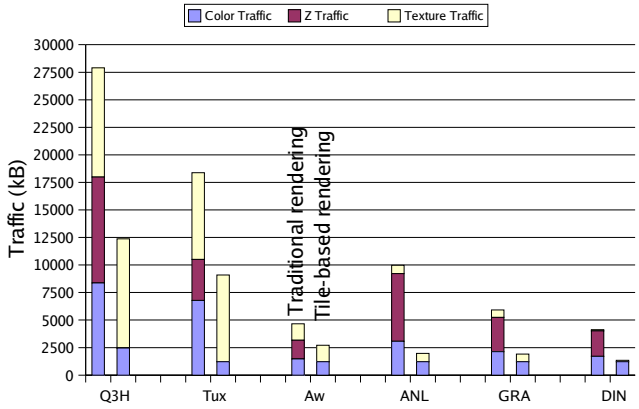
generates more *data_front* traffic than the traditional architecture. On average, using the geometric mean, the tile-based architecture increases the amount of *data_front* traffic by a factor of 2.66 compared to the conventional renderer. The figure also shows that the amount of *data_front* traffic is dominated by the geometrical data and that the increase is due for a large part to the increase in the amount of geometrical data transferred. However, the amount of state change traffic is also significantly increased and cannot be neglected. More details of the state change traffic and methods to reduce it are presented in Chapter 6.

Figure 4.4(b) depicts the amount of data transferred between the rasterizer and the off-chip color and z buffers and texture memory (the *data_back* term). Furthermore, the *data_back* component has been split into data transferred from/to the frame buffer (color traffic), z buffer, and texture memory. Due to the fact that our rasterizer simulator is much slower when rendering larger tiles due to the fact that some of the used data types, when scaled, can no longer be mapped directly to native CPU data types, and the texture miss ratio changed only marginally for tiles with sizes from 16×16 up to 64×64, we have approximated the texture traffic for a traditional renderer with the texture traffic generated by a 64×64 tile-based rasterizer.

On average, the tile-based architecture reduces the *data_back* traffic by a factor of 2.71 compared to the traditional renderer (geometric mean). Furthermore,



(a) *data_front*



(b) *data_back*

Figure 4.4: The front and back data traffic components

for the conventional architecture the *data_back* traffic is dominated by the traffic between the rasterizer and the frame/z buffers, whereas in a tile-based renderer this traffic is eliminated almost completely. For a tile-based renderer, the texture traffic is the largest component of the *data_back* traffic.

Finally, Figure 4.5 depicts the total amount of external data traffic produced by the conventional and the tile-based renderer. The total traffic has been divided into *data_front* and *data_back* traffic. It can be seen that since the amount of *data_back* traffic is much larger than the amount of *data_front* traffic, the tile-based architecture reduces the total amount of external traffic significantly. The geometric mean of the traffic reductions over all benchmarks is a factor of 1.96. However, the advantage of tile-based rendering is workload dependent. For example, for the Aw benchmark the reduction is only 23.4%. This can be

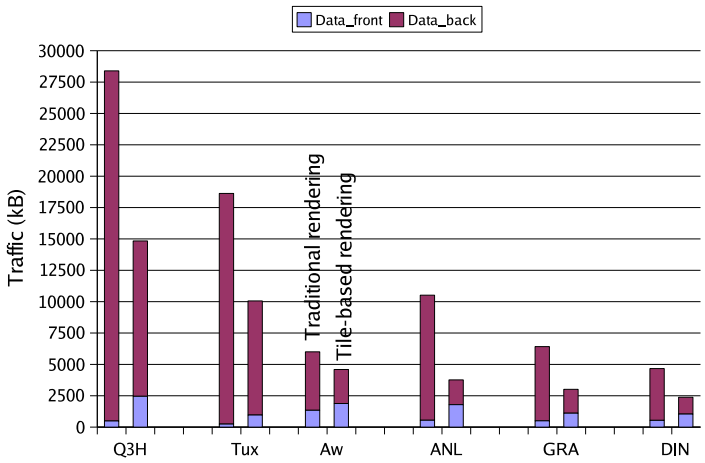


Figure 4.5: Total external data transferred (kbytes) per frame for a tile-based and a traditional architecture

explained as follows. Since tile-based rendering increases the *data_front* component (depending on the overlap factor) but decreases the *data_back* component (depending on the overdraw factor), it is more suitable than traditional rendering for workloads with low overlap and high overdraw. On the other hand, for workloads with high overlap and low overdraw, tile-based rendering does not reduce the total amount of external data traffic significantly.

Since some of the workloads from our benchmark suite do not exhibit high overdraw, the results obtained for the tile-based renderer for these workloads are not significantly better than traditional rendering.

4.4 Conclusions

In this chapter we have presented a comparison of the total amount of external data traffic required by traditional and tile-based renderers. Besides improving performance, reducing the amount of external data traffic also improves energy consumption,since external memory accesses are a major source of power consumption.

For tile-based renderers, based on the total data traffic variation with respect to the on-chip memory (tile size), a tile size of 32×32 pixels was found to yield the best trade-off between the amount of on-chip memory and the amount of external data traffic. We have also shown that tile-based rendering reduces

the total amount of external traffic due to the considerable data traffic reduction between the accelerator and the off-chip memory while maintaining an acceptable increase in data traffic between the CPU and the renderer. Considering that external memory accesses consume a significant amount of power, this indicates that tile-based rendering might be a suitable technique for low-power embedded 3D graphics implementations. We mention, however, that the reduction in bandwidth of tile-based rendering when compared to traditional rendering depends significantly on the workload used. For workloads with a high overlap factor and low overdraw, tile-based rendering does not perform significantly better than traditional rendering, while for workloads with a low overlap factor and high overdraw, the tile-based rendering is more suitable than traditional rendering.

In Chapter 5 we investigate the memory bandwidth required for sorting the geometrical primitives into bins corresponding to the tiles. In Chapter 6 we consider ways to reduce the amount of state change information that needs to be sent to a tile-based accelerator. As shown in Figure 4.4(a), although this data is generally less than the amount of geometry data, it cannot be neglected.

Finally, in Chapter 7, a low-cost technique to reduce texture traffic is considered. Figure 4.4(b) shows that for some workloads this external traffic component is also significant.

Chapter 5

Scene Management Algorithms for Tile-Based Architectures

In the previous chapter we have shown that tile-based rendering architectures appear to be promising for low-power implementation because they reduce the amount of communication between the accelerator and the frame and z buffers. However, the bandwidth required to place the primitives into bins was ignored because we assumed that this task is performed by the CPU, but in order to achieve real-time performance, this task needs to be performed fast. Furthermore, in low-cost systems the amount of memory available to store the primitives may be limited, so the memory requirements of the algorithm employed to sort the primitives is an important metric too.

In this chapter we make three contributions. First, several scene management algorithms for sorting the primitives into bins are presented and their computational complexity and memory requirements are evaluated. Experimental results show that various trade-offs between the computational power and required memory can be achieved. Second, we present an exact test that determines if a triangle overlaps a tile. This test has been proposed before but in a different context and we adapted it so that no coverage mask is needed to determine if a triangle intersects a tile. Third, "dynamic" versions of the commonly employed bounding box test are presented in which the comparisons are performed in an order that depends on the position of the current tile.

This chapter is organized as follows. Section 5.1 briefly describes related work. In Section 5.2 we describe the triangle to tile overlap tests we considered for our implementation. The scene management algorithms are presented in Section 5.3. Experimental results for scene management algorithms are presented

in Section 5.4. Additional bounding box test optimizations and results are provided in Section 5.5, and conclusions are given in Section 5.6.

5.1 Related Work

In [60], Samanta and Funkhouser studied the impact of sorting the primitives to tiles before or after screen space conversion. They have also focused on the efficiency of grouping primitive and testing their bounding box overlap with tiles.

These studies are, however, not very related to our study since we consider a low-power architecture in which the tiles are rendered sequentially one by one. In such architectures, sorting the primitives to tiles before space conversion would increase the computational power (due to possible multiple space conversion per primitive) without any benefit for balancing the workload for the following stages.

Even though tile-based rendering has been used in power-aware architectures [57, 39], no details have been provided on how the primitives are sorted into bins corresponding to the tiles. Furthermore, the only triangle-to-tile overlap test we could find in literature as implemented in hardware is thebounding box test [57, 39, 21, 17]. The reason for this is given by the fact that the bounding box test, even if not accurate, is much faster when compared with algorithms that perform exact bounding box testing by using for instance triangle scan conversion with a tile size granularity.

Other exact (but still computationally expensive) algorithm to determine exact triangle to tile overlap is given in [54]. This algorithm determines if a triangle overlaps a tile by performing a series of tests such as: a triangle vertex is inside the tile or a triangle corner is inside the tile or a triangle edge intersects the tile.

On the other hand, algorithms developed for other purposes (e.g., anti-aliasing [62] or collision detection [25, 8]) can be adapted for faster, exact primitive to tile sorting.

5.2 Overlap Tests

In this section we describe two triangle to tile overlap tests. Our contribution consist of introducing an overlapping test based on [62], but with the difference that our test does not require any coverage mask.

We remark that the bounding box test performs only a partial classification. This means that it may yield a positive result even if a triangle does not intersect a tile. This conservative approach is preferable to predicting that a triangle and a tile do not overlap while in reality they do, because then "holes" may appear in the rendered scene if triangles have been discarded incorrectly. As can be expected, in general, more accurate tests are computationally more expensive. This allows various algorithms with different time and memory complexities to be developed (Section 5.3).

5.2.1 Bounding Box Test

The Bounding Box (BBOX) test determines if the axis aligned bounding box of a triangle intersects with the tile. The BBOX test consists of two steps. First, the 2D axis aligned BBOX of the primitive is computed. Thereafter, it is determined if the BBOX intersects the tile. Let a triangle *Tr* be defined by three points p_1, p_2, and p_3, whose x coordinates are given by $p_i.x$ and whose y-coordinates are given by $p_i.y$. Then the BBOX of *Tr* is defined by the tuple $(\textit{BBOX.MinX}, \textit{BBOX.MinY}, \textit{BBOX.MaxX}, \textit{BBOX.MaxY})$ where

$$\begin{aligned} \textit{BBOX.MinX} &= \text{MIN}(p_1.x, p_2.x, p_3.x) \\ \textit{BBOX.MinY} &= \text{MIN}(p_1.y, p_2.y, p_3.y) \\ \textit{BBOX.MaxX} &= \text{MAX}(p_1.x, p_2.x, p_3.x) \\ \textit{BBOX.MaxY} &= \text{MAX}(p_1.y, p_2.y, p_3.y). \end{aligned}$$

Let the tile be given by the tuple $(\textit{T.MinX}, \textit{T.MinY}, \textit{T.MaxX}, \textit{T.MaxY})$. Then a possible implementation of the BBOX test in C is:

```
if (BBOX.MaxX < T.MinX)    /* Test 1 */
   return NoOverlap;

if (BBOX.MinX >= T.MaxX)   /* Test 2 */
   return NoOverlap;

if (BBOX.MaxY < T.MinY)    /* Test 3 */
   return NoOverlap;

if (BBOX.MinY >= T.MaxY)   /* Test 4 */
   return NoOverlap;

return MightOverlap;
```

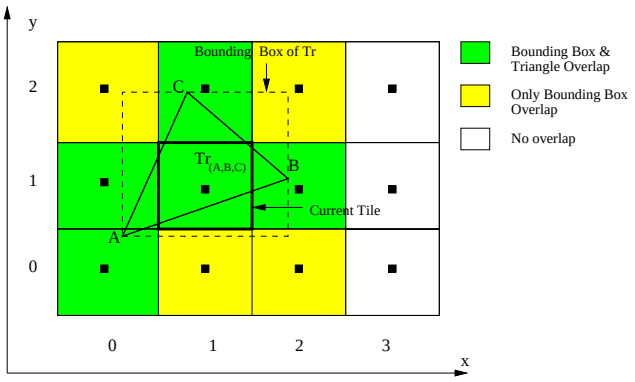


Figure 5.1: Triangle to tile BBOX test

We remark that even if all tests fail then the triangle might overlap the tile but it might also be the case that the BBOX intersects with the tile but the triangle does not. This is illustrated in Fig. 5.1. The yellow tiles intersect with the BBOX but not with the triangle. Thus the BBOX test performs only a partial classification. There are two ways to deal with this situation. Either an additional, exact but more expensive test is performed, or the triangle is sent to the rasterizer in which case no fragments are generated for the triangle if it does not overlap the tile. In practice, if a triangle is small, the BBOX test can be accurate (actually the accuracy of this test depends also on the thinness and orientation of a triangle) while for larger triangles the accuracy might drop significantly. We experimentally determined that, on common workloads, the BBOX test can generate up to 30% false intersections for large triangles.

5.2.2 Linear Edge Function Test (LET)

This test employs *edge functions* [56] to determine if a triangle intersects a tile. Edge functions are normally used to determine if a point is inside a triangle or, for instance, to compute a coverage mask for anti-aliasing[62]. In our case, we extended the equations presented in [62] so that no coverage mask is needed to determine if a triangle intersects a tile.

Consider a 2D vector defined by two points $A(X, Y)$ and $B(X+dX, Y+dY)$, and a line L_{AB} that passes through the two points. The edge function for a certain point (x, y) is defined as:

$$E_{L_{AB}}(x, y) = (x - X) \cdot dY - (y - Y) \cdot dX. \tag{5.1}$$

The edge function can be also be written using an incremental form as:

$$E_{L_{AB}}(x+\delta x, y+\delta y) = E_{L_{AB}}(x,y) + \delta x \cdot dY - \delta y \cdot dX. \tag{5.2}$$

The incremental form can be used to evaluate the edge function for a sequence of points more efficiently. In this case, only for the first point the edge needs to be computed using Equation (5.1) while for the rest of the points the incremental form can be used. The incremental computation of the edge function requires fewer operations than Equation (5.1). Furthermore, commonly $\delta x = 1$ or $\delta y = 1$, in which case, the incremental version requires only one multiplication.

The edge function can be used to determine the position of a point (x, y) relative to the line L_{AB} as follows:

$$\begin{array}{ll} \text{If } E_{L_{AB}}(x,y) > 0 & \text{then the point is to the right of } L_{AB} \\ \text{If } E_{L_{AB}}(x,y) = 0 & \text{then the point is on } L_{AB} \\ \text{If } E_{L_{AB}}(x,y) < 0 & \text{then the point is to the left of } L_{AB} \end{array} \tag{5.3}$$

The conditions that can be used (but are not sufficient) to determine if a counter-clockwise oriented triangle T, defined by three vertices $A(x_A, y_A)$, $B(x_B, y_B)$, $C(x_C, y_C)$, intersects a square S defined by a center point $CS(x_{cs}, y_{cs})$ and having a total width of l are:

$$\begin{array}{l} E_{L_{AB}}(x_{cs}, y_{cs}) \leq \frac{l}{2} \cdot (|x_B - x_A| + |y_B - y_A|) \\ E_{L_{BC}}(x_{cs}, y_{cs}) \leq \frac{l}{2} \cdot (|x_C - x_B| + |y_C - y_B|) \\ E_{L_{CA}}(x_{cs}, y_{cs}) \leq \frac{l}{2} \cdot (|x_A - x_C| + |y_A - y_C|) \end{array} \tag{5.4}$$

Considering that a tile can be regarded as a rectangle R defined by a center point $M(x_m, y_m)$, and a width of w and a height of h, we can transform the rectangle R into a square S with a width of 1 using a normalization (division by (w, h)). By performing the same operation on the vertices of the triangle T, the conditions presented in Equations (5.4), where $l = 1$ can be used to test if the triangle intersects the tile. Because we do not use any masking mechanism as used in [62], the Equations (5.4) are not sufficient to correctly classify the overlap of a triangle with a tile. In order to eliminate false intersections, a bounding box test should be performed first and followed by the Equations (5.4). Figure 5.2 depicts the outcome of the triangle to tile test using linear functions. The LET test is exact as opposed to the BBOX test where triangles might have been classified as overlapping a tile even if there was no real triangle to tile overlap.

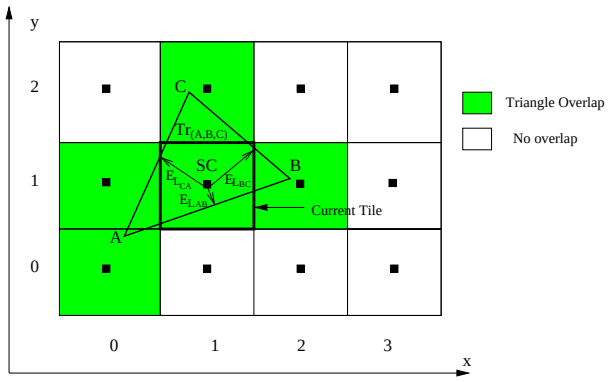


Figure 5.2: Triangle to tile test using linear functions

We remark that there are more tests to accurately classify the triangle to tile overlaps. An example is the Separating Axes Test (SAT) test described in [8, 25], and used in the implementation of the Chromium [40] parallel rendering system. The SAT test is based on the observation that two *convex* objects do not overlap if there exists a line for which their respective projections on the line do not intersect. It is shown in [24] that is enough to consider the normals to the edges of each polygon as projection lines. This method, however, even after aggressive optimizations remained more computationally intensive than the LET method and it was not further employed in our study.

5.3 Scene Management Algorithms

In this section we present several algorithms for sorting primitives into bins corresponding to tiles and determine their computational complexity and memory requirements.

Sorting and sending the primitives involves two steps. First, the primitives are generated and buffered. After that, the primitives are sent to the graphics accelerator in tile-based order. By performing different computations in each step, various implementations with different time and memory complexity are possible. For example, one approach is to sort the primitives while they are buffered. This approach, however, requires a substantial amount of additional memory because primitives generally cover several tiles. Another approach is to leave the primitives unsorted but to sort them while they are sent to the accelerator. This approach requires no additional memory but generally consumes

more time than the previous algorithm. We show that several intermediate approaches are possible as well.

Algorithm DIRECT This algorithm simply scans the whole list of primitives for each tile and sends the primitives that (potentially) overlap the current tile to the rasterizer.

```
for each triangle Tr
    buffer Tr

for each tile T
    for each triangle Tr
        compute bbox of Tr
        if bbox of Tr and T overlap
            send Tr
```

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

$$t_{buf} \cdot \#triangles + t_{bbox\text{-}total} \cdot \#tiles \cdot \#triangles + t_{send} \cdot \#triangles \cdot bbox\_overlap,$$

where t_{buf} is the cost of placing a triangle in the scene buffer, $t_{bbox\text{-}total}$ is the cost of computing the bounding box of a triangle and determining if a bounding box and a tile overlap, t_{send} is the cost of sending a triangle to the rasterizer, and *bbox_overlap* is the overlap factor (i.e., the average number of tiles a triangle covers) if the bounding box test is employed.

The main advantage of algorithm DIRECT is that it requires no memory in addition to the scene buffer. We also remark that here we used the bounding box test, but other tests may also be employed.

Algorithm TWO_STEP In this algorithm the bounding box of each triangle is computed and stored during the buffering stage. This avoids having to recompute the bounding box for each triangle/tile tuple during the sending stage. It requires, however, that the bounding box of each primitive is kept.

```
for each triangle Tr
```

```
    buffer Tr
    compute and store bbox of Tr

for each tile T
    for each triangle Tr
        if bbox of Tr and T overlap
            send Tr
```

The complexity of this algorithm is

$$(t_{buf} + t_{bbox\text{-}compute}) \cdot \#triangles + \\ t_{bbox\text{-}test} \cdot \#tiles \cdot \#triangles + \\ t_{send} \cdot \#triangles \cdot bbox\_overlap,$$

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

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

Algorithm TWO_STEP_LET This algorithm is similar to the TWO_STEP algorithm described above. The difference is that in the second stage a LET overlap test instead of just BBOX is used. Since the LET test contains a BBOX test, the main LET test (Equations 5.4) is applied only to triangles that have passed the BBOX test.

```
for each triangle Tr
    buffer Tr
    compute and store bbox of Tr

for each tile T
    for each triangle Tr
        if LET test indicates Tr and T overlap
            send Tr
```

The complexity of this algorithm is

$$(t_{buf} + t_{bbox\text{-}compute}) \cdot \#triangles +$$
$$t_{bbox\text{-}test} \cdot \#tiles \cdot \#triangles +$$
$$t_{let\text{-}test} \cdot \#triangles \cdot bbox\_overlap +$$
$$t_{send} \cdot \#triangles \cdot let\_overlap,$$

where $t_{let\text{-}test}$ is the cost of testing if a triangle and a tile overlap using LET test, and *let_overlap* is the LET overlap factor (i.e., the average number of tiles covered by a triangle if the LET test is employed).

While the TWO_STEP_LET algorithm takes more time than the TWO_STEP algorithm, the number of triangles sent to the rasterizer by the TWO_STEP_LET algorithm ($\#triangles \cdot let\_overlap$) is lower than or equal to the number of triangles sent to the rasterizer in the TWO_STEP algorithm ($\#triangles \cdot bbox\_overlap$) since the LET test is accurate while the BBOX test is approximative. By sending less triangles to the accelerator this algorithm reduces the computational requirements at the accelerator.

The amount of additional memory required by algorithm TWO_STEP_LET is the same as for the TWO_STEP algorithm.

Algorithm SORT In this algorithm for each tile there is a buffer with pointers to the primitives that overlap the tile according to the BBOX test. For each tile only the primitives that have a pointer in the corresponding tile buffer will be sent to the rasterizer.

```
for each triangle Tr
    buffer Tr
    compute bbox of Tr
    for each tile T that overlaps bbox of Tr
        insert pointer to Tr in the buffer of T

for each tile T
    for each triangle Tr in the buffer of T
        send Tr
```

The complexity of this algorithm is

$$
\begin{aligned}
&(t_{buf} + t_{bbox\text{-}compute}) \cdot \#triangles + \\
&t_{insert} \cdot \#triangles \cdot bbox\_overlap + \\
&t_{tiletrav} \cdot \#tiles + \\
&t_{send} \cdot \#triangles \cdot bbox\_overlap,
\end{aligned}
$$

where t_{insert} is the cost of inserting a pointer to a triangle in the buffer of the tile. There is no need to add a $t_{bbox\text{-}test}$ cost since there is no BBOX test performed (we can determine from the bounding box coordinates in which tiles to insert pointers). The $t_{tiletrav}$ is the cost to traverse a tile.

The amount of additional memory required by the SORT algorithm is $\#triangles \cdot bbox\_overlap \cdot 2 \cdot sizeof(pointer) + \#tiles \cdot 2 \cdot sizeof(pointer)$, where *sizeof(pointer)* denotes the size of a pointer (4 bytes). In our current implementation we use a (preallocated) linked list of pointers to primitives, thus we need to store two pointers for each primitive (one pointing to the primitive and one to the next primitive in the tile). We also use two pointers for each tile (to the first primitive for the tile and the last primitive inserted).

Algorithm SORT_LET This algorithm is similar to the previous algorithm. The difference is that eventually the LET test is used to determine if a triangle overlaps a tile. For each tile only the primitives that have a pointer in the corresponding tile buffer will be sent to the rasterizer.

```
for each triangle Tr
    buffer Tr
    compute bbox of Tr
    for each tile T that overlaps bbox of Tr
        if LET test indicates Tr and T overlap
        insert pointer to Tr in the buffer of T

for each tile T
    for each triangle Tr in the buffer of T
        send Tr.
```

The complexity of this algorithm is

$$
\begin{aligned}
&(t_{buf} + t_{bbox\text{-}compute}) \cdot \#triangles + \\
&t_{let\text{-}test} \cdot \#triangles \cdot bbox\_overlap + \\
&t_{insert} \cdot \#triangles \cdot let\_overlap + \\
&t_{tiletrav} \cdot \#tiles + \\
&t_{send} \cdot \#triangles \cdot let\_overlap,
\end{aligned}
$$

The amount of additional memory required by the SORT_LET algorithm is $\#triangles \cdot let\_overlap \cdot 2 \cdot sizeof(pointer) + \#tiles \cdot 2 \cdot sizeof(pointer)$.

5.4 Experimental Results

In order to determine the computational and memory requirements of the algorithms, we have simulated several traces of 3D graphics applications and measured certain statistics such as the BBOX and LET overlap factors. Furthermore, we estimated most parameters such as t_{buf} and t_{insert} by counting the number of elementary operations (assignments, comparisons, etc.) required to implement the operation. Other parameters such as $t_{\text{bbox\_test}}$ can vary because the implementation of this operation contains if-then-else statements which implies that the time depends on the control flow. In order to estimate these parameters, we wrote a program that performs these tests and inserted counters to determine how often each branch was executed. These statistics have been subsequently substituted in the complexity formulae of the algorithms. We remark that cycle-accurate simulations of the algorithms on all workloads are not feasible, because that is too time consuming.

This section is organized as follows. Section 5.4.1 describes the experimental setup and presents the values of the statistics and the parameters used to calculate the time and memory required by each algorithm.

The efficiency of the overlap tests is discussed in Section 5.4.2. Finally, the runtime results and memory requirements of the algorithms are presented and discussed in Section 5.4.3.

5.4.1 Experimental Setup

In order to compare the efficiency of the proposed algorithms we used the GraalBench benchmark suite described in Chapter 3.

	Q3L	Q3H	Tux	Aw	ANL	GRA	DIN
$t_{bbox\_compute}$	14.02	14	13.98	14.25	14.25	14.25	14.20
t_{bbox_test}	2.08	1.995	1.780	1.86	1.93	1.77	1.78
t_{let_test}	50.5	43.40	48.93	59.15	52.9	54.66	57.79

Table 5.1: Complexity parameters for each workload

To determine the number of triangles and the overlap factors, the traces were fed to our modified Mesa library. The Mesa library performed primitive backface culling and generated lists of remaining primitives. The list of primitives was sent to our tile-based accelerator simulator, where different scene management algorithms were used. We used a tile size of 32x16 pixels (as used in current tile-based hardware accelerators [57]), and the window sizes were 320x240 for Q3L, and 640x480 for the other benchmark suite components.

Some of the parameters used to estimate the complexity of the algorithms ($t_{bbox\_compute}, t_{bbox\_test}, t_{let\_test}$) can vary across the workloads. In order to reduce the errors obtained by estimating them statistically, we wrote programs to compute the average number of elementary operations needed to implement each test and obtained particular values for each workload. The results are presented in Table 5.1. It can be seen that the obtained $t_{bbox\_compute}$ and t_{bbox_test} parameters are quite uniform across the workloads while t_{let_test} has a larger variation.

Other parameters of the workloads such as the average or maximum number of triangles per frame are presented in Table 5.2. The *average triangles / frame* statistics represents the average number of triangles sent from the Mesa library to our driver after backface culling. This number is actually the $\#triangles$ parameter used to compute the complexity of the algorithms. The *max. triangles* represent the maximum number of triangles sent for one frame. This number can be used to determine the maximum amount of memory required to buffer the triangles for one frame. This number can also be used to determine the computational power required for real-time operation.

For the other parameters, the following assumptions were employed: $t_{buf} = 50$, $t_{insert} = 6$ (two additions, three assignments, and one comparison), $t_{tiletrav} = 4$ (two comparisons, one assignment, and one increment), $t_{send} = 40$ (the number of I/O writes currently used to transfer the data for a triangle in our simulator).

	Q3L	Q3H	Tux	Aw	ANL	GRA	DIN
number of frames	1,379	1,379	1,363	603	600	599	600
average triangles / frame	3,350	3,436	1,825	11,053	4,455	3,681	4,150
max. triangles / frame	7,074	7,170	2,980	14,102	14,236	6,907	4,313
max. bbox / frame	19,288	53,154	11,789	18,822	37,771	11,233	9,858
max. let / frame	14,175	26,542	8,478	18,465	33,469	10,109	9,088
bbox_overlap	3.06	7.03	4.17	1.34	4.08	2.28	2.06
let_overlap	2.39	4.32	3.05	1.32	3.4	1.98	1.95
max. scene buffer memory required	594k	602k	250k	1,185k	1,196k	580k	362k

Table 5.2: Relevant characteristics of the benchmarks

	Q3L	Q3H	Tux	Aw	ANL	GRA	DIN
BBOX total I/O	701M	1,907M	664.5M	342M	524M	357M	215M
BBOX triangles I/O	432M	1,024M	320M	265M	338M	152M	160M
LET total I/O	583M	1,394M	565M	337M	459M	322M	206M
LET triangles I/O	340M	633M	235M	260M	281M	131M	150M

Table 5.3: I/O writes for various overlapping tests

5.4.2 Efficiency of the Overlap Tests

Table 5.3 presents the number of I/O write accesses to the accelerator based on our simulator using the BBOX test or LET test. The total I/O numbers represent the number of accesses due to state-changing information (e.g., enable or disable depth test) plus the number of I/O due to sending triangles. Since the number of primitives sent to the accelerator depends only on the overlapping test performed, these numbers are independent of the algorithm used. The number of I/O writes needed to sent triangles to the accelerator for the BBOX test was up to 62% larger than for the LET test. Another interesting result from this table is that the LET test not only reduces the number of I/Os, but also the number of state changes. This happens due to the fact that we used a *lazy* update mechanism for the state change (the state changes are committed to the renderer when they are needed, thus if some of the primitives are discarded some state changes might not be required). Thus, using a more accurate overlapping test is beneficial also for the amount of state change information sent to the accelerator.

5.4.3 Runtime Results and Memory Requirements

This section presents the results we obtained for the scene management algorithms. Once again, we were not able to perform a cycle accurate simulation of the workload due to the fact that it is too time consuming.

The average time taken by each scene management algorithm to process one frame of every benchmark is presented in Table 5.4, while Figure 5.3 depicts the time required by each algorithm relative to the amount of time taken by algorithm DIRECT. As expected, algorithm DIRECT requires the largest number of operations by far, while SORT takes the least amount of time. On average, across all benchmarks, the SORT algorithm is 44 times as fast as DIRECT. The TWO_STEP algorithm, even though it also scans the entire scene buffer for each tile, has reasonable performance. It is slower than algorithm SORT by a factor of 6 on average. It can also be observed that algorithm TWO_STEP_LET is hardly slower than TWO_STEP and, therefore, preferable, since it sends fewer triangles to the rasterizer which means that the computational load on the rasterizer is reduced. Algorithm SORT_LET, on the other hand, is slower than algorithm SORT by a factor of 1.6 on average.

The amount of memory required by each algorithm, in addition to the scene buffer which is needed to buffer the primitives, is presented in Table 5.5. It is also visually depicted in Figure 5.4. As explained before, algorithm DIRECT does not require any additional memory. Furthermore, as expected, the SORT algorithm needs the most memory, since the amount of additional memory it requires is proportional to the number of triangles and the BBOX overlap factor. Because the LET test is exact while the BBOX test is not, SORT_LET requires less memory than SORT. However, the difference is significant only for one benchmark (Q3H) for which SORT needs almost twice the amount of additional memory as SORT_LET. For the other benchmarks the difference is much smaller (a factor of 1.17 on average). The reason is that the BBOX test is fairly exact for all benchmarks except Q3H. The TWO_STEP and TWO_STEP_LET algorithms require the same amount of memory and are, therefore, depicted together. On average, TWO_STEP requires a factor of 3.2 less additional memory than SORT. However, this difference depends strongly on the benchmark. For benchmarks with a small overlap factor (e.g., Aw), the difference is hardly significant, while for benchmarks with a large overlap factor (in particular Q3H) the difference is considerable.

Which algorithm is preferable depends, of course, on the computational power and the amount of memory of a particular implementation. DIRECT is probably not a practical algorithm because it has poor performance. Our results

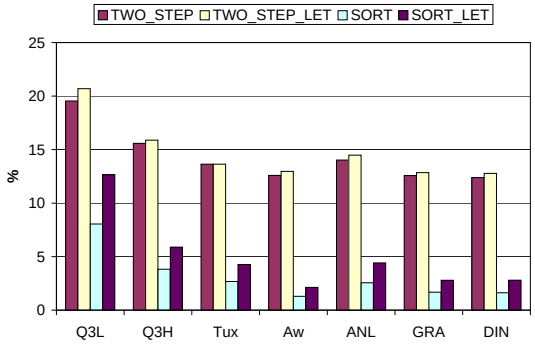


Figure 5.3: Time taken by each scene management algorithm, relative to the amount of time taken by algorithm DIRECT

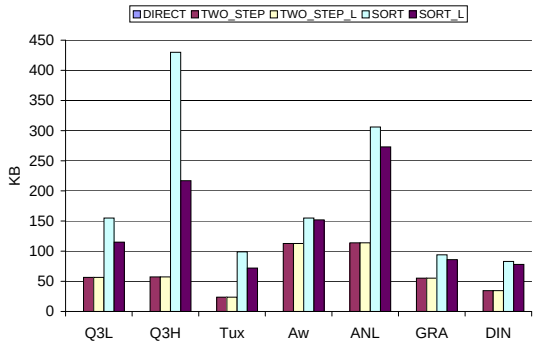


Figure 5.4: Memory requirements of the scene management algorithms

indicate that TWO_STEP_LET is better than TWO_STEP, since they require the same amount of memory and because TWO_STEP_LET takes only a bit more time than TWO_STEP. Furthermore, TWO_STEP_LET sends fewer triangles to the rasterizer, which implies that the rasterizer has to perform less work. SORT and SORT_LET take less time than TWO_STEP_LET, but require more memory. So the 3D graphics system designer has to make a trade-off between these algorithms. If SORT_LET is preferable to SORT or vice versa can also not be assured. Although SORT_LET sends fewer triangles to the rasterizer, SORT requires significantly less time than SORT_LET while SORT_LET reduces the memory requirements only marginally for all but one benchmark.

	Q3L	Q3H	Tux	Aw	ANL	GRA	DIN
DIRECT	8.7M	34M	17.6M	108M	44.2M	35.8M	40.7M
TWO_STEP	1.7M	5.3M	2.4M	13.6M	6.2M	4.5M	5.0M
TWO_STEP_LET	1.8M	5.4M	2.4M	14M	6.4M	4.6M	5.2M
SORT	0.7M	1.3M	0.47M	1.4M	1.13M	0.6M	0.66M
SORT_LET	1.1M	2.0M	0.75M	2.3M	1.9M	1.0M	1.1M

Table 5.4: Number of elementary operations per frame for each scene management algorithm.

	Q3L	Q3H	Tux	Aw	ANL	GRA	DIN
DIRECT	0	0	0	0	0	0	0
TWO_STEP/ TWO_STEP_LET	56.6k	57.4k	23.8k	112.8k	113.9k	55.26k	34.5k
SORT	155k	430k	99k	155k	306k	94k	83k
SORT_LET	115k	217k	72k	152k	273k	86k	78k

Table 5.5: Additional maximum memory requirements (bytes) per frame for each scene management algorithm.

5.5 Static and Dynamic Versions of the Bounding Box Test

In this section we show that the efficiency of the BBOX test can be improved significantly by adaptively varying the order in which the comparisons are performed depending on the position of the current tile.

As mentioned before, the scene management algorithms can be implemented in the accelerator or by the host processor. Because the amount of chip area of our low-cost target system is severely limited, we have assumed it needs to be performed by the host CPU. If sorting is performed in software then the order in which the comparisons of the BBOX test are applied can have a significant impact on the performance. For example, for the tile in the upper-left corner we expect that most triangles are located to the east and/or south of it.

Computing the BBOX of a primitive is in general more expensive than testing if the BBOX and the tile intersect. However, in some algorithms for sorting the primitives the BBOX calculation is performed only once per primitive while the second step of the BBOX test that checks if the BBOX intersects with the current tile is performed for every combination of primitive and tiles.

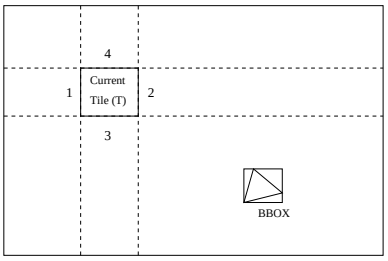


Figure 5.5: Triangle to tile BBOX test using 1D prediction with correlation

The second step of the BBOX test, therefore, has a larger impact on the time consumption of the sorting scheme than the first step.

The algorithm that we consider for BBOX test optimizations in this section is the TWO_STEP algorithm. The reason for choosing the TWO_STEP algorithm is that it has reasonable performance and does not require a significant amount of memory in addition to the scene buffer.

5.5.1 Static Bounding Box

The four comparisons required to determine if a BBOX and a tile overlap (see Section 5.2.1) can be performed in an arbitrary order. This gives a total of 24 possible arrangements. However, not every order produces the same number of comparisons on average. In this section we discuss two versions of the static bounding box test that might generate a different number of comparisons on average.

A tile divides the screen into five, possibly intersecting regions: the tile itself, the region to the east of the tile ($x >= T.MaxX$), the region to the west of the tile ($x < T.MinX$), the region to the north ($y >= T.MaxY$), and the region to the south ($y < T.MinY$). If a certain test (comparison) fails, then there is a high probability that the test in the opposite direction along the same dimension succeeds. This is because after these two tests there is only a small region left where the BBOX of a primitive can be situated.

Fig. 5.5 illustrates the case in which first the region to the west of the tile is checked, then the opposite region along the same dimension (east), then the region to the south, and, finally, the region to the north. After performing only two comparisons (the horizontal intersection tests), all primitives that are completely located to the east or west of the tile are rejected. This usually leaves

only a small number of triangles that require more than two comparisons. Of course, different orders are also possible (for example, north, south, west, east). However, assuming that the primitives are equally distributed over the scene, they should produce the same number of comparisons on average. We refer to this scheme as *STATIC1*.

To determine if *STATIC1* reduces the average number of comparisons, we will compare it to a scheme in which the first and second (and, hence, the third and fourth) comparison check different dimensions. For example, one possible order is west, south, east, north. Statistically, there should be no difference between the possible orders. We refer to this static variant as *STATIC2*.

5.5.2 Dynamic Bounding Box

As stated before, a tile divides the scene into four regions (five if we include the tile itself). The probability that a primitive is completely located in the largest region is the highest. This observation is the basis of our "dynamic" versions of the bounding box test.

We describe two dynamic schemes. In the first, referred to as *DYNAMIC1*, we first check the largest region. Thereafter, the opposite direction along the same dimension is tested. The third test examines the largest region in the other dimension, and the fourth test checks the remaining region. The second dynamic version of the bounding box test is referred to as *DYNAMIC2*. In this scheme, the comparison corresponding to the largest region is applied first, then the comparison corresponding to the second largest region, etc. The region to the east of the tile is the largest and checked first, then the region to the south, then the one to the north, and, finally, the region to the west.

We remark that although these schemes are called dynamic, the order in which the comparisons are applied depends only on the tile position and can be determined statically off-line. For example, for all tiles in the upper left sub-scene under the main diagonal, the order is east, south, north, west.

5.5.3 Experimental Results

In order to compare the efficiency of the proposed algorithms we used the GraalBench benchmark suite described in Chapter 3. We used a tile size of 32×16 pixels, and the window sizes were 320×240 for Q3L, and 640×480 for the other benchmark suite components.

Fig. 5.6 depicts, for each workload, the average number of comparisons per

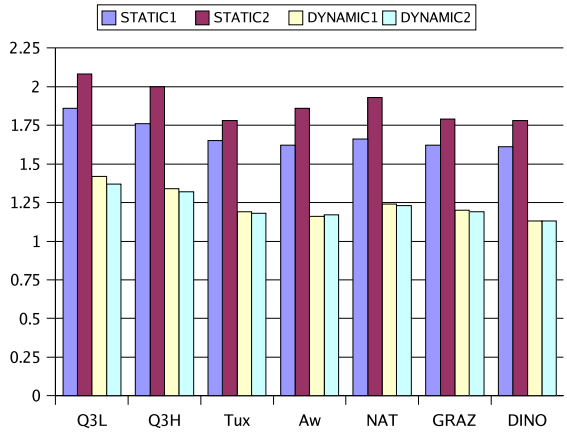


Figure 5.6: The average number of comparisons per primitive for each workload

primitive required by each version of the bounding box test. As expected, the *STATIC1* scheme indeed performs better than the *STATIC2* scheme. On average, across the benchmarks, *STATIC1* requires 11% fewer comparisons than *STATIC2*. It can also be seen that there is little difference between the two dynamic versions. This can be explained as follows. If the largest region has been checked, then a large part of the second largest region has also been checked. Suppose, for example, that the largest region is to the east of the tile and that the second largest region is to the south. If the region to the east has been checked, then the region to the southeast of the tile has also been checked, while the region to the west has not. Nevertheless, we observe that *DYNAMIC2* performs slightly better than *DYNAMIC1*. Furthermore, both dynamic schemes require fewer comparisons than the best static version. On average, the best dynamic version *DYNAMIC2* requires 26% fewer comparisons than the best static scheme *STATIC1*.

These results show that by dynamically varying the order in which the comparisons are performed depending on the position of the current tile indeed reduces the average number of comparisons needed to determine that a triangle does not intersect the tile.

5.6 Conclusions

A two stage model for triangle to tile repartition for tile-based graphics accelerators has been presented. In addition, different overlap tests have been described. By comparing factors like computational power or memory required, a 3D graphics accelerator designer can chose the most suitable algorithm for an implementation. While the number of primitives sent to the accelerator depends only on the overlap test used, the memory required to sort and store the primitives before being sent to the accelerator and also the computational power depends largely on the algorithm employed.

The DIRECT algorithm is probably not a practical algorithm because it has poor performance. However, the TWO_STEP algorithm, even though it also scans the entire scene buffer for each tile, has reasonable performance while it does not require a large amount of additional memory. If the computational performance is more important than additional memory required, then the SORT algorithm is more suitable. To determine the best algorithm for a particular implementation it should be taken into consideration also the fact that using the LET overlap test, instead of BBOX, more computational power is required at the scene management stage. However, depending on the computational power required to process and discard additional triangles on the accelerator, it might still be a better choice to use LET instead of BBOX since the computations at the accelerator to discard the additional triangles generated by BBOX can be higher.

In this chapter we have also described several possible software implementations of the bounding box test. The static versions always perform the comparisons involved in the same order, while the dynamic versions base the order on the position of the current tile. The experimental results show that the dynamic scheme in which the comparison corresponding to the largest region is applied first, then the comparison corresponding to the second largest region, etc., requires the least comparisons on average (26% fewer comparisons than the best performing static version).

Chapter 6

Efficient State Management for Tile-Based 3D Graphics Architectures

In Chapter 4 we have shown that tile-based rendering considerably reduces the memory traffic between the rasterizer and the off-chip frame and z buffers. However, it increases the amount of state change information such as enable/disable z testing and create/delete texture commands that need to be sent to the rasterizer. This is because determining which state change operations can be safely removed and when is not trivial. For instance, if a delete texture command is encountered while rendering the current tile, the texture can be safely deleted only when all primitives (from all tiles) that use this texture are rendered or it can be deleted when multiple copies of the texture are kept in memory. Also, including all the state change operations to each tile is not a practical solution since it requires duplicating large amounts of state variables (e.g., texture objects) for each tile. In some cases, the state change operations account for 63% of the data sent to a tile-based renderer.

In this chapter we determine the optimal state change operations (e.g., enable/disable z testing, create/delete a texture) that should be included for each tile. Experimental results obtained using several suitable 3D graphics workloads show that a significant amount of state change traffic can be saved.

This chapter is organized as follows. Section 6.1 describes the OpenGL state information. In Section 6.2 algorithms suitable for tile-based state management are described. Experimental results are presented in Section 6.3, and the conclusions are given in Section 6.4.

```
EnableDepth
Triangle(1)
DisableDepth
Triangle(2)
EnableDepth
Triangle(3)
```

Figure 6.1: Static state, initial instruction stream fragment.

6.1 OpenGL State Information

The OpenGL state information can be divided into two parts. The first part is the static state information, that is the state information that needs to be stored irrelevant of the application. For instance, the information that describes the state of the depth unit is always defined. The second part of the state information is the dynamic state information. The dynamic state information contains the state information which is application dependent. For instance, the texture state depends on the number of textures loaded by the application.

6.1.1 Static State Information

The static state information part of the OpenGL state machine is usually less than the dynamic state information and it has no side effects. More precisely, duplicating the static information to each tile does not affect the execution semantics of OpenGL. However, since the primitives that do not overlap a tile are deleted from the instruction list of the respective tile, some state information might be also eliminated. Figure 6.1 depicts a fragment of instructions sent to the rasterizer. Assuming that triangle 1 overlaps tile 2, triangle 2 overlaps tile 1, and triangle 3 overlaps tiles 1 and 2, the instruction stream that might be generated by a tile-based driver is depicted in Figure 6.2. The emphasized state changing instructions can be also deleted from the tiled instruction stream. By eliminating the unnecessary state change instructions, the data traffic to the rasterizer is decreased. Nevertheless, we note that determining if a state instruction can be eliminated might consume actually more system bandwidth than sending it directly to the rasterizer.

```
Tile 1
   EnableDepth
   DisableDepth
   Triangle(2)
   EnableDepth
   Triangle(3)

Tile 2
   EnableDepth
   Triangle(1)
   DisableDepth
   EnableDepth
   Triangle(3)
```

Figure 6.2: Static state, tiled instruction stream fragment.

6.1.2 Texture State Information

In this section we describe in more detail the state information required to be stored for texturing. Figure 6.3 depicts the texture state information organization. Each texture unit supported by the hardware has a link to a current texture object. Each texture object, identified by a texture id, contains information such as the texture image format, the width and the height of the largest texture level, the minification and magnification functions, and links to texture images for valid texture levels. A texture object can be bound to a texture unit using the BindTexture OpenGL command. In the instruction streams pseudo-code from the following sections, the BindTexture command can be also encountered under the MakeCurrentTexture name. When a texture object is no longer needed, it can be deleted using the DeleteTexture command.

6.2 State Management Algorithms for Tile-Based Rendering

This section presents two algorithms that can be used to correctly handle the state information when using a tile-based rendering model.

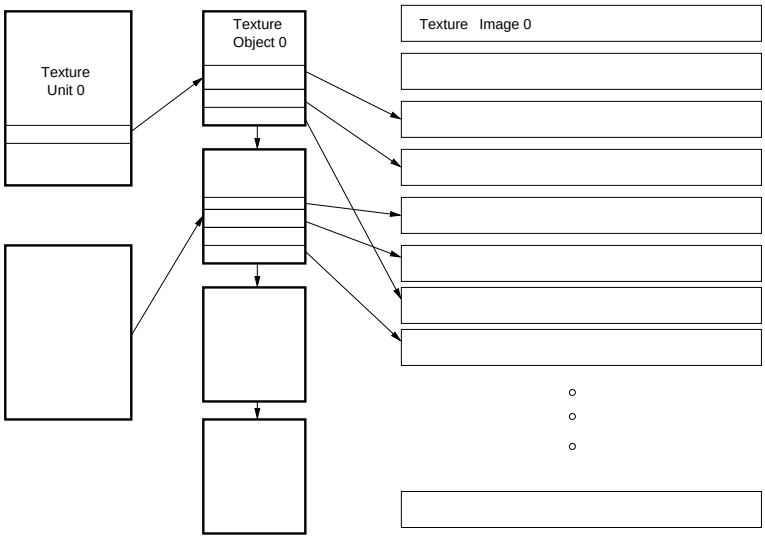


Figure 6.3: Texture state information.

```
Start Frame
   CreateTexture(i)
   MakeCurrentTexture(i)
   Triangle(1)
   Triangle(2)
   DeleteTexture(i)
   CreateTexture(i)
   MakeCurrentTexture(i)
   Triangle(3)
End Frame
```

Figure 6.4: Instruction stream.

6.2.1 Partial Rendering Algorithm

In this algorithm, whenever an instruction that has side-effects (e.g., DeleteTexture) is encountered in the input stream, the driver renders all previously buffered instructions and then executes the instruction. While partial rendering is a solution to rendering commands with side effects, it might also introduce significant rendering overhead. For example, consider that the instruction stream depicted in Figure 6.4 was sent to the rasterizer. The assumptions are

```
Start Frame
Tile 1
   c1=SaveCurrentContext
   RestoreTileFromGlobalBuffer
   CreateTexture(i)
   MakeCurrentTexture(i)
   Triangle(2)
   SaveTileToGlobalBuffer
   c2=SaveCurrentContext
Tile 2
   RestoreContext(c1)
   RestoreTileFromGlobalBuffer
   MakeCurrentTexture(i)
   Triangle(1)
   SaveTileToGlobalBuffer
   DeleteTexture(i)

Continue rendering the remaining primitives for
tiles 1 and 2 after deleting texture i.

Tile 1
   c1=SaveCurrentContext
   RestoreTileFromGlobalBuffer
   CreateTexture(i)
   MakeCurrentTexture(i)
   Triangle(3)
   SaveTileToGlobalBuffer
Tile 2
   RestoreContext(c1)
   RestoreTileFromGlobalBuffer
   MakeCurrentTexture(i)
   Triangle(3)
   SaveTileToGlobalBuffer
End Frame
```

Figure 6.5: Tiled instruction stream using partial rendering.

```
Start Frame
Tile 1
   c1=SaveCurrentContext
   RestoreTileContentsFromGlobalBuffer
   CreateTexture(i)
   MakeCurrentTexture(i)
   Triangle(2)
   MarkDeleteTexture(i)
   RenameTexture(i,j)
   MakeCurrentTexture(j)
   Triangle(3)
   SaveTileContentsToGlobalBuffer
Tile 2
   RestoreContext(c1)
   RestoreTileContentsFromGlobalBuffer
   MakeCurrentTexture(i)
   Triangle(1)
   MakeCurrentTexture(j)
   Triangle(3)
   SaveTileContentsToGlobalBuffer

After Last Tile
   DeleteTexture(i)
   MoveTextureLinks(i,j)
End Frame
```

Figure 6.6: Tiled instruction stream using delayed commit.

the same as described in Section 6.1.1. The tile-based driver or a tile-based rasterizer state management engine could issue the instruction stream depicted in Figure 6.5. Since tiles are rendered sequentially, all the instructions preceding the DeleteTexture(*i*) instruction, in all tiles, must be rendered so that all the primitives using texture *i* are rendered and thus the DeleteTexture instruction can be executed. For each partial rendering the introduced overhead consists of saving (SaveTileToGlobalBuffer) and reloading (RestoreTileFromGlobalBuffer) the contents of the enabled tile buffers (e.g., color, depth, and stencil) from the global buffers and also the state information save (SaveCurrentContext) and reload (RestoreContext) operations.

6.2.2 Delayed Execution Algorithm

In this algorithm, when commands that affect dynamic state, e.g., Delete Texture or TextureImage, are encountered in the input stream, the driver will postpone their execution until all the primitives depending on them are rendered or the end of the current frame is reached. For instance, assume that a DeleteTexture was encountered. As long as no request to create new textures are received, thus no reuse required for the texture ids, the execution of the DeleteTexture command can be safely delayed until all the primitives that use the texture are executed. However, if the application requests a new a texture id from the OpenGL front-end and obtains a texture id that was deleted on the same frame but not committed, then the tile-based driver must create a new texture object that will be linked with a new id, while the old texture object will remain accessible until all primitives using the old texture are rendered or until the end of the frame. In Figure 6.6, texture *i* is only marked to be deleted, and the new texture that should replace it is named *j* instead of *i*, using the MarkDeleteTexture(*i*) and RenameTexture(*i*,*j*) commands. Each following reference in the original stream for texture *i* is replaced with a reference to texture *j* instead. At the end of the frame, texture *i* can be deleted and texture *j* can be renamed as *i* and all the requests for texture *i* will no longer require an id change. Executing (committing) the commands when all primitives depending upon them are finished has a higher computational power than executing it at the end of the frame since it requires determining the last tile and the last primitive depending on it.

6.3 Experimental Results

As benchmarks we used the GraalBench benchmark suite described in Chapter 3. We used a tile size of 32×16 pixels, and the window sizes were 320×240 for Q3L, and 640×480 for the other benchmark suite components.

Figure 6.7 depicts the percentage of the state information and primitives sent to the accelerator. The average percentage of unoptimized tile-based state information across the benchmarks is 44%. This high percentage is obtained due to the overhead of state information replication and also additional tile-based specific state change information such as load and save tile operations. The Aw component has the lowest percentage of state change due to the fact that most of the state change is performed in the first frames and there is little state variation from frame to frame. GRAZ, Tux, and Q3, on the other hand, combine multiple texture and blending modes and depth tests so they require more

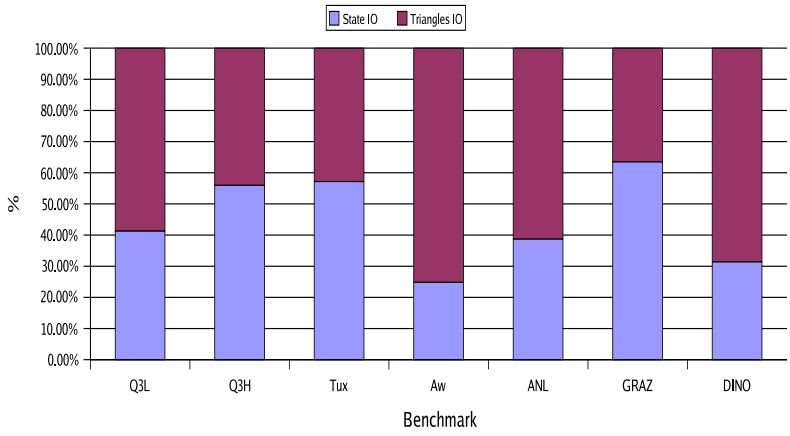


Figure 6.7: Percentage of state information and triangles sent to the accelerator per frame.

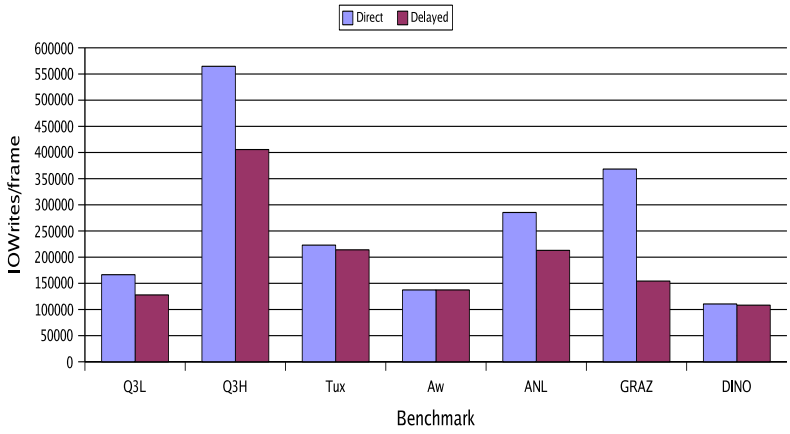


Figure 6.8: Average number of state information writes to the accelerator per frame.

state information to be sent to the accelerator.

Figure 6.8 presents the state changes sent to the rasterizer using direct transfer and lazy commit (delayed) methods. The delayed method reduces the number of writes to the accelerator by up to 58% by filtering the state information and eliminating unnecessary writes, thus providing the optimal state traffic at the expense of a small additional buffer. The obtained results show that the state information for the Q3H, GRAZ, and ANL components can be reduced significantly. The state traffic for the Aw, Tux, and DINO components, however, could not be decreased substantially since there were hardly any unnecessary state changes.

6.4 Conclusions

In this chapter we have presented several state management algorithms for tile-based renderers. While in traditional (non tile-based) rendering the state information traffic can be negligible compared to the traffic generated by the primitives, in tile-based rendering architectures, since the state information might need being duplicated in multiple streams, the required processing power and generated traffic can increase significantly. Moreover, removing primitives from the instruction stream of a tile depends only on the primitive position and the tile coordinate. To remove a state change instruction from the instruction stream of a tile, information about the previous or the following state change instructions and/or primitives is required. Thus, in order to send an optimal state change stream to the accelerator, i.e., use minimal bandwidth, additional processing power and more processor bandwidth is required. By sending an optimal state change stream to the accelerator, the state change traffic to the accelerator was decreased by up to 58% compared to the state change traffic generated by directly sending the intial state.

Chapter 7

A Low-Cost, Power-Efficient Texture Cache Architecture

In Chapter 4 we have shown that besides the color traffic (the traffic between the accelerator and the external frame buffer) and the z traffic, texture traffic also requires significant memory bandwidth. For some workloads the texture traffic generated by a tile-based renderer can be as high as 80% of the total external traffic. Consequently, techniques that reduce the amount of texture traffic will decrease the total external data traffic significantly.

A common technique to reduce the amount of memory bandwidth required for texture mapping is to employ a large texture cache. In our targeted system, however, a large cache is unaffordable given the gate count limitations described in Section 1.1.3. Furthermore, large caches consume more power than small caches. In this chapter we, therefore, propose to employ a very small (128-512 byte) texture cache and investigate the bandwidth saved by employing such a small cache and its energy consumption.

The reason we believe that such a small cache is suitable for texture mapping is the following. Texture mapping, as described in more detail in Section 2.4.3, consists of mapping an image (texture) to an object. Any object can be decomposed in triangles after which each triangle can be rendered separately, having its own coordinates in the texture space. One method to render a triangle is to decompose it in horizontal lines (scanlines) and then to walk along each scanline. In order to find the color component of each point of the scanline, a projection in the texture space is performed and a value corresponding to an interpolation among neighbors is computed. The case of bilinear interpolation is illustrated in Figure 7.1. As can be seen, point P_1 corresponds to point U_1

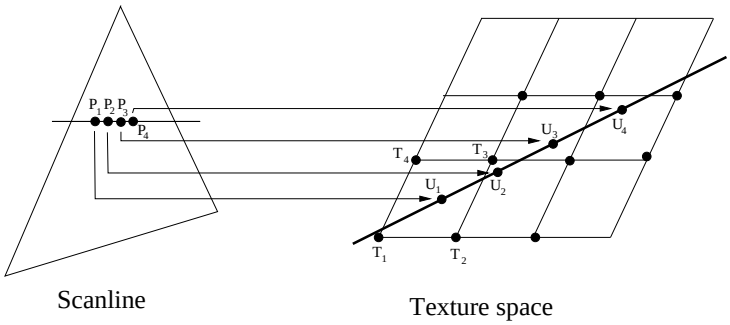


Figure 7.1: Bilinear Interpolation.

in texture space. The value of U_1 is computed by bilinear interpolation among the texels T_1, T_2, T_3, and T_4. When computing the corresponding value for the next point of the scanline, P_2, the texels T_2 and T_3 are reused. In general, for each next point of the scanline two previous texels from the texture space are likely to be needed again. Furthermore, in the case of the more often used trilinear interpolation, it is likely that four of the previous eight texels are needed again.

Due to this high spatial locality, we assume that even a small texture cache can improve the performance and reduce the power consumption of texture mapping substantially.

In this chapter we focus on the dynamic power consumption due to switching activity. Since recently, however, the static power consumption due to leakage current is becoming more of a concern [15]. The reasons we focus on dynamic power consumption are the following. First, at the time this research was performed the static power consumption was negligible. Second, employing a small cache not only reduces the dynamic power consumption but also the static power consumption. In fact, techniques to reduce the static power consumption are based on switching off parts of the cache during periods of inactivity [45]. Third, in the technology we are targeting, the dynamic power consumption is still the dominating factor of the cache power consumption.

This chapter is organized as follows. In Section 7.1 related work is described. Section 7.2 describes the model used to estimate cache power consumption. Section 7.3 presents the experimental methods and workloads, and presents the experimental results. Conclusions and directions for future research are given in Section 7.4.

7.1 Related Work

Related work can be divided into two categories: techniques to reduce texture traffic and low-power cache organizations in general. Furthermore, there are two main techniques to reduce texture traffic: texture caching and texture compression.

Hakura and Gupta [36] found that the factors important to texture cache behavior are the representation of texture images in memory, the rasterization order, and the cache organization. They used a blocked representation, in which texels that are within a square region (block) of a two-dimensional image are stored consecutively in memory. The downside of this representation is, however, that it involves more addressing overhead than the conventional representation. They also proposed to reduce the working set size by the use of a tiled rasterization order. Finally, they investigated the effects of different cache parameters such as line size and associativity. However, they focused on high performance and proposed rather large caches (16 KBytes with 2-way associativity).

Igehy et al. [41] proposed a prefetching texture cache architecture. It is questionable, however, if prefetching can be used to reduce the energy consumption because the structures needed such as a reorder buffer also consume power.

Kim and Kim [46] proposed to adaptively select the cache index depending on whether the texture space is traversed more along the horizontal direction or along the vertical direction. Because data can be placed in two cache lines, their method, however, requires to read two cache lines on each access, which increases the power consumption.

Besides texture caching, texture compression can also be used to reduce the amount of texture traffic. Fenney [28] presented a lossy texture compression technique especially suitable for low-bandwidth devices. It uses a representation that consists of blending two, or more, low frequency signals using a high frequency but low precision modulation signal. By using a fixed-rate encoding, this technique also provides low decompression costs. The achieved quality of this lossy technique however varies based on the uncompressed source texture complexity.

Because the power consumption of on-chip caches constitutes a significant part of the total power consumption of modern microprocessors, low-power cache organizations for embedded applications in general have been proposed as well. Most related to our work is the work of Kin et al. [47], who proposed to place a small, level-0 (L0) cache between the processor and the L1 cache.

This cache has been coined filter cache because it filters references to the L1 cache. In general, however, the hit rate of the filter cache is rather low.

7.2 Cache Power Consumption

In this section we describe the model used to estimate cache power consumption, and discuss and motivate the metrics chosen to measure energy efficiency. We did not use the model described by Kin et al. [47] because it can be rather inaccurate. For example, they assume that half the address lines switch during each memory request. However, this assumption is not valid for caches (and, moreover, biases small block sizes) due to temporal and spatial locality of references. In our estimations we, therefore, used precise counts obtained by simulation.

7.2.1 Cache Power Model

The cache power model is based on the cache timing and power consumption model used in the CACTI 2.0 tool [59], which in turn is based on the cache model proposed by Wilton and Jouppi [75]. The source of power dissipation in this model is the charging and discharging of capacitative loads caused by signal transitions. The energy dissipated for a voltage transition $0 \rightarrow V$ or $V \rightarrow 0$ is approximated by:

$$E = \frac{1}{2} C V^2, \tag{7.1}$$

where C is the capacitance driven. An analytical model of the cache power consumption includes the equivalent capacitance of cache components considered and the voltage swing of a transition. The power consumption is estimated by combining Eq.(7.1) and the transition count at the inputs and outputs of each modeled component, usually obtained by simulation.

To reduce wordline and bitline capacitance, and achieve minimal access time, the cache SRAM array is subdivided into subarrays. The wordlines are subdivided in N_{dwl} partitions, each containing the total number of cache rows and a full row decoder. The bitlines are subdivided in N_{dbl} parts, each containing $\frac{1}{N_{dbl}}$ of the total cache rows. Another parameter, N_{spd}, allows to map more sets in a single wordline and thus change the overall access time without breaking the array into smaller subarrays. The power model used takes into account the

value of these organizational parameters. The optimal values for the array organization parameters $N_{dwl}, N_{dbl}, N_{spd}$ depend on the cache size and the block length, and are computed using CACTI.

The other cache parameters used in the formulae presented below are summarized in Table 7.1. The number of columns N_{cols} and rows N_{rows} of the SRAM subarray can be derived from the cache parameters.

parameter	**description**
A	Associativity.
C	Cache size (in bytes).
B	Block size (in bytes).
D_{bits}	Data Word size.
A_{bits}	Address size.
V_{dd}	Supply voltage.
V_{pre}	Bitline precharge voltage.
T_{acc}	Access time.
N_{acc}	Number of cache accesses.
N_{rd}	Number of cache read accesses.
N_{wr}	Number of cache write accesses (write hits).
N_{rmiss}	Number of cache misses on a read.
N_{wmiss}	Number of cache misses on a write.
$N_{atr,m}$	Total address bit transitions on the outgoing lines.
$N_{atr,c}$	Total address bit transitions on the cache decoders.
W	Average number of bits written per write operation.

Table 7.1: Cache parameters and transition counts.

To obtain good estimates of the power dissipated, accurate transition counts are essential. Transition counts can be determined exactly by simulation or, when this is not possible, can be approximated by multiplying the expected transition probability at a node by the cycle count [14]. In our estimations we use precise counts for cache accesses and address bit transitions to and from memory. The average width of a data item written to memory (W) is estimated assuming an equal distribution of bytes, half-words and 32-bit words, as in [47]. We also assume that the transition counts of address and data bits are evenly distributed between accesses that hit and miss the cache.

The following cache components are fully modeled: address decoder, wordline, bitline, sense amplifiers (see Figure 7.2), and data output drivers (Figure 7.3). In addition, the address lines going off-chip and the data lines (both going off-chip and going to the CPU, are taken into account. The following sections describe the energy contributions listed above to the overall dissipa-

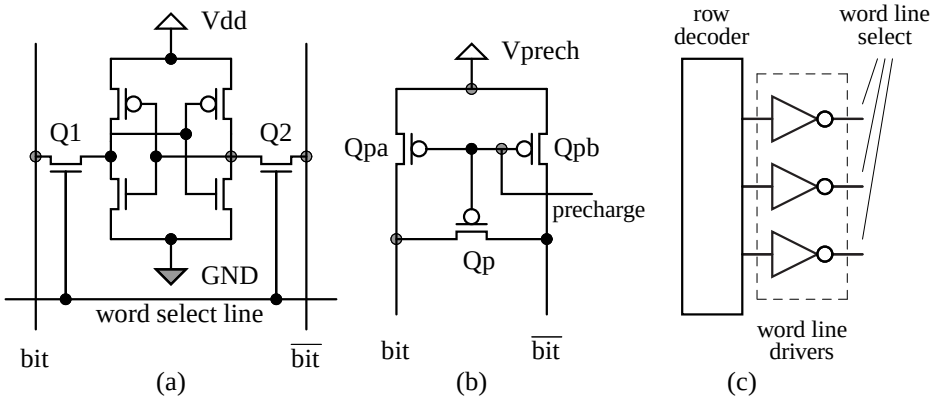


Figure 7.2: Components of a static RAM considered in the cache power model: (a) 6-transistor bit cell; (b) bit line precharging logic; (c) word line drive logic.

tion.

7.2.2 Bitlines

A major fraction of the power consumption is needed to drive the bitlines of the data SRAM arrays. The cache bitlines are driven during precharging (Eq.(7.2)), read (Eq.(7.3)) and write (Eq.(7.4)).

$$E_{bit,pre} = \frac{1}{2} V_{dd} V_{pre} \cdot N_{acc} \cdot N_{cols} \cdot C_{bitline}, \tag{7.2}$$

$$E_{bit,rd} = \frac{1}{2} V_{dd}^2 \cdot N_{rd} \cdot D_{bits} A \cdot C_{bitline,out} + \frac{1}{2} V_{dd} V_{pre} \cdot N_{rd} \cdot N_{cols} \cdot C_{bitline}, \tag{7.3}$$

$$E_{bit,wr} = \frac{1}{2} V_{dd} V_{pre} \cdot (N_{wr} \cdot W + N_{rmiss} \cdot 8B) \cdot C_{bitline}. \tag{7.4}$$

The capacitance of the bitline, $C_{bitline}$, is given by:

$$C_{bitline} = N_{rows} \cdot \left(\frac{1}{2} C_{d,Q1} + C_{bwire} \right) + 2C_{d,Qpa} + C_{d,Qp} + C_{d,mux}$$

The value of drain and gate capacitances depend on the size of the transistor and are computed using the equations presented by Wilton and Jouppi.

The factor $\frac{1}{2}$ accounts for the fact that the drain of the pass transistor of a bit cell is shared with the adjacent cell, therefore the capacitance $C_{d,Q1}$ is reduced by half. C_{bwire} is the capacitance of a bitline segment one bit cell high. $C_{d,Qpa}, C_{d,Qp}$ are the drain capacitances of precharge and equilibration transistors, respectively. The drain capacitance $C_{d,mux}$ is due to the bitline multiplexor at the input of the bit line sense amplifiers, which is present only if $8B \cdot N_{dbl} N_{spd} > D_{bits}$.

The capacitance $C_{bitline,out}$ is due to the input transistors of the sense amplifiers at the end of the bitlines and, if present, the drains of the bitline multiplexors.

$$C_{bitline,out} = 2C_{g,senseamp} + \left(\frac{8B \cdot N_{dbl} N_{spd}}{D_{bits}} \right) \cdot C_{d,mux}$$

Notice that the caches modeled are write-through, therefore a write operation causes a cache update only if the word is already in the cache (write hit).

7.2.3 Wordline

The energy dissipated by the wordlines is given by:

$$\begin{aligned} E_{word} = & V_{dd}^2 \cdot N_{acc} \cdot (N_{cols} \cdot (2C_{g,Q1} + C_{wwire}) \\ & + C_{d,wdrv} + C_{g,wdrv} + C_{d,winv}). \end{aligned} \tag{7.5}$$

The factor $\frac{1}{2} V_{dd}^2$ is multiplied by 2 because the word select signal is assumed to be pulsed, thus each read or write operation causes two transitions on the selected word line. The capacitance $C_{d,wdrv}$ is due to the drain of the P- and N-channel transistors of the wordline driver. The overall load of the wordline inverter stage at the input of the wordline driver is modeled as the combined capacitance of the driver gates $C_{g,wdrv}$ and the inverter drains, $C_{d,winv}$.

7.2.4 Sense Amplifiers

Each sense amplifier detects the voltage variation of a bitline resulting from a read operation and amplifies such variation to the full voltage swing V_{dd}. According to the cache model used, the energy dissipated in the sense amplifiers is a large fraction of the overall dissipation, and cannot be ignored:

$$E_{senseamp} = P_{sense} \cdot T_{acc} \cdot N_{acc} \cdot AD_{bits}. \tag{7.6}$$

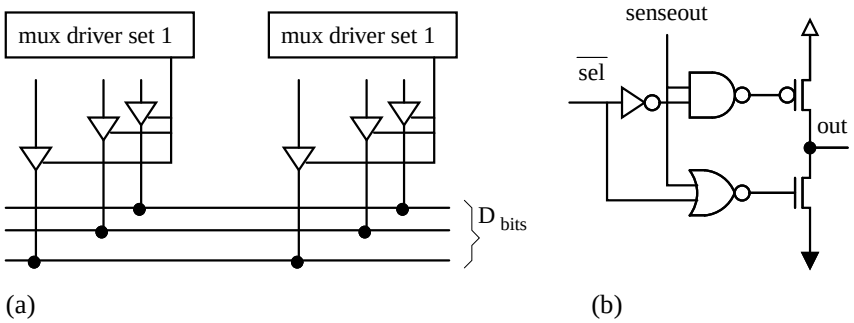


Figure 7.3: Output data bus: (a) overview; (b) output driver (tri-state).

P_{sense} is the average power consumption of a sense amplifier when a bitline transition occurs. Notice that the number of sense amplifiers is reduced by multiplexing the bitlines, hence the factor AD_{bits}.

7.2.5 Data and Address Output

The data output driver drives the cache output data lines with the signals from the sense amplifiers. In a set-associative cache there are AD_{bits} sense amplifier outputs and only D_{bits} outgoing data lines, therefore each output driver is actually a tri-state that charges or discharges the output line only if the corresponding set is selected (see Figure 7.3).

The energy dissipated in the data and address output lines is given by:

$$\begin{aligned} E_{dout} =& \frac{1}{2} V_{dd}^2 \cdot N_{rd} \cdot D_{bits} \cdot (C_{d,outnor} + C_{g,outdrv} \\ & + A \cdot C_{d,outdrv} + C_{wwire} \cdot 8BA \cdot N_{spd} \cdot vstack \\ & + C_{doutc}) + \frac{1}{2} V_{dd}^2 \cdot \frac{1}{2} W \cdot N_{wr} \cdot C_{doutm}, \end{aligned} \tag{7.7}$$

$$E_{aout} = \frac{1}{2} V_{dd}^2 \cdot N_{atr,m} \cdot C_{aoutm}. \tag{7.8}$$

where $C_{d,outnor}$ is the total capacitance of the drains of the NOR gate P- and N-channel transistors, $C_{g,outdrv}$ is the gate capacitance of the output driver and $C_{d,outdrv}$ is the capacitance of the output line due to the drivers attached to it. The remaining output line capacitance is due to the wire (whose estimated length is $8BA \cdot N_{spd} \cdot vstack$, where $vstack$ is the number of subarrays arranged horizontally) and C_{doutc}, the overall capacitance of the path between

cache output line and the CPU. Similarly, C_{doutm}, C_{aoutm} are, respectively, the capacitance of the data and address paths going off-chip.

7.2.6 Address Decoder

The cache model used in CACTI assumes that each subarray has its own decoder, which is made of three stages. The first stage contains predecoder blocks. These take 3 address bits as input and produce a 1-of-8 output signal by means of 8 3-input NAND ports. The 1-of-8 codes from different blocks are combined using N_{3to8}-port NOR gates, where

$$N_{3to8} = \left\lceil \frac{1}{3}\log_2(N_{rows}) \right\rceil .$$

Each NOR gate controls the selection signal of one wordline.

The energy dissipated in the decoder is subdivided in three components corresponding to the three stages:

$$\begin{aligned} E_{adec,1} =& \frac{1}{2}V_{dd}^2 \cdot 2N_{atr,c} \cdot (C_{d,ddrv} + N_{dbl}N_{dwl} \cdot 4C_{g,3to8} \\ & + 2BA \cdot N_{dbl}N_{dwl} \cdot C_{wwire}), \end{aligned} \tag{7.9}$$

$$\begin{aligned} E_{adec,2} =& \frac{1}{2}V_{dd}^2 \cdot N_{acc} \cdot \left(C_{d,3to8} + C_{g,decnor} \cdot \frac{N_{rows}}{8} \right. \\ & \left. + C_{bwire} \cdot N_{rows} \right), \end{aligned} \tag{7.10}$$

$$E_{adec,3} = \frac{1}{2}V_{dd}^2 \cdot N_{acc} \cdot (C_{d,decnor} + C_{g,winv}). \tag{7.11}$$

The energy dissipated in the first and third stage is based on the precise transition counts obtained via simulation: $N_{atr,c}$ and N_{acc}, respectively. The capacitance load of the drivers at the input of the first stage, $C_{d,drv}$, due to the drains of the transistors, is shared across all the subarray decoders. Each decoder contributes to the overall capacitance of the first stage with the wire that connects the driver output with the NAND gates of the 3-to-8 predecoder blocks and with the transistor gates of the blocks. The factor 2 of $N_{atr,c}$ accounts for the fact that for each address bit there also the inverted bit must be driven. For each address bit undergoing a transition, all 8 gates of the 3-to-8 block undergo a transition (4 are charged and 4 are discharged). The factor 2 before the term due to the wire capacitance accounts for the fact that the SRAM arrays are assumed to be organized so that the predecoders are at the center of the array

and the wires that connect them to the input driver are a quarter of the total array length. The last stage spends power to drive the capacitance of the NOR drains of the selected word line and the gates of the inverter at the input of the wordline driver. Note that the power consumed by the wordline inverter is accounted for in Eq.(7.5).

The power consumed in the tag path to select and retrieve the cache block tag is also taken into account. The expressions for the tag path are similar to those for the data path. In this case, the number of columns is the number of tag bits. The tag array is organized in subarrays independent from the data subarrays, i.e. the tag path is characterized by three organizational parameters. The model presented above does not include the power dissipated by comparators that verify a tag / address match, nor data steering logic and cache control logic. These components give a minimal ($< 2\%$) contribution to the overall power dissipation.

7.2.7 Energy-Related Metrics

In order to evaluate the efficiency of the texture cache architecture, we also measure the energy-delay (E-D) product. This metric was proposed by Gonzales and Horowitz [33], who argue it is superior to the commonly used power or energy metrics because it combines energy dissipation and performance.

To compute the delay D we assumed a clock frequency compatible with the access times estimated by CACTI. The E-D product is given by:

$$ED = E \cdot D = P \cdot D^2 = P \cdot (N_{cycles} \cdot T_{clock})^2. \tag{7.12}$$

Although the E-D metric presents important advantages, like the reduced dependence from technology, clock speed and implementations, we also present our measurements for energy consumption. As argued in [14], the energy consumption is an important metric for battery-operated processors in portable devices, because it determines their battery duration.

7.3 Experimental Evaluation

7.3.1 Tools and Benchmarks

In order to evaluate the efficiency of various cache sizes and organizations, the QuakeIII Arena game was used. This benchmark requires an OpenGL compli-

ant library, for which we used Mesa. Since the source of the Mesa library is freely available, we could instrument the code to generate address traces. We used our own trace-driven cache simulator (called BOCS) to gather information such as the number of cache accesses and transition counts. This information was subsequently fed to CACTI, which produces power, energy and energy-delay estimates. We did not use all GraalBench workloads because at the time the research presented in this chapter was performed, GraalBench has not been defined yet.

Details on Tools Settings

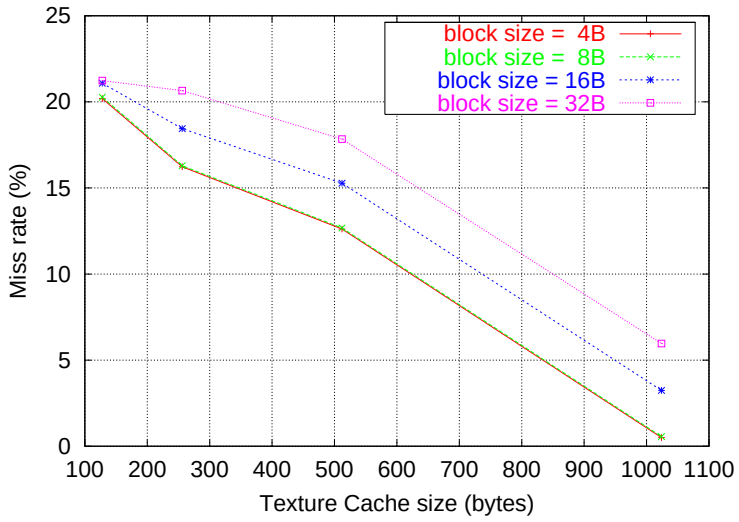
The QuakeIII Arena application was chosen due to its flexibility and a rich set of features that allowed us to customize the benchmark to our needs. The graphic options were set to "High Image Quality" (trilinear filtering) and the screen resolution was chosen as "640x480". One of the features of QuakeIII is to be able to run a prerecorded demo as a reference. The first available record (demo001) was used and we gathered statistics for all frames starting from frame 40 (the first 40 frames are not relevant since they are introductory frames).

The Mesa library was instrumented with new code so that each reference to a 2D texture element was logged to a file. Write operations were not logged since we are interested in read operations mostly and assume that the number of writes is insignificant compared to the number of reads. Furthermore, write operations are encountered at the initialization phase only. The address trace file obtained from Mesa was simulated using BOCS for several cache and block sizes. We only consider direct-mapped caches, because associativity increases the amount of data and control information read out on each cache access, and therefore associative caches consume more power [47]. The assumptions we made for the cache simulator are that the word size is 32 bits and that the miss penalty is $12 + w$ cycles, where w is the block size in words.

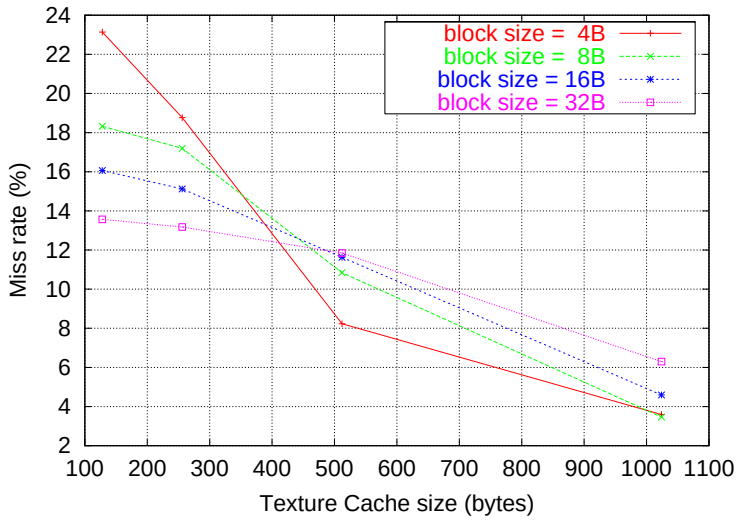
7.3.2 Experimental Results

At first, we used the Mipmap benchmark from the Mediabench suite [48] instead of the QuakeIII Arena Demo. However, this benchmark showed anomalous behavior, as is explained below.

Figure 7.4(a) depicts the miss rates for the texture mapping phase of the Mipmap benchmark as a function of the cache size, for four different block

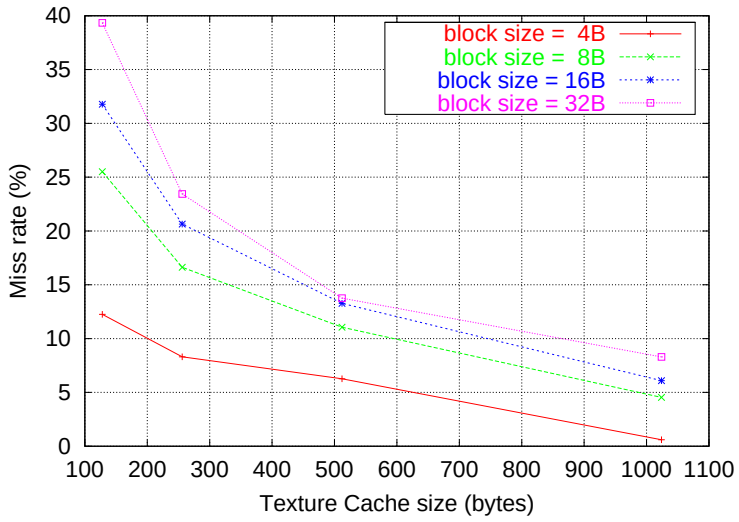


(a) Behavior using original benchmark

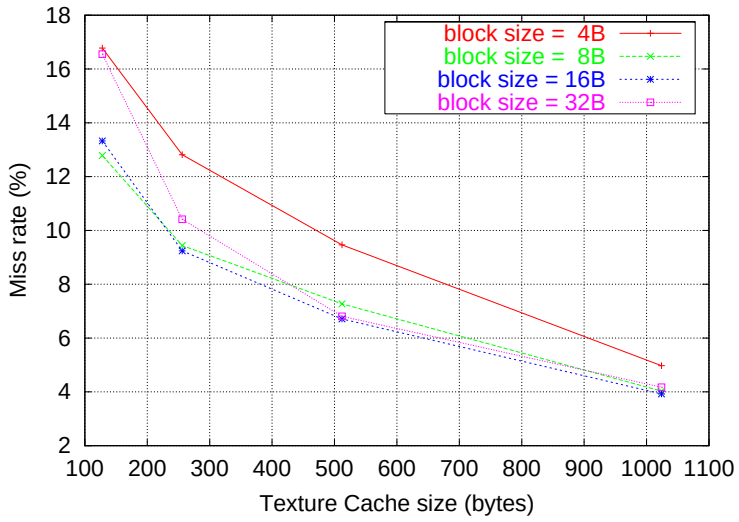


(b) Realistic behavior

Figure 7.4: Mipmap miss rates using point sampling.



(a) Behavior using original benchmark



(b) Realistic behavior

Figure 7.5: Mipmap miss rates using trilinear interpolation.

sizes. Somewhat surprisingly, the miss rate increases with the block size. This effect became even more significant when we changed the sampling method from point sampling to the more often used trilinear interpolation (Figure 7.5(a)). Although it can be expected that for small caches small block sizes are better due to the small number of entries and the risk of cache conflicts, it is really surprising that, for example, a 256-byte cache with a block size of 8 bytes exhibits a higher miss rate than a 128-byte cache with a block size of 4 bytes, even though they have the same number of entries. Furthermore, as explained in the introduction, trilinear filtering should be able to take advantage of spatial locality. We, therefore, studied the code of the Mipmap benchmark, and discovered that there was not only a high degree of spatial locality, but also a high degree of temporal locality because most of the texels from the texture were covering more than one pixel. So, the same texel was reused for the coverage of more than one pixel. Such a behavior is not realistic since it can be encountered only when, for instance in a 3D game, we are very close to a wall and we have a low resolution texture. Figures 7.4(b) and 7.5(b) show the miss rates after modifying the number of tiles applied on the respective surface. They show that this indeed changes the cache behavior, and that the Mipmap benchmark is not representative. We therefore used the QuakeIII Arena Demo instead.

The results obtained were very similar across frames. We, therefore, present only the results for one representative frame (Frame 520).

Table 7.2 depicts the power consumed by the texture cache when frame 520 is processed as a function of the cache size and block size. As can be expected, the smaller the cache and the block size, the lower the power consumption. One interesting observation about this table is that doubling the block size does not imply a similar increase in power consumption, although there are twice as many lines. This can be explained by the improved hit rate that compensates for the increased power per reference, and by the spatial locality of references (larger blocks imply fewer address line transitions).

Table 7.3 shows the energy (power times delay) consumption as a function of the cache and block size. In this case the best results are not necessarily obtained for the smallest cache and block sizes, since they incur a large delay. It appears that the best results are achieved when the cache size is 128 or 256 bytes, and that the block size is less relevant. Although employing larger blocks improves the hit rate and reduces the number of address line transitions, this appears to be almost neutralized by the increase in the number of bitline and wordline transitions.

The miss rate as a function of the cache and block size is depicted in Table 7.4. It shows that the 128-byte cache is too small to be useful as a texture cache, since it incurs miss rates in the order of 20-50%. However, a 256-byte texture cache with a block size of 16 bytes already reduces the miss rate to less than 3.7%. We also observe that although the miss rate decreases when we increase the block size, the improvements quickly become smaller.

However, the most important metric for our study is the energy-delay product, which is shown in Table 7.5. As can be seen, the 256-byte cache with a 16-byte block size yields the best result. This was the case for most frames, but for some frames characterized by intense geometry requirements, higher polygon counts and higher overdraw, the 512-byte cache with a block size of 16 bytes yielded the best result. To confirm this, Table 7.6 depicts the energy-delay product for one of these frames.

For comparison reasons, we have also included the results for a conventional cache size (16KB) in Table 7.7. It can be verified that a 16KB cache consumes much more power and energy than a small texture cache, but of course, it exhibits a lower miss rate. Overall, however, the small cache yields a much smaller energy-delay product. For example, when processing frame 520, the 256-byte cache with a block size of 16 bytes improves the energy-delay product by a factor of 2.1 compared to a 16KB cache with the same block size.

Cache Size (B)	Block Size (B)			
	4	8	16	32
128	11.16	13.47	13.67	13.45
256	39.02	45.87	46.39	48.90
512	49.72	59.23	61.94	59.15
1024	64.75	77.14	83.80	85.03

Table 7.2: Power for Frame 520 (mW).

In order to emphasize the efficiency of the 256-byte texture cache with a block size of 16 bytes, we depict in Figure 7.6 the hit rate for each frame for 1340 consecutive frames. Although the hit rate varies between 59.37% and 98.72%, the average is 92.88% which indicates that even such a small cache size is suitable for implementation. Figure 7.7 depicts the total number of texture accesses per frame and the number of corresponding hits so that the hit rate variation can be related to scene complexity for each frame. As expected, the hit rate usually drops when the scene becomes more complex.

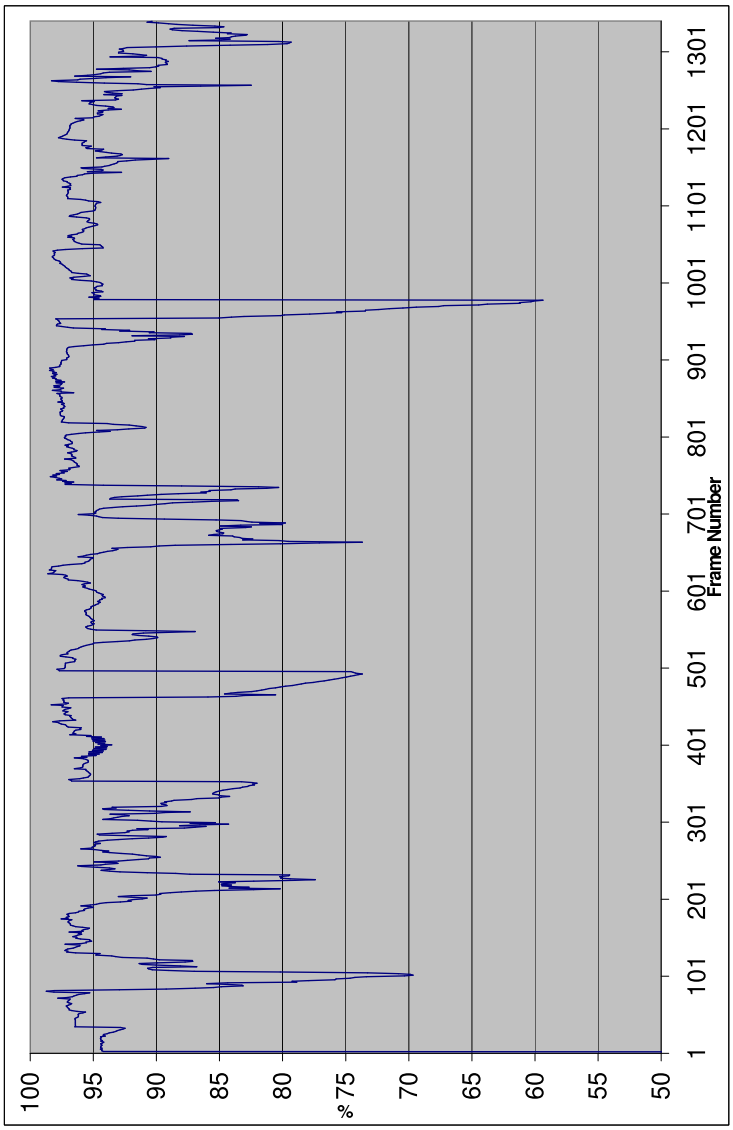


Figure 7.6: The hit rate for a 256-byte cache with a block size of 16 bytes.

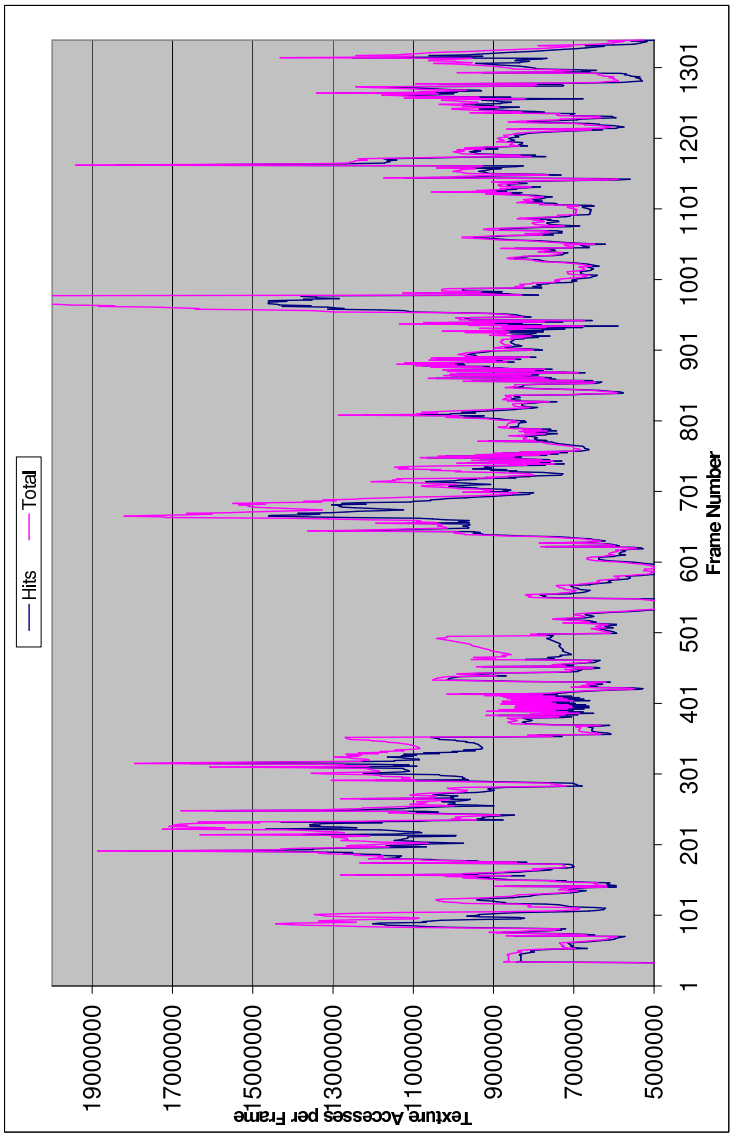


Figure 7.7: The total number of accesses and the hit rate for a 256-byte cache with a block size of 16 bytes.

Cache Size (B)	Block Size (B)			
	4	8	16	32
128	7.34	7.11	6.74	7.39
256	7.08	7.11	6.85	7.54
512	7.71	8.02	8.03	7.96
1024	9.03	9.51	9.87	10.20

Table 7.3: Energy for Frame 520 (mJ).

Cache Size (B)	Block Size (B)			
	4	8	16	32
128	46.67	33.39	26.89	24.51
256	7.31	4.67	3.67	3.28
512	5.14	3.25	2.46	2.22
1024	3.84	2.33	1.66	1.49

Table 7.4: Miss Rate for Frame 520 (%).

Cache Size (B)	Block Size (B)			
	4	8	16	32
128	4.82	3.75	**3.32**	4.05
256	1.28	1.10	**1.01**	1.16
512	1.20	1.09	**1.04**	1.07
1024	1.26	1.17	**1.16**	1.23

Table 7.5: Energy*Delay for Frame 520 (mJ*s). For each cache size, the block size that achieves the best result is shown in bold.

Cache Size (B)	Block Size (B)			
	4	8	16	32
128	11.40	8.75	**7.77**	9.53
256	5.12	4.25	**3.93**	4.74
512	2.57	2.34	**2.25**	2.33
1024	2.50	**2.39**	2.44	2.66

Table 7.6: Energy*Delay for Frame 120 (mJ*s). For each cache size, the block size that achieves the best result is shown in bold.

	Block Size (B)			
	4	8	16	32
E*d	2.39	2.33	2.15	3.14
E	21.70	22.70	21.60	32.00
P	197.93	220.95	217.50	326.54
Miss Rate	1.39	0.75	0.43	0.27

Table 7.7: Results for a 16KB cache using Frame 520.

Overall, a 256-byte texture cache with a block size of 16 bytes yields the lowest energy-delay product. If we assume that texel addresses are 32 bits long and that memory is byte-addressable, each tag is 24 bits since the byte offset is 4 bits and the number of sets 16 and, hence, 4 bits of index. With a valid bit, the total implementation cost of this cache is $(1 + 24 + 128) \times 16 = 2,448$ bits. A 16KB cache with a block size of 16 bytes, on the other hand, requires $(1 + 18 + 128) \times 1024 = 147$ Kbits. In other words, the total implementation cost of the 256-byte cache is smaller than the 16KB cache by a factor of 61.5. Furthermore, a 16KB cache would require a large part of the at most 500K gates available for graphics accelerators for mobile devices (see Section 1.1.3).

7.4 Conclusions

In this chapter we have proposed and evaluated using a small cache between the graphics accelerator and the texture memory in portable graphics devices. Such devices cannot afford large caches due to restrictions such as gate count limitations and the need for low power.

For most frames in the used benchmark a cache of 256 bytes and with a block size of 16 bytes yields the best results w.r.t. the Energy-Delay metric, but for some frames better results are obtained when a 512-byte cache with a block size of 16 bytes is used. The performance penalty compared to a conventional cache size of 16KB is approximately a factor of 1.5, and it remains to be evaluated if this is acceptable. We have also shown that a widely-used benchmark exhibits anomalous behavior.

In the future we intend to evaluate the robustness of the texture cache architecture by conducting experiments using other scenes with different characteristics, and also the influence of blocked texture representation and texture compression on energy efficiency of small caches.

Chapter 8

Conclusions

In this dissertation we have investigated 3D graphics accelerators based on the tile-based rendering paradigm. In essence, we have studied the advantages of tile-based rendering over traditional rendering. In order to have representative workloads for our study we have also developed a 3D graphics benchmarking set suitable for low-power mobile devices. A significant effort has been made to design efficient tile-based algorithms for tile-based scene and state management. This chapter summarizes our investigations and achievements as follows. In Section 8.1 we discuss the overall conclusions. Section 8.2 presents our main contributions. In Section 8.3 further research directions are proposed.

8.1 Summary

The work presented in this thesis can be summarized as follows.

In Chapter 2 we have provided a brief survey of the 3D Graphics pipeline. The application stage generates a high level description of a 3D world. The geometry stage performs the transform and lighting computations. And finally, the rasterizer stage transforms the rendering primitives (points, lines, and triangles) into pixels. Out of the three stages, we have focused on the rasterizer stage since this stage is usually responsible for the majority of external data traffic which represents one of the most important design factors for a low power device.

In Chapter 3 we have shown that 3D benchmarks employed for desktop computers are not suitable for mobile environments. Consequently, we have pre-

sented GraalBench, a set of 3D graphics workloads representative for contemporary and emerging mobile devices. In addition, we have presented detailed simulation results for a typical rasterization pipeline. The presented results have shown that the proposed benchmarks use only a part of the resources offered by current 3D graphics libraries. Finally, we have discussed the architectural implications of the obtained results for hardware implementations.

In Chapter 4 we have presented a comparison of the total amount of external data traffic required by traditional and tile-based renderers. For tile-based renderers, based on the total data traffic variation with respect to the on-chip memory (tile size), a tile size of 32×32 pixels has been found to yield the best trade-off between the amount of on-chip memory and the amount of external data traffic. While tile sizes such as 64×64 pixels might reduce the external data traffic, the number of gates required to implement such tiles on-chip can easily exceed gates budgets for current low-power devices. For example, for a 64×64 elements tile, where each element has 32 bits allocated for RGBA color, 24 bits allocated for depth, and 8 bits for stencil, the number of bits required to implement the tile is 256Kbits. In a SRAM implementation (6 gates per bit), this represents 1536k gates which exceeds the gate budget mentioned in Section 1.1.3.

We have also shown that tile-based rendering reduces the total amount of external traffic due to the considerable data traffic reduction between the accelerator and the off-chip memory while maintaining an acceptable increase in data traffic between the CPU and the renderer. Considering that external memory accesses consume a significant amount of power, this indicates that tile-based rendering might be a suitable technique for low-power embedded 3D graphics implementations. We mention, however, that the reduction in bandwidth of tile-based rendering when compared to traditional rendering depends significantly on the workload used.

In Chapter 5 we have presented several algorithms for sorting the primitives into bins and evaluated their computational complexity and memory requirements. In addition, we have described and evaluated several tests for determining if a triangle and a tile overlap. Experimental results obtained using GraalBench show that various trade-offs can be made and that, usually, better performance can be obtained by trading it for memory. This information allows the designer to select the appropriate method depending on the amount of memory available and the computational power.

In this chapter, a two stage model for triangle to tile repartition for tile-based graphics accelerators has been presented. In addition, different algorithms to

test the triangle to tile overlap have been described. The DIRECT algorithm, which scans the entire scene buffer for each tile and sends the primitives that overlap the current tile to the accelerator, is probably not a practical algorithm because it has poor performance. However, the TWO_STEP algorithm, which stores the bounding box of each triangle and therefore avoids recomputing it, even though it also scans the entire scene buffer for each tile, has reasonable performance while it does not require a large amount of additional memory. If the computational performance is more important than additional memory required, then the SORT algorithm, which for each tile adds pointers to the primitives that overlap it, is more suitable.

In this chapter we have also shown that the efficiency of the bounding box test can be improved significantly by adaptively varying the order in which the comparisons are performed depending on the position of the current tile. Experimental results obtained using several 3D graphics workloads show that the dynamic bounding box test reduces the average number of comparisons per primitive by 26% on average compared to the best performing static version in which the order of the comparisons is fixed.

In Chapter 6 we have presented several state management algorithms for tile-based renderers. While in traditional (non tile-based) rendering the state information traffic can be negligible compared to the traffic generated by the primitives, in tile-based rendering architectures, since the state information might need to be duplicated in multiple streams, the required processing power and generated traffic can increase significantly. Removing primitives from the instruction stream of a tile depends only on the primitive position and the tile coordinate, but to remove a state change instruction from the instruction stream of a tile, information about the previous or the following state change instructions and/or primitives is required. For instance, an instruction that enables depth comparison, can be removed if there is an instruction that disables depth comparison following it and there are no primitives between them. Thus, in order to send an optimal state change stream to the accelerator, i.e., use minimal bandwidth, additional processing power and more processor bandwidth is required. By sending an optimal state change stream to the accelerator, the state change traffic to the accelerator was decreased by up to 58% compared to the state change traffic generated by directly sending the intial state.

In Chapter 7 we have proposed and evaluated using a small filter cache between the graphics accelerator and the texture memory in portable graphics devices. Such devices cannot afford large caches due to restrictions such as gate count limitations and the need for low power. For most frames in the used record

a cache of 256 bytes and with a block size of 16 bytes yields the best results w.r.t. the Energy-Delay metric, but for some frames better results are obtained when a 512-byte cache with a block size of 16 bytes is used. The performance penalty compared to a conventional cache size of 16KB is approximately a factor of 1.5, and it remains to be evaluated if this is acceptable. We have also shown that a widely-used benchmark exhibits anomalous behavior.

8.2 Main Contributions

In this section we emphasize the main contributions of our research that is described in this dissertation.

- We have presented GraalBench, a set of 3D graphics workloads representative for contemporary and emerging mobile devices. In addition, we have presented detailed simulation results for a typical rasterization pipeline. We have shown that the proposed benchmarks use only a part of the resources offered by current 3D graphics libraries. We have also provided evidence indicating that 3D benchmarks employed for desktop computers are not suitable for mobile environments.

- We have proposed several algorithms for sorting the primitives into bins and evaluated their computational complexity and memory requirements. In addition, we have described and evaluated several tests for determining if a triangle and a tile overlap. Based on the obtained results, a designer can select the appropriate method depending on the amount of memory available and the computational power.

- We have presented a dynamic bounding box test. We have shown that the efficiency of the bounding box test can be improved significantly by adaptively varying the order in which the comparisons are performed depending on the position of the current tile. Experimental results obtained using several 3D graphics workloads have show that the dynamic bounding box test reduces the average number of comparisons per primitive by 26% on average compared to the best performing static version in which the order of the comparisons is fixed.

- We have presented a “delayed execution” state management algorithm for tile-based renderers. While in traditional (non tile-based) rendering the state information traffic can be negligible compared to the traffic generated by the primitives, in tile-based rendering architectures, since

the state information might need being duplicated in multiple streams, the required processing power and generated traffic can increase significantly.

The "delayed execution" state management commits only the necessary state changes to the accelerator, thus using a minimal amount of bandwidth.

- We have proposed and evaluated using a small cache between the graphics accelerator and the texture memory in portable graphics devices. Such devices cannot afford large caches due to restrictions such as gate count limitations and the need for low power. For most frames in the used record a cache of 256 bytes and with a block size of 16 bytes yields the best results w.r.t. the Energy-Delay metric, but for some frames better results are obtained when a 512-byte cache with a block size of 16 bytes is used. We have also shown that a widely-used benchmark exhibits anomalous behavior.

- We have presented a comparison of the total amount of external data traffic required by traditional and tile-based renderers while previous work has focused mainly on determining the overlap.

  For tile-based renderers, we found that a tile size of 32×32 pixels yields the best trade-off between the amount of on-chip memory and the amount of external data traffic.

  Moreover, we have shown that the reduction in bandwidth of tile-based rendering when compared to traditional rendering depends significantly on the workload used. The tile-based rendering is most suitable for workloads with a low overlap factor and high overdraw.

8.3 Future Directions

As a continuation of the research the following topics are suggested.

- The number of components for the GraalBench benchmark suite should be extended to include OpenGL ES applications or the current components can be translated to use the OpenGL ES interface. Since OpenGL ES is widely available on low-power 3D graphics platforms, this extensions would provide a relevant workload for upcoming generations of 3D graphics mobile devices.

- The texture cache investigation can be extended to by conducting experiments using other scenes with different characteristics.
- The obtained results for tile based architecture data traffic can be extended by investigating the impact of using multitexturing for tile-based architectures. Furthermore, since the amount of texture traffic is significant for a tile-based renderer, the effect of using various texture compression techniques can be investigated.
- Evaluation of non-tile-based low-power techniques (e.g., [9]) using GraalBench and comparison to tile-based.

Bibliography

[1] ATI Technologies Inc., http://www.ati.com.

[2] Bitboys Oy, http://www.bitboys.com.

[3] Falanx Microsystems AS, http://www.falanx.no.

[4] Imagination Technologies Ltd., http://www.imgtec.com.

[5] NVIDIA Corporation, http://www.nvidia.com.

[6] Portable 3d research group at korea advanced institute of science and technology, http://ssl.kaist.ac.kr/.

[7] Yonsei university, 3d graphics accelerator group, http://msl.yonsei.ac.kr/3d/.

[8] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering (second edition)*. A.K. Peters Ltd., 2002.

[9] Tomas Akenine-Möller and Jacob Ström. Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones. *ACM Trans. Graph.*, 22(3):801–808, 2003.

[10] Iosif Antochi, Ben Juurlink, Andrea Cilio, and Petri Liuha. Trading Efficiency for Energy in a Texture Cache Architecture. *Proc. Euromicro Conf. on Massively-Parallel Computing Systems (MPCS'02)*,2002, Ischia, Italy, pp. 189-196.

[11] ARM Ltd. ARM 3D Graphics Solutions.

[12] Portable 3D Research Group at Korea Advanced Institute of Science and Technology. MobileGL - The Standard for Embedded 3D Graphics. Available at http://ssl.kaist.ac.kr/projects/portable3D.html/main_mgl_defnition.htm.

[13] A.C. Barkans. High Quality Rendering Using the Talisman Architecture. In *Proc. The 1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pp. 79–88, Los Angeles, August 1997.

[14] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proc. 27th Annual Int. Symp. on Computer Architecture*, pages 83–94, Vancouver, British Columbia, June 2000.

[15] J. Adam Butts and Gurindar S. Sohi. A Static Power Model for Architects. In *Proc. 33rd Annual ACM/IEEE Int. Symp. on Microarchitecture (MICRO 33)*, pages 191–201, New York, NY, USA, 2000. ACM Press.

[16] Francky Catthoor, Frank Franssen, Sven Wuytack, Lode Nachtergaele, and Hugo De Man. Global Communication and Memory Optimizing Transformations for Low-Power Signal Processing Systems. In *Proc. VLSI Signal Processing Workshop*, 1994.

[17] Milton Chen, Gordon Stoll, Homan Igehy, Kekoa Proudfoot, and Pat Hanrahan. Simple Models of the Impact of Overlap in Bucket Rendering. In *Proc. ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 105–112, Lisbon, Portugal, 1998. ACM Press.

[18] Futuremark Corporation. 3DMark01SE. Available at http://www.futuremark.com/products/3dmark2001/.

[19] Futuremark Corporation. 3DMark03. Available at http://www.futuremark.com/products/3dmark03/.

[20] Futuremark Corporation. SPMark04. Available at http://www.futuremark.com/products/spmark04/.

[21] Michael Cox and Narendra Bhandari. Architectural Implications of Hardware-Accelerated Bucket Rendering on the PC. In *Proc. 1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 25–34. ACM Press, 1997.

[22] F.C. Crow. Summed-Area Tables for Texture Mapping. *Computer Graphics* (Proc. Siggraph), Vol. 13, No. 3, 207-212, 1984.

[23] J.C. Dunwoody and M.A. Linton. Tracing Interactive 3D Graphics Programs. *Proc. ACM Symp. on Interactive 3D Graphics*, 1990.

[24] David H. Eberly. Intersection of Convex Objects: The Method of Separating Axes. http://www.magic-software.com/.

[25] David H. Eberly. *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*. Morgan Kaufmann/Academic Press, 2001.

[26] Jon P. Ewins, Marcus D. Waller, Martin White, and Paul F. Lister. MIP-Map level selection for texture mapping. *IEEE Transactions on Visualization and Computer Graphics*, 4(4), 1998.

[27] Falanx. Falanx Microsystems AS. Available at http://www.falanx.no/download/Mali55_Mali110_Product.pdf.

[28] Simon Fenney. Texture Compression using Low-Frequency Signal Modulation. In *Proc. ACM SIGGRAPH/EUROGRAPHICS Conf. on Graphics Hardware (HWWS '03)*, pages 84–91, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[29] R. F. Ferraro. *Programmer's Guide to the EGA, VGA, and Super VGA Cards*. Third Edition. Addison-Wesley, 1994.

[30] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics, Principles and Practice, Second Edition*. Addison-Wesley, Reading, Massachusetts, 1990.

[31] Richard Fromm, Stylianos Perissakis, Neal Cardwell, Christoforos Kozyrakis, Bruce McGaughy, David Patterson, Tom Anderson, and Katherine Yelick. The Energy Efficiency of IRAM Architectures. In *Proc.* 24^{th} *Annual Int. Symp. on Computer Architecture*, pages 327–337. ACM Press, 1997.

[32] Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories. In *Proc. of the* 16^{th} *Annual Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH '89)*, pages 79–88, New York, NY, USA, 1989. ACM Press.

[33] R. Gonzalez and M. Horowitz. Energy Dissipation in General Purpose Microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1258–1266, 1996.

[34] JSR-184 Expert Group. Mobile 3D Graphics API for Java™2 Micro Edition. Available at http://jcp.org/aboutJava/communityprocess/final/jsr184/index.html.

[35] The Khronos Group. OpenGL ES Overview. Available at http://www.khronos.org/opengles/index.html.

[36] Z.S. Hakura and A. Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. In *Proc. 24th Internationnal Symposium on Computer Architecture*, pages 108–120, 1997.

[37] P. Heckbert. Survey of Texture Mapping. *IEEE Computer Graphics and Applications*, pp. 56-67, November, 1986.

[38] P. Heckbert. Fundamentals of Texture Mapping and Image Warping. Master's Thesis (Technical Report No. ucb/csd 89/516), University of California, Berkeley, 1989.

[39] Emile Hsieh, Vladimir Pentkovski, and Thomas Piazza. ZR: A 3D API Transparent Technology for Chunk Rendering. In *Proc. 34th ACM/IEEE Int. Symp. on Microarchitecture (MICRO-34)*, 2001.

[40] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: A Stream Processing Framework for Interactive Rendering on Clusters. In *Proc. 29th Annual Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH 2002)*, pages 693–702, 2002.

[41] H. Igehy, M. Eldridge, and K. Proudfoot. Prefetching in a Texture Cache Architecture. In *Proc. EUROGRAPHICS/SIGGRAPH Workshop on Graphics Hardware*, pages 133–142, 1998.

[42] Systems in Motion. VRMLView. Available at http://www.sim.no.

[43] Id Software Inc. Quake III, Available at http://www.idsoftware.com.

[44] Masatoshi Kameyama, Yoshiyuki Kato, Hitoshi Fujimoto, Hiroyasu Negishi, Yukio Kodama, Yoshitsugu Inoue, and Hiroyuki Kawai. 3D Graphics LSI Core for Mobile Phone "Z3D". In *Proc. ACM SIGGRAPH/EUROGRAPHICS Conf. on Graphics Hardware (HWWS '03)*, pages 60–67, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[45] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. *SIGARCH Computer Architecture News*, 29(2):240–251, 2001.

[46] Chun-Ho Kim and Lee-Sup Kim. Adaptive Selection of an Index in a Texture Cache. In *Proc. IEEE Int. Conf. on Computer Design (ICCD'04)*, pages 295–300, Washington, DC, USA, 2004. IEEE Computer Society.

[47] Johnson Kin, Munish Gupta, and William H. Mangione-Smith. Filtering Memory References to Increase Energy Efficiency. *IEEE Trans. on Computers*, 49(1), Jan 2000.

[48] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proc. 30th Annual Int. Symp. on Microarchitecture*, pages 330–335, 1997.

[49] B.-S. Liang and C.-W. Jen. Computation-effective 3-d graphics rendering architecture for embedded multimedia system. *IEEE Transactions on Consumer Electronics*, 46(3):735–743, August 2000.

[50] Joel McCormack, Robert McNamara, Christopher Gianos, Larry Seiler, Norman P. Jouppi, and Ken Correll. Neon: A Single-chip 3D Workstation Graphics Accelerator, booktitle = Proc. of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware (HWWS '98), year = 1998, isbn = 0-89791-097-X, pages = 123–132, location = Lisbon, Portugal, doi = http://doi.acm.org/10.1145/285305.285320, publisher = ACM Press, address = New York, NY, USA,.

[51] T. Mitra and T. Chiueh. Dynamic 3D Graphics Workload Characterization and the Architectural Implications. *Proc. 32nd ACM/IEEE Int. Symp. on Microarchitecture (MICRO)*, 1999, pp. 62-71.

[52] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Comput. Graph. Appl.*, 14(4):23–32, 1994. IEEE Computer Society Press.

[53] Carl Mueller. The Sort-First Rendering Architecture for High-Performance Graphics. In *Proc. 1995 Symp. on Interactive 3D Graphics*, pages 75–84. ACM Press, 1995.

[54] Marc Olano and Trey Greer. Triangle scan conversion using 2d homogeneous coordinates. In *Proc. ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics Hardware(HWWS '97)*, pages 89–95, New York, NY, USA, 1997. ACM Press.

[55] M. Pichler, G. Orasche, K. Andrews, E. Grossman, and M. McCahill. VRweb: a Multi-System VRML Viewer. *Proc. First Symp. on Virtual Reality Modeling Language*, 1995, San Diego, California, United States, pp. 77-85.

[56] Juan Pineda. A Parallel Algorithm for Polygon Rasterization. In *Proc. 15th Annual Conference on Computer Graphics and Interactive Techniques*, pages 17–20. ACM Press, 1988.

[57] PowerVR. 3D Graphical Processing (Tile Based Rendering - The Future of 3D), White Paper. http://www.beyond3d.com/reviews/videologic/vivid/PowerVR_WhitePaper.pdf, 2000.

[58] The Mesa Project. The Mesa 3D Graphics Library. Available at http://www.mesa3d.org.

[59] G. Reinman and N.P. Jouppi. An Integrated Cache Timing and Power Model. Technical report, COMPAQ Western Research Lab, Palo Alto, California, 1999.

[60] Rudrajit Samanta and Thomas Funkhouser. Dynamic Algorithms for Sorting Primitives Among Screen-Space Tiles in a Parallel Rendering System. Technical report, Department of Computer Science, Princeton University, Available at http://www.cs.princeton.edu/~funk/sorting.pdf, 1998.

[61] A. Schilling, G. Knittel, and W. Strasser. Texram: A Smart Memory for Texturing. IEEE Computer Graphics and Applications, 16(3), 32-41, 1996.

[62] Andreas Schilling. A New Simple and Efficient Antialiasing With Subpixel Masks. In *Proc. 18th Annual Conference on Computer Graphics and Interactive Techniques*, pages 133–141. ACM Press, 1991.

[63] Dave Schreiner, Dave (Ed.) Schreiner, and Dave Shreiner. *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[64] M. Segal and K. Akeley. The OpenGL TM Graphics System: A Specification. Silicon Graphics, 1999.

[65] Mark Segal and Kurt Akeley. *The $OpenGL^{TM}$ Graphics System: A Specification*. Silicon Graphics, April 1999.

[66] Hawk Software. GLTrace Programming Utility. Available at http://www.hawksoft.com/gltrace.

[67] J. Sohn, R. Woo, and H.J. Yoo. Optimization of Portable System Architecture for Real-Time 3D Graphics. *Proc. IEEE Int. Symp. on Circuits and Systems (ISCAS 2002)*, Volume: 1 , 26-29 May 2002 pp. I-769 - I-772 vol.1.

[68] SourceForge. spyGLass: an OpenGL Call Tracer and Debugging Tool. Available at http://www.sourceforge.net/projects/spyglass.

[69] SPEC. SPECviewperf 6.1.2. Available at http://www.specbench.org/gpc/opc.static/opcview.htm.

[70] Sunspire Studios. Tux Racer. Available at http://tuxracer.sourceforge.net/.

[71] PowerVR (Imagination Technologies). Imagination Technologies' PowerVR^{TM} in STMicroelectronics' KYRO^{TM} PC Graphics Accelerator Unveiled. Available at http://www.imgtec.com/News/Release/index.asp?ID=548.

[72] PowerVR (Imagination Technologies) and ARM. MBX Acceleration. Available at http://www.arm.com/products/esd/multimediagraphics_mbx.html.

[73] Jay Torborg and James T. Kajiya. Talisman: Commodity Realtime 3D Graphics for the PC. In *Proc. 23^{rd} Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH' 96)*, pages 353–363, New York, NY, USA, 1996. ACM Press.

[74] Lance Williams. Pyramidal Parametrics. In *Proc. 10^{th} Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH: 83)*, pages 1–11, New York, NY, USA, 1983. ACM Press.

[75] S.J.E. Wilton and N.P. Jouppi. An Enhanced Access and Cycle Time Model. Technical Report 5, Digital Western Research Laboratory, Palo Alto, California, July 1994.

List of Publications

International Conferences

- **I. Antochi**, B.H.H. Juurlink, S. Vassiliadis, P. Liuha, "Efficient Tile-Aware Bounding-Box Overlap Test for Tile-Based Rendering", Proceedings of the 2004 International Symposium on System-on-Chip 2004, pp. 165-168, Tampere, Finland, November 2004
- **I. Antochi**, B.H.H. Juurlink, S. Vassiliadis, P. Liuha, "Memory Bandwidth Requirements of Tile-Based Rendering", Proceedings of the Third and Fourth International Workshops SAMOS 2003 and SAMOS 2004 (LNCS 3133), pp. 323-332, Samos, Greece, July 2004
- **I. Antochi**, B.H.H. Juurlink, S. Vassiliadis, P. Liuha, "Scene Management Models and Overlap Tests for Tile-Based Rendering", Proceedings of the EUROMICRO Symposium on Digital System Design, 2004 (DSD 2004), pp. 424 - 431, Rennes, France, August 2004
- **I. Antochi**, B.H.H. Juurlink, S. Vassiliadis, P. Liuha, "GraalBench: A 3D Graphics Benchmark Suite for Mobile Phones", Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools, pp. 1-9, Washington, DC, USA, June 2004
- **I. Antochi**, B.H.H. Juurlink, A. G. M. Cilio, P. Liuha, "Trading Efficiency for Energy in a Texture Cache Architecture", Proceedings of the 2002 Euromicro Conference on Massively-Parallel Computing Systems, pp. 189-196, Ischia, Italy, April 2002

Local Conferences

- **I. Antochi**, B.H.H. Juurlink, S. Vassiliadis, P. Liuha, "Efficient State Management for Tile-Based 3D Graphics Architectures", Proceedings

of the 15th Annual Workshop on Circuits, Systems and Signal Processing, ProRISC 2004, pp. 336-340, Veldhoven, The Netherlands, November 2004

- **I. Antochi**, B.H.H. Juurlink, S. Vassiliadis, P. Liuha, "3D Graphics Benchmarks for Low-Power Architectures", Proceedings of the 14th Annual Workshop on Circuits, Systems and Signal Processing, ProRISC 2003, pp. 18-22, Veldhoven, The Netherlands, November 2003
- **I. Antochi**, B.H.H. Juurlink, S. Vassiliadis, "Selecting the Optimal Tile Size for Low-Power Tile-Based Rendering", Proceedings ProRISC 2002, pp. 1-6, Veldhoven, The Netherlands, November 2002
- **I. Antochi**, B.H.H. Juurlink, S. Vassiliadis, "A Flexible Simulator for Exploring Hardware Rasterizers", 13th Annual Workshop on Circuits, Systems, and Signal Processing (ProRISC2002), Veldhoven, The Netherlands, November 2002

Deliverables and Technical Reports

- **I. Antochi**, B. Juurlink, S. Vassiliadis, "A Low Power 2D/3D Graphics Accelerator – Tracing Interactive OpenGL Applications", Deliverable no. (2002)–04, Computer Engineering Laboratory, Faculty of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology, 2002.
- **I. Antochi**, B. Juurlink, S. Vassiliadis, "A Low Power 2D/3D Graphics Accelerator – A Preliminary ISA ", Deliverable no. (2002)–01, Computer Engineering Laboratory, Faculty of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology, 2002.
- **I. Antochi**, B. Juurlink, S. Vassiliadis, "A Low Power 2D/3D Graphics Accelerator – The Initial Design", Deliverable no. (2001)–03, Computer Engineering Laboratory, Faculty of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology, 2001.
- D. Crisu, **I. Antochi**, S.D. Cotofana, B. Juurlink, S. Vassiliadis, "Low-Power Techniques and 2D/3D Graphics Architectures", Deliverable no. (2001)–01, Computer Engineering Laboratory, Faculty of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology, 2001.

Samenvatting

In dit proefschrift behandelen we architecturen voor acceleratoren die op laag vermogen hoge prestaties kunnen leveren voor 3D grafische toepassingen. Het doel van deze acceleratoren is om de centrale processor te ontlasten van zware grafische berekeningen. Ook draagt het gebruik hiervan bij aan een efficiënter energie verbruik vergeleken met dezelfde berekeningen op de centrale processor. Het meeste dataverkeer in de 3D grafische pijplijn wordt normaal gesproken veroorzaakt door het rasteren. Doordat veel operaties in dit onderdeel op pixels gebeurden, neemt ook de vereiste rekenkracht snel toe. Om deze redenen concentreren we ons in dit proefschrift voornamelijk op het optimaliseren van dataverkeer en berekeningen in dit onderdeel van de grafische pijplijn. Een veelbelovende techniek om de hoeveelheid extern data verkeer in het raster onderdeel van de grafische pijplijn te verminderen is renderen-per-tegel. Bij deze techniek wordt een scène in tegels opgedeeld en worden de tegels één voor één verwerkt. Hierdoor is het mogelijk om de kleur componenten en de Z-waarden van één tegel in een kleine buffer op de chip, zodat alleen de pixel die in de uiteindelijke scène zichtbaar zijn in de externe beeldbuffer opgeslagen hoeven te worden. Acceleratoren voor renderen-per-tegel, daarentegen, vereisen een grote buffer om de af te beelden primitieven in op te slaan. Hoewel er reeds onderzoek is gedaan naar renderen-per-tegel voor hoge-prestatie systemen, bespreken wij met name de toepasbaarheid van deze techniek op apparatuur dat op laag vermogen werkt. Om de verscheidene grafische architecturen die op laag vermogen werken te evalueren, presenteren wij eerst GraalBench. Dit is een verzameling grafische werklasten die representatief zijn voor hedendaagse en in opkomst zijnde draagbare apparaten. Verder introduceren we een aantal algoritmes voor het renderen-per-tegel, waarmee de scènes en de status kunnen worden geregeld. Vervolgens analyseren we de prestaties van het renderen-per-tegel en vergelijken dit met traditionele oplossingen. Ook onderzoeken we de invloed van de tegelgrootte op de hoeveelheid dataverkeer dat door het raster onderdeel wordt

geproduceerd. Vanwege de grote hoeveelheid lokaliteit in tijd en ruimte bij de aanvragen van texturen, hebben we ook een aantal cache-structuren onder de loep genomen om de hoeveelheid dataverkeer tussen het geheugen en de grafische versneller nog verder te verminderen. De verkregen resultaten laten zien dat het voorgestelde algoritme voor renderen-per-tegel bij het raster onderdeel van de 3D grafische pijplijn effectief de hoeveelheid dataverkeer en de vereiste rekenkracht kan verminderen.

Curriculum Vitae

Iosif Antochi was born in Bucharest, Romania on the 3^{rd} of November 1976. Between 1996 and 2000, he was a student of the Faculty of Automatic Control and Computers at the ”Politehnica” University of Bucharest. In 2000 he was granted a scholarship at the Delft University of Technology where he did his graduation thesis ”RETAS: A Retargetable MIPS Assembler”. In January 2001 he joined as a Ph.D. student the Computer Engineering Group at Delft University of Technology, where he worked on ”Low-Power High-Performance Graphics Architectures”. His research interests includes low-power 3D graphics algorithms and architectures, tile-based rendering systems, 3D graphics workload generation and characterization, computer architectures, multimedia processors, and embedded systems.