GraalBench: A 3D Graphics Benchmark Suite for Mobile Phones

Iosif Antochi
Computer Engineering Laboratory
Delft University of Technology
Mekelweg 4, 2628 CD
Delft, The Netherlands
tkg@ce.et.tudelft.nl

Ben Juurlink
Computer Engineering Laboratory
Delft University of Technology
Mekelweg 4, 2628 CD
Delft, The Netherlands
benj@ce.et.tudelft.nl

Stamatis Vassiliadis
Computer Engineering Laboratory
Delft University of Technology
Mekelweg 4, 2628 CD
Delft, The Netherlands
stamtatis@ce.et.tudelft.nl

Petri Liuha
NOKIA Research Center
Visiokatu-1, SF-33720
Tampere, Finland
petri.liuha@nokia.com

ABSTRACT

In this paper we consider implementations of embedded 3D graphics and 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. For instance, while each benchmark uses the texturing unit for more than 70% of the generated fragments, the alpha unit is employed for less than 13% of the fragments. The Fog unit was used for 84% of the fragments by one benchmark, but the other benchmarks did not use it at all. Our experiments on the proposed suite suggest that the texturing, depth and blending units should be implemented in hardware, while, for instance, the dithering unit may be omitted from a hardware implementation. Finally, we discuss the architectural implications of the obtained results for hardware implementations.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

General Terms

Performance, Measurement

LCTES'04, June 11–13, 2004, Washington, DC, USA.

Keywords

3D graphics benchmarking, embedded 3D graphics architectures.

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 expect that 3D graphics applications will become an important workload of wireless devices. For example, according to [10], 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, several companies and universities have started to develop a low-power 3D graphics accelerator. However, to the best of our knowledge, there is no publicly available benchmark suite that can be used to guide the architectural exploration of such devices.

This paper presents GraalBench, a 3D graphics benchmark suite suitable for 3D graphics on low-power, mobile systems, in particular mobile phones. These benchmarks were collected to facilitate our studies on low-power 3D graphics accelerators in the Graal (GRAphics AcceLerator) project [5]. 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 [18], 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 paper 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 paper is organized as follows. Previous work on 3D graphics benchmarking is described in Section 2. In this section we also give reasons why current 3D graphics benchmarks are not appropriate for mobile environments. Section 3 first explains how the benchmarks were obtained, describes our tracing environment, the simulator we used to collect the statistics and, after that, describes the components of the proposed benchmark suite and presents some general characteristics of the workloads. Section 4 provides a workload characterization of the benchmarks and discusses architectural implications. Conclusions and directions for future work are given in Section 5.

2. RELATED WORK

To the best of our knowledge, 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 [18], 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 (800×600 pixels), while common display resolutions for mobile phones are QCIF (176×144) and QVGA (320×240).
- 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 [11]). 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 [3, 4] based on the DirectX API, they are not suitable for our study since DirectX implementations are available only on Windows systems.

There have been several studies related to 3D graphics workload characterization (e.g., [11, 6]). Most related to our investigation is the study of Mitra and Chiueh [11], 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 [1, 16] 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 [8], Java mobile 3D Graphics API (JSR-184) [7], and Mobile GL [20] 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. THE GraalBench BENCHMARK SET

In this section we describe the environment we used to create the benchmarks, the components of our benchmark set and also some general characteristics of the workloads.

3.1 Tracing Environment

Due to their interactive nature, 3D games are generally not repeatable. In order to obtain a set of repeatable workloads, we traced existing applications logging all OpenGL calls. Our tracing environment consists of two components: a tracer and a trace player. Our tracer is based on GLtrace from Hawksoft [15]. It intercepts and logs OpenGL calls made by a running application, and then calls the OpenGL function invoked by the application. No source code is required provided the application links dynamically with the OpenGL library, meaning that the executable only holds links to the required functions which are bounded to the corresponding functions at run-time. Statically linked applications, in which case the required libraries are encapsulated in the executable image, cannot be traced using this mechanism when the source code is not available.

We improved GLtrace in two ways. First, GLtrace does not log completely reproducible OpenGL calls (for example, textures are not logged). We modified the GLtrace library so that all OpenGL calls are completely reproducible. Second, the trace produced by GLtrace is a text trace, which is rather slow. We improved its performance by adding a binary logging mode that significantly reduces the tracing overhead.

In addition, 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 [14], which is a public-domain implementation of OpenGL.

3.2 The Benchmarks

The proposed benchmark suite consists of the following components:

Q3L and Q3H Quake III [9] 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 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 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

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

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

Figure 1: Screenshots of the GraalBench workloads

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*) [19] 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 1(b).

AWadvs-04 (*AW*) [18] 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 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 [13] to navigate through the scenes, but we found that the VRMLView [12] navigator produces less texture traffic because it uses the `glBindTexture` mechanism. Screenshots of ANL, GRA, and DIN are depicted in Figure 1(d), (e), and (f), respectively.

GraalBench is the result of extensive searching on the World Wide Web. The applications were selected on the basis of several criteria. First, since the display resolution of contemporary mobile phones is at most 320×240, we excluded applications with substantially higher resolution. Specifically, we used a maximum resolution of 640×480. Second, the applications should be relevant for a mobile phone, i.e., it should be likely that it will be offered on a mobile phone. CAD/CAM applications were excluded for this reason. Third, the level of details of the applications should not be too high, because otherwise, most rendered polygons will be smaller than one pixel on the display of a mobile phone. Fourth, and finally, the benchmarks should have different characteristics. For example, several links to 3D games have recently been provided on Mesa's website (`www.mesa3d.org`). 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 2. We expect that more 3D graphics applications for low-power mobile devices will appear when accelerators for these platforms will be introduced.

3.3 General Characteristics

Table 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 320×240 pixels. Since in the near future the typical resolution is expected to increase we decided to use a resolution of 640×480. The Q3L benchmark uses a lower resolution (320×240) 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 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.

Table 2: Stress variation and stress strength on various stages of the 3D graphics pipeline

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

4. WORKLOAD CHARACTERIZATION

This section provides 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.

4.1 Detailed Workload Statistics

One important aspect for 3D graphics benchmarking is to determine possible bottlenecks in a 3D graphics environment since the 3D graphics environment has a 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 transform & lighting (T&L) 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 Figure 4 (c,d,g,h), 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, generate a variable number of triangles while the generated area is almost constant, thus they can be used to profile bottlenecks in the T&L part of the graphics pipeline.

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 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

Table 1: General statistics of the benchmarks

Bench.	Resolution	Frames	Textures (MB)	Received Triangles		Processed Triangles		Area	
				Avg.	Max.	Avg.	Max.	Avg.	Max.
Q3L	320×240	1,379	12.84	4.5k	9.7k	3.25k	6.8k	422k	1,327k
Q3H	640×480	1,379	12.84	4.6k	9.8k	3.36k	6.97k	1,678k	5,284k
Tux	640×480	1,363	11.71	3k	4.8k	1.8k	2.97k	760k	1,224k
AW	640×480	603	3.25	23k	25.7k	10.55k	13.94k	63k	307k
ANL	640×480	600	1.8	4.45k	14.2k	4.45k	14.2k	776k	1,242k
GRA	640×480	599	2.1	4.9k	10.8k	3.6k	6.9k	245k	325k
DIN	640×480	600	1.7	4.15k	4.3k	4.15k	4.3k	153k	259k

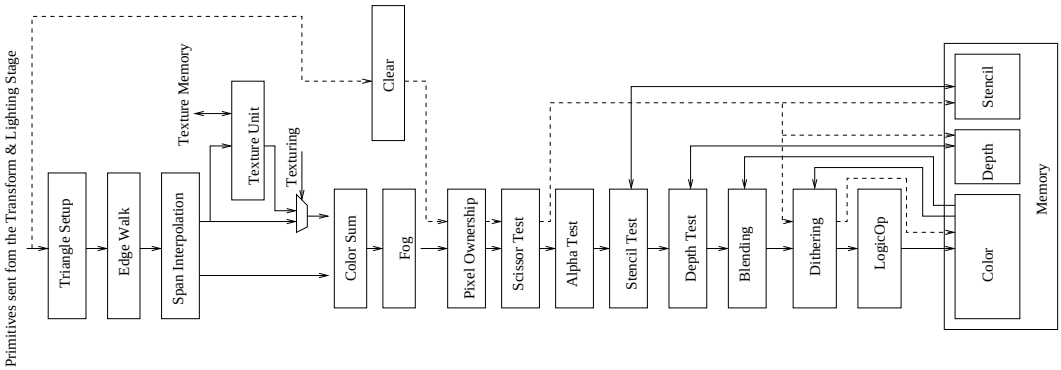


Figure 2: Graphics pipeline for rasterization.

the graphics pipeline is the rasterization part. This is the reason why only the rasterizer stage was additionally studied. The units for which we further gathered results are depicted in Figure 2 and described in the following:

Triangle Setup, EdgeWalk, and Span Interpolation. These units convert primitives to fragments. We used the same algorithm as employed in the Mesa library rasterizer. We remark that the number of processed triangles in Table 3 is smaller than the number of processed triangles in Table 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. The clear unit is used to fill the Depth buffer and/or the Color buffer with a default value. As can be seen in Table 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 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. When enabled, this unit combines the color of the incoming fragment with a texture sample color. 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. 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 locality, even by adding a small texture cache, the traffic from the off-chip texture memory to the texture unit can be substantially reduced[2].

Fog Unit. The Fog unit is to used modify the fragment color in order to simulate atmospheric effects such as haze, mist, or smoke. 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.

Table 3: Triangle Setup, Edge Walk, and Span Interpolation units statistics

	Q3L	Q3H	Tux	AW	ANL	GRA	DIN
Triangles processed	4,147k	4,430k	2,425k	5,537k	2,528k	1,992k	2,487k
Generated spans	58,837k	117,253k	27,928k	20,288k	66,806k	14,419k	23,901k
Generated fragments (frags.)	581,887k	2,306,487k	1,037,222k	38,044k	466,344k	146,604k	91,824k

Table 4: Clear unit statistics

	Q3L	Q3H	Tux	AW	ANL	GRA	DIN
Clear depth calls	5,470	5,470	1,363	603	601	600	602
Clear color calls	1	1	1,363	604	601	600	602
Clear depth pixels	105,821k	423,284k	418,714k	185,242k	184,320k	184,013k	189,934k
Clear color pixels	76,800	307,200	418,714k	185,549k	184,320k	184,013k	189,934k

Scissor Unit. This unit is used to discard fragments that are outside of a specified rectangular area. 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 discards fragments based on their alpha color component. 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 discards a fragment based on a comparison between its depth value and the depth value stored in the depth buffer in the fragment's corresponding position. This unit was used intensively by all benchmarks as can be seen in Table 4.1. 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. This unit combines the color of the incoming fragment with the color stored at the corresponding position in the framebuffer. As depicted in Figure 3, 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 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.

4.2 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 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 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 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

Table 5: Depth unit statistics

	Q3L	Q3H	Tux	AW	ANL	GRA	DIN
Incoming frags.	581,887k	2,306,487k	1,037,222k	38,044k	466,344k	146,604k	91,824k
Processed frags.	578,345k	2,292,357k	512,618k	38,044k	466,344k	146,604k	91,824k
Passed frags.	461,045k	1,822,735k	473,738k	35,037k	281,684k	137,109k	73,268k
Frags. written to the depth buffer	166,624k	666,633k	462,520k	35,037k	281,684k	137,109k	73,268k

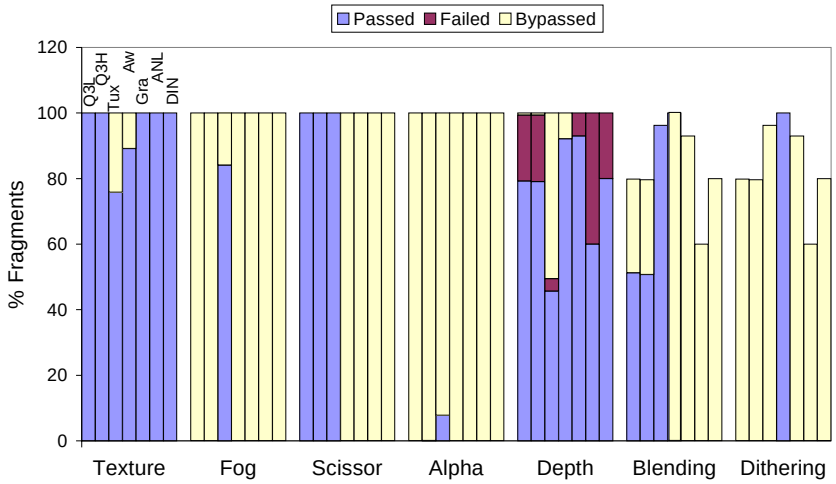


Figure 3: Rasterization pipeline units usage

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.

5. CONCLUSIONS AND FUTURE WORK

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 paper we have described a set of relevant applications for embedded 3D graphics accelerators performance evaluation. Also one of the objectives of this paper 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.

The OpenGL applications that were used to create the benchmarks and the GLtrace tracer are accessible via the first author's website (`http://ce.et.tudelft.nl/~tkg/`). The benchmarks (i.e. the traces) cannot be made public currently, because they are of no use without the trace player and the trace player is confidential at the moment. However, the Quake III (demo version) and the AWadvs-04 components do not require the use of the trace player in order to generate repeatable workloads. We hope to be able to make the benchmark suite publicly available in the future.

As future work, we intend to extend the number of components for this benchmark suite, and we also intend to extend the statistics to include results from embedded graphics architectures that are using a tile-based rendering mechanism.

6. REFERENCES

[1] T. Akenine-Möller and J. Ström, "Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones", ACM Trans. on Graph.,vol 22, nr 3, 2003, pp. 801-808.

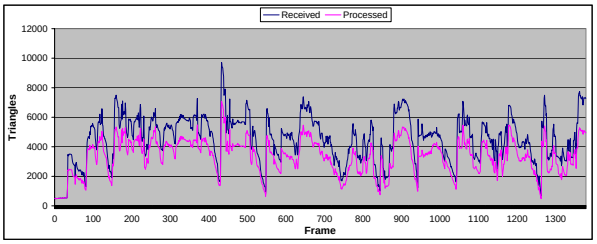
[2] I. Antochi, B.H.H. Juurlink, A. G. M. Cilio, and P. Liuha. "Trading Efficiency for Energy in a Texture Cache Architecture", *Proc. Euromicro Conf. on Massively-Parallel Computing Systems (MPCS'02)*, 2002, Ischia, Italy, pp. 189-196.

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

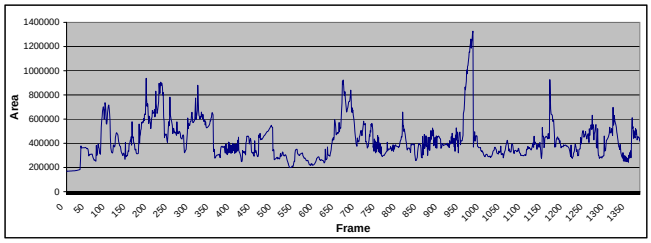
[4] Futuremark Corporation, "3DMark03", Available at http://www.futuremark.com/products/3dmark03/

[5] D. Crisu, S.D. Cotofana, S. Vassiliadis, and P. Liuha, "GRAAL — A Development Framework for Embedded Graphics Accelerators", *Proc. Design, Automation and Test in Europe (DATE 04)*, Paris, France, February 2004.

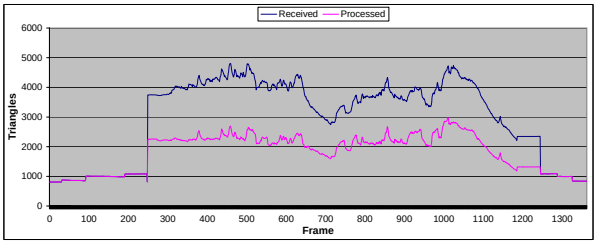
[6] J.C. Dunwoody and M.A. Linton. "Tracing Interactive 3D Graphics Programs", *Proc. ACM Symp. on Interactive 3D Graphics*, 1990.



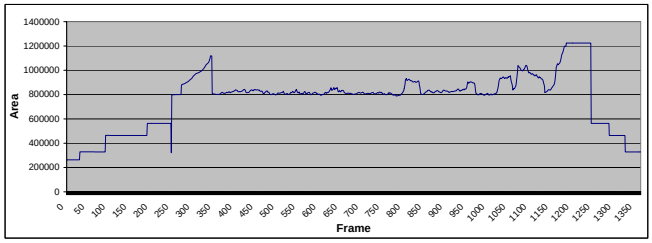
(a) Q3L Triangles



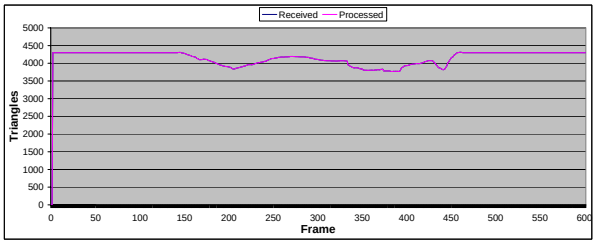
(b) Q3L Area



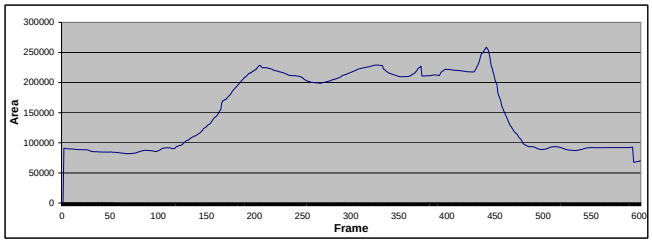
(c) Tux Triangles



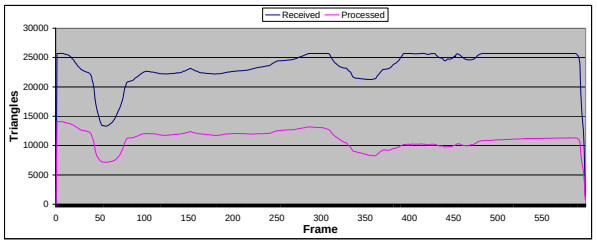
(d) Tux Area



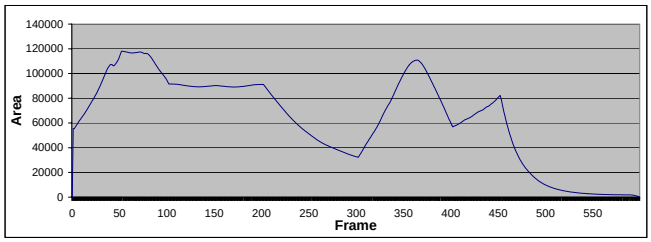
(e) DIN Triangles



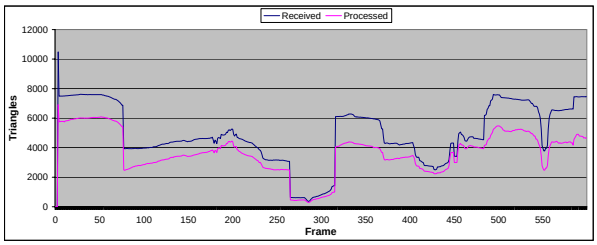
(f) DIN Area



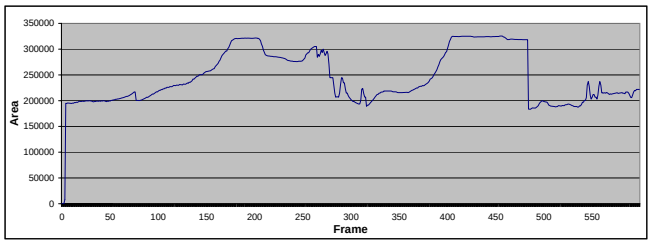
(g) AW Triangles



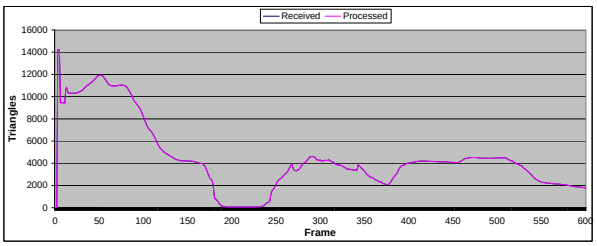
(h) AW Area



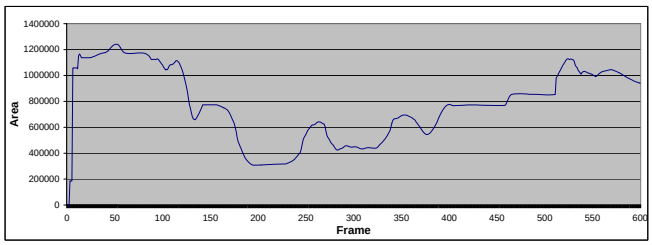
(i) GRA Triangles



(j) GRA Area



(k) ANL Triangles



(l) ANL Area

Figure 4: Triangle and area statistics for the GraalBench components

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

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

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

[10] ARM Ltd., “ARM 3D Graphics Solutions”, Available at http://www.arm.com/miscPDFs/1643.pdf

[11] 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.

[12] Systems in Motion, “VRMLView”, Available at http://www.sim.no

[13] 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.

[14] The Mesa Project, “The Mesa 3D Graphics Library”, Available at http://www.mesa3d.org

[15] Hawk Software, “GLTrace Programming Utility”, Available at http://www.hawksoft.com/gltrace/

[16] 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.

[17] SourceForge, “spyGLass: an OpenGL Call Tracer and Debugging Tool”, Available at http://spyglass.sourceforge.net/

[18] SPEC, “SPECviewperf 6.1.2”, Available at http://www.specbench.org/gpc/opc.static/opcview.htm

[19] Sunspire Studios, “Tux Racer”, Available at http://tuxracer.sourceforge.net/

[20] 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

[21] Stanford University, “GLSim & GLTrace”, Available at http://graphics.stanford.edu/courses/cs448a-01-fall/glsim.html

[22] Yonsei University, 3D Graphics Accelerator Group, http://msl.yonsei.ac.kr/3d/