

Workload Characterization of 3D Games

Jordi Roca, Victor Moya, Carlos González, Chema Solís, Agustín Fernández,
Department of Computer Architecture, Universitat Politècnica de Catalunya,
jroca@ac.upc.edu

and Roger Espasa, *Intel, DEG, Barcelona*

Abstract—The rapid pace of change in 3D game technology makes workload characterization necessary for every game generation. Comparing to CPU characterization, far less quantitative information about games is available. This paper focuses on analyzing a set of modern 3D games at the API call level and at the microarchitectural level using the Attila simulator. In addition to common geometry metrics and, in order to understand tradeoffs in modern GPUs, the microarchitectural level metrics allow us to analyze performance key characteristics such as the balance between texture and ALU instructions in fragment programs, dynamic anisotropic ratios, vertex, z-stencil, color and texture cache performance.

I. INTRODUCTION

GPU design and 3D game technology evolve side by side in leaps and bounds. Game developers deploy computationally demanding 3D effects that use complex shader programs with multiple texture accesses that are highly tuned to extract the maximum performance on the existing and soon-to-be-released GPUs. On the other hand, GPU designers carefully tailor and evolve their designs to cater for the needs of the next generation games expected to be available when the GPU launches. To this end, they put on the market high-end and middle-end graphics cards with substantial vertex/pixel processing enhancements w.r.t. previous generations, expecting to achieve high frame rates in newly released games.

As with any other microprocessor design, carefully understanding, characterizing and modeling the workloads at which a given GPU is aimed, is key to predict and meet its performance targets. There is extensive literature characterizing multiple CPU workloads [24][25][26]. Compared to the CPU world, though, there is a lack of published data on 3D workloads in general, and interactive games in particular. The reasons are manifold: GPUs still have a wide variety of fixed functions that are difficult to characterize and model (texture sampling, for example), GPUs are evolving very fast, with continuous changes in their programming model (from fixed geometry and fixed texture combiners to full-fledged vertex and fragment programs), the games are also rapidly evolving to exploit these new programming models, and new functions are constantly added to the rendering pipeline as higher silicon densities makes the on-die integration of these functions cost-effective (geometry shaders and tessellation, for example). All these reasons combine to produce an accelerated rate of change in the workloads and even relatively recent studies rapidly obsolete.

For example, [1][2] characterize the span processing workload in the rasterization stage. However, today all GPUs use the linear edge function rasterization algorithm [6], making span processing no longer relevant. As another example, [1] studies the geometry BW requirements per frame, which has nowadays substantially decreased thanks to the use of indexed modes and storing vertexes in local GPU memory.

The goal of this work is to analyze and characterize a set of recent OpenGL (OGL) and Direct3D (D3D) interactive games at the API level as well as at the microarchitectural level. In general, 3D games can be analyzed at different levels: at application CPU instruction level, disc I/O requests level, or PCI-AGP bus transactions level as [3] does. In this paper, the approach taken is to characterize the graphics API level (OGL or D3D) and complement the information with microarchitectural measurements taken from the back-end of the rendering pipeline using the ATTILA simulator [9] configured to closely match an ATI R520 [27]. At the API level we analyze common metrics such as indexes, primitives, batches per frame and the average size of these batches. We also take a look at the number of state changes per frame as an indirect measure of API overhead/complexity.

To understand the trade-offs in modern shader processors, we take a look at the number of instructions in shader programs and in particular at the ratio of texture accesses versus computation instructions in the fragment shader program. We complete texture request filtering information using microarchitecture level metrics to know the actual average bilinear samples per texture request. With all this information we characterize the texture requests overhead in the shader program execution. To determine texture caches performance, we also explore the BW required from GDDR memory modules to access texture data.

The remainder of the paper is organized as follows. Section II describes the game workload selection and the framework used to collect statistics. Section III presents the data for each major “stage” of the rendering pipeline: geometry, rasterization, fragment, and raster operations. Section IV discusses related work and Section V summarizes the major findings of this paper.

II. STATISTICS ENVIRONMENT

A. Workload selection

We used three main criteria to select our benchmarks: 1) Benchmarks should cover the two major Graphics APIs in the marketplace, Direct3D and OpenGL, 2) Benchmarks should

TABLE I
GAME WORKLOAD DESCRIPTION

Game/Timedemo	# Frames	Duration at 30 fps	Texture quality	Aniso level	Shaders	Graphics API	Engine	Release date
UT2004/Primeval	1992	1' 06"	High/Anisotropic	16X	NO	OpenGL	Unreal 2.5	March 2004
Doom3/trdemo1	3464	1' 55"	High/Anisotropic	16X	YES	OpenGL	Doom3	August 2004
Doom3/trdemo2	3990	2' 13"	High/Anisotropic	16X	YES	OpenGL	Doom3	August 2004
Quake4/demo4	2976	1' 39"	High/Anisotropic	16X	YES	OpenGL	Doom3	October 2005
Quake4/guru5	3081	1' 43"	High/Anisotropic	16X	YES	OpenGL	Doom3	October 2005
Riddick/MainFrame	1629	0' 54"	High/Trilinear	-	YES	OpenGL	Starbreeze	December 2004
Riddick/PrisonArea	2310	1' 17"	High/Trilinear	-	YES	OpenGL	Starbreeze	December 2004
FEAR/built-in demo	576	0' 19"	High/Anisotropic	16X	YES	Direct3D	Monolith	October 2005
FEAR/interval2	2102	1' 10"	High/Anisotropic	16X	YES	Direct3D	Monolith	October 2005
Half Life 2 LC/built-in	1805	1' 00"	High/Anisotropic	16X	YES	Direct3D	Valve Source	October 2005
Oblivion/Anvil Castle	2620	1' 27"	High/Trilinear	-	YES	Direct3D	Gamebryo	March 2006
Splinter Cell 3/first level	2970	1' 39"	High/Anisotropic	16X	YES	Direct3D	Unreal 2.5++	March 2005

cover the most successful “middle ware” game engines, including those from Epic, Valve and IDSoftware and, 3) Benchmarks should predominantly use the newer vertex and fragment programs rather than the old fixed function API (with the exception of UT2004, kept for correlation purposes). The resulting selection is shown in TABLE I.

For each game we traced either a “timedemo” available from a benchmarking web page [13], or we recorded our own game sequence. Timedemos are recorded sequences of game playing that can be reproduced on different machines at different speeds without skipping any frame, giving a different completion time and thus computing an average frame rate for the graphics board performance, as explained in [16]. Timedemos are widely used to make performance comparisons across different 3D graphics boards and system configurations. The selected timedemos cover a variety of scenarios, ranging from inner room sequences, such as in Quake4 and Doom3, to open countryside scenarios with lots of geometry, as in the Oblivion timedemo.

For all sequences we used the same 1024x768 resolution and configured each game to its highest quality setting. The fixed resolution was chosen to ease comparisons across benchmarks. Although higher resolutions are common in benchmarking tests, we have used the most common resolution in current 17” TFT displays.

It should be noted that averages and statistics presented throughout the paper are computed using the full number of frames indicated in TABLE I. However, and for the sake of clarity, some plots have been drawn using only the first 2000 frames of each timedemo.

B. Tracing setup

To collect API-level statistics we use a home-grown OGL tracer (GLInterceptor, part of the ATTILA framework [9]) and the PIX tool from the D3D SDK [23]. We have developed a

“player” that can reproduce the traces captured by PIX and collect the same set of statistics that GLinterceptor collects. The benchmarks described in the previous section were run on an ATI Radeon 9800 and the corresponding traces collected and statistics gathered. For the OGL benchmarks, the traces were then used to drive the ATTILA simulator to collect low-level micro-architecture characteristics of the games that can not be derived from the API level statistics. Unfortunately, ATTILA is limited to running OGL programs only. Hence, for the microarchitectural measurements, only the OGL benchmarks have been used. The simulation parameters in ATTILA have been configured to match a reference high-end GPU architecture, the ATI R520, as shown in TABLE II .

TABLE II
ATTILA CONFIGURATION

	R520	ATTILA
Vertex/Fragment Shaders	8/16	16 (unified)
Triangle Setup	2 triangles/cycle	2 triangles/cycle
Texture Rate	16 bilinears/cycle	16 bilinears/cycle
ZStencil / Color Rates	16 / 16 fragments/cycle	16 / 16 fragments/cycle
Memory BW	> 64 bytes/cycle	64 bytes/cycle

The above configuration parameters do not affect any results shown in this work because either they are based on static API level measurements not dependent on the GPU model or results are exclusively dependent on the polygon rejection, rasterization or HZ algorithms implemented rather than on the GPU model parameters. But, as we will see in Section III.E, the concrete caches configuration directly affects the memory BW consumed by the selected workloads.

III. GAME ANALYSIS

A. CPU load analysis

Starting at the graphics API level, the number of batches per frame (the different vertex input streams which are processed down through the rendering pipeline forming the entire frame) is a good first order approximation of the rendered scene complexity. Figure 1 shows the number of batches per frame for the first 2000 frames of our benchmark set (for clarity only a timedemo per benchmark has been used). The data show that the interactive nature of games, in contrast to data presented in [2], makes the number of batches per frame highly variable over time.

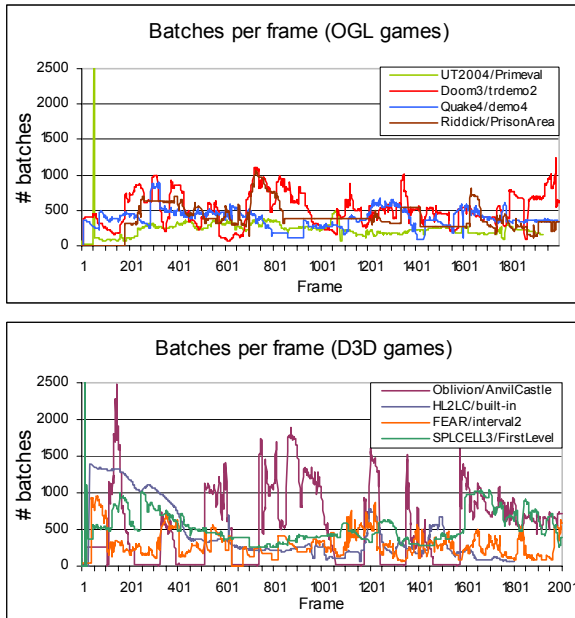


Figure 1 Total batches per frame

Each batch processed involves a certain amount of computation on the CPU side. Modern games use “indexed mode” for sending batch geometry to the GPU. In indexed mode, the vertex geometry data is sent at startup time to the GPU and stored in its local memory. Then, every time the geometry needs to be rendered the CPU sends a list of indices into the vertex data. TABLE III presents the average number of indexes per batch as well as the average number of indices in a frame for each benchmark. Note that the size of each index is constant across the entire execution and is dependent on the middleware used in the game. Figure 2 shows, for a subset of our benchmarks, the megabytes/frame transferred from the CPU to the GPU to carry the index data.

The data in TABLE III and Figure 2 show that the total BW requirements between CPU and GPU are relatively low: even assuming a rendering speed of 100 frames-per-second, index traffic amounts to much less than a gigabyte per second. This is important to later understand the choice of primitive used by most games (see section III.B).

A second measure of “CPU Overhead” incurred in a game is the number of API calls made per frame. Fig. 3. shows, in logarithmic scale, the average number of state calls per frame

TABLE III

AVERAGE INDICES PER BATCH AND FRAME AND TOTAL BW

Game/Timedemo	avg. indexes per batch	avg. indexes per frame	bytes per index	BW @100fps
UT2004/Primeval	1110	249285	2	50 MB/s
Doom3/trdemo1	275	196416	4	79 MB/s
Doom3/trdemo2	304	136548	4	55 MB/s
Quake4/demo4	405	172330	4	69 MB/s
Quake4/guru5	166	135051	4	54 MB/s
Riddick/MainFrame	356	214965	2	43 MB/s
Riddick/PrisonArea	658	239425	2	48 MB/s
FEAR/built-in demo	641	331374	2	66 MB/s
FEAR/interval2	1085	307202	2	61 MB/s
Half Life 2 LC/built-in	736	328919	2	66 MB/s
Oblivion/Anvil Castle	998	711196	2	142 MB/s
Splinter Cell 3/first level	308	177300	2	35 MB/s

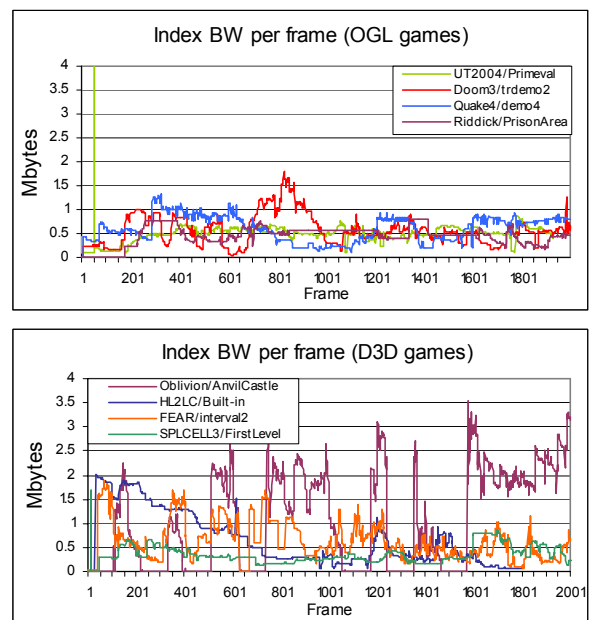


Figure 2 Index BW per frame

for the same subset of benchmarks. As expected, the first frames in a game contain a much larger number of API calls required to “set up” geometry and texture data. Eventually, though, the number of calls becomes relatively flat. It is interesting to note the peaks in FEAR and Oblivion. These peaks correspond to inter-scene transitions where new data (both geometry and texture) are loaded.

B. Geometry Pipeline

This section focuses on the behaviour of our workload when going through the geometry stages of the rendering pipeline.

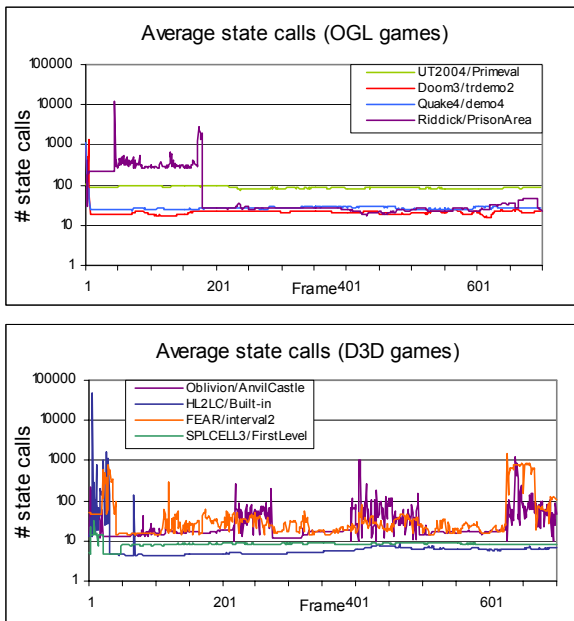


Figure 3 Average state calls between batches

The geometry stage projects primitives specified in world coordinates onto 2D viewport coordinates. To this end, modern GPUs use a vertex program to transform vertex coordinates and then form primitives (typically triangles) using the transformed vertices.

We start measuring the amount of computation required to transform vertices. TABLE IV presents separately for OGL and D3D benchmarks the average number of vertex program instructions executed for each vertex. In general, results have shown little variability in average vertex shader lengths across frames. Only the Oblivion timedemo experiences a significant increase in average vertex program length in the second half of the execution and, hence, we present an average for each of its two distinctive regions.

TABLE IV
AVERAGE VERTEX SHADER INSTRUCTIONS

OGL games	Average vertex shader instructions	D3D games	Average vertex shader instructions
UT2004/Primeval ¹	23.46	FEAR/built-in demo	18.19
Doom3/trdemo1	20.31	FEAR/interval2	21.02
Doom3/trdemo2	19.35	Half Life 2 LC/built-in	27.04
Quake4/demo4	27.92	Oblivion/AnvilCastle	Reg1: 18.88
Quake4/guru5	24.42		Reg2: 37.72
Riddick/MainFrame	16.70	Splinter Cell 3/ FirstLevel	28.36
Riddick/PrisonArea	20.96		

1. Although UT2004 does not use vertex programs, because of the ATTLA shader architecture, its low-level driver translates transparently the fixed function functionality into shader programs, and thus the corresponding statistic could be obtained.

After transforming the vertices, the Primitive Assembly

stage groups transformed vertices into primitives according to the primitive type. TABLE V characterizes the most common primitives used in our benchmarks. While both OGL and D3D provide a rich variety of primitives, including points, lines, polygons, quads and quad strips, our benchmarks only use triangle lists (TL), triangle strips (TS) and triangle fans (TF). Surprisingly, triangle strips and triangle fans have a very low usage across all benchmarks, except Oblivion and, to a lesser extent, Splinter Cell. Strips and fans are specifically designed to improve geometry throughput by “sharing” vertices across adjacent primitives, as shown in Figure 4, so this result was unexpected.

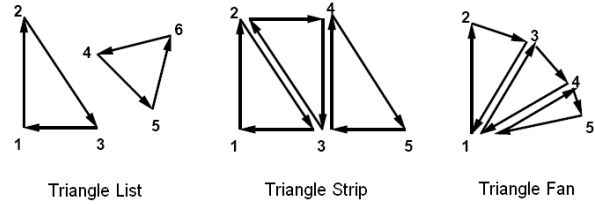


Figure 4 Triangle primitives

The explanation resides in the existence of post-transform vertex caches in current GPUs. When using indexed mode, this vertex cache allows reusing already transformed vertices, provided that two references to a vertex are close in time. Thus, if repeated vertices in a triangle list are run through a vertex cache close enough, the triangle list will behave, from a vertex shading point of view, like a triangle strip (where vertices are shared by construction). Figure 5 illustrates this point showing the vertex cache hit rate in the ATTLA simulator for the three OGL benchmarks. As the data show, the performance of the vertex cache is close to the theoretical 66% hit rate for the adjacent triangles case. The observed lower ratios are because of the existence of more separated triangles in the scene and the higher ratios are explained because certain face orderings improve even more vertex cache locality, as the face ordering resulting from algorithms explained in [15].

TABLE V
PRIMITIVE UTILIZATION

Game/timedemo	TL	TS	TF	Avg. primitives per frame
UT2004/Primeval	99.9%		0.1%	83095
Doom3/trdemo1	100%			65472
Doom3/trdemo2	100%			45516
Quake4/demo4	100%			57443
Quake4/guru5	100%			45017
Riddick/MainFrame	100%			71655
Riddick/PrisonArea	100%			79808
FEAR/built-in demo	100%			110458
FEAR/interval2	96.7%	3.3%		102402

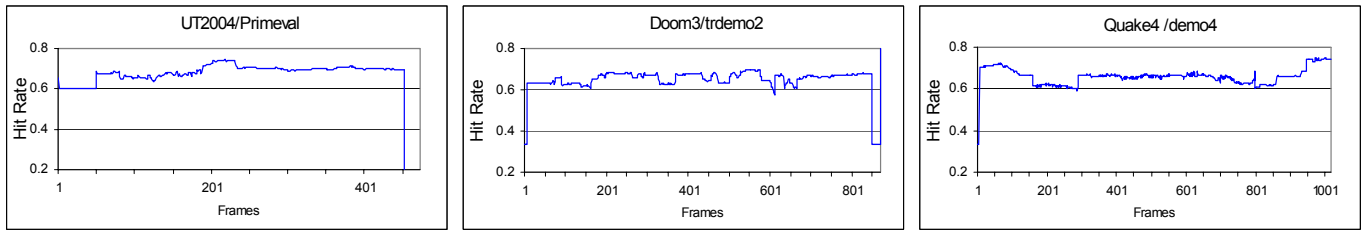


Figure 5 Post-transform vertex cache hit rate

TABLE V
PRIMITIVE UTILIZATION

Game/timedemo	TL	TS	TF	Avg. primitives per frame
Half Life 2 LC/built-in	100%			109640
Oblivion/Anvil Castle	46.3%	53.7%		551694
Splinter Cell 3/first level	69.1%	26.7%	4.2%	107494

Given the presence of vertex caches in all current GPUs, the only advantage of using triangle strips over triangle lists would be the reduction in number of indexes sent from CPU to GPU. However, as seen before, the index BW requirements of our benchmarks does not even come close to 1GB/s, which is still far away from the today available system bus BWs, as shown in TABLE VI. Consequently, it is not surprising to see most game developers choose triangle lists for their games, due to lists being easier to create, manage and import from other modeling tools. Only in the Oblivion sequence there is a substantial amount of open terrain that is better and easier modeled using triangle strips.

TABLE VI
CURRENT SYSTEM BUS BWs

Bus	Width	Bus Speed	Bus BW
AGP 4X	32 bits	66x4 MHz	1.056 GB/s
AGP 8X	32 bits	66x8 MHz	2.112 GB/s
PCI Express x4 lanes	1 bit ¹	2.5 Gbaud x 4	1 GB/s
PCI Express x8 lanes	1 bit	2.5 Gbaud x 8	2 GB/s
PCI Express x16 lanes	1 bit	2.5 Gbaud x 16	4 GB/s

1. PCI Express uses serial links with a 10bits/byte encoding

After primitive assembly, the clipper stage tests the assembled triangles against the view frustum volume and front face or back face triangle culling is performed, to avoid rasterization of non-visible triangle faces. Figure 6 shows, for the OGL simulated timedemos, from top to bottom, the number of indices per frame (“indices”), then number of assembled triangles per frame (“assembled”, exactly 1/3 of the indices, given that these three benchmarks only use triangle lists) and how many of these triangles actually pass through the clipping and culling stages to be further traversed by the

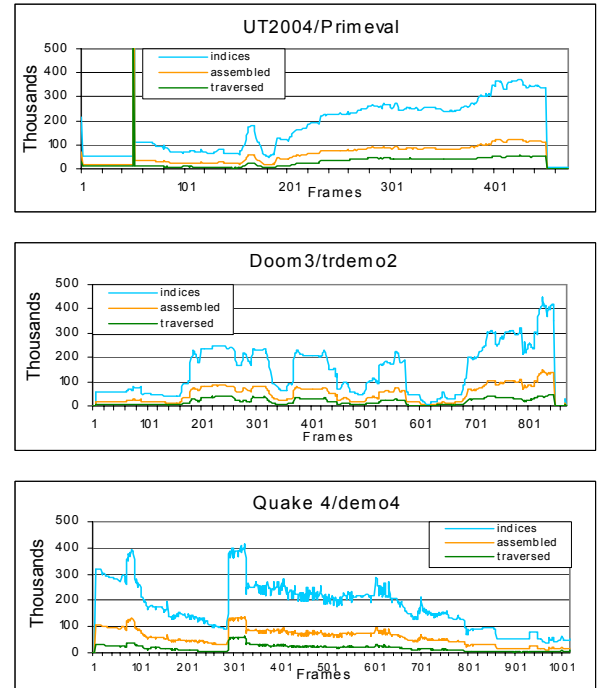


Figure 6 Indices, triangles assembled and triangles traversed

rasterization stage (“traversed”).

To complement these data, TABLE VII further breaks down the primitives assembled indicating the percentage discarded due to clipping, the percentage discarded due to culling and the remainder percentage of traversed.

TABLE VII
PERCENTAGE OF CLIPPED, CULLED AND TRAVERSED TRIANGLES

Game/timedemo	% clipped	% culled	% traversed
UT2004/Primeval	30%	21%	49%
Doom3/trdemo2	37%	28%	35%
Quake4/demo4	51%	21%	28%

C. Rasterization, HZ, Z and Stencil, Alpha Test, Color Mask

The rendering algorithm in modern GPUs is based on the process of converting triangles projected onto a 2D viewport into small surface patches (that could be approximated as a rectangle) named fragments. A fragment corresponds to either a part or the whole area of a pixel in the destination framebuffer. This process is called rasterization and happens after the geometry stage previously discussed. The fragments

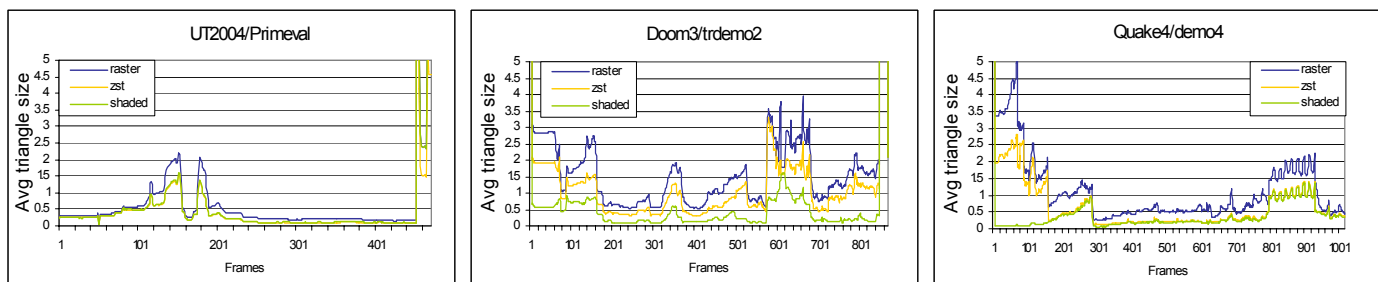


Figure 7 Average triangle size per frame at different stages (in thousands of fragments)

generated from the input triangles pass through a number of tests (scissor, depth test, stencil test and alpha test) and are further processed in a second shading stage (fragment shading) before updating the color buffer in the framebuffer to produce the final rendered image.

Multiple rasterization algorithms exist: Scan conversion based or edge equation based, single fragment or tile based, etc. Modern GPUs implement a tiled and edge equation based algorithm as described in [6]. In particular, the ATTILA simulator implements a recursive rasterization algorithm based on [17] that works at two different tile levels: an upper level with a 16x16 footprint and at a lower level generating each cycle 8x8 fragment tiles. These tiles are then processed in the next stages and partitioned into 2x2 fragment tiles, called quads. Quads are the working unit of the subsequent GPU pipeline stages.

The generated quads pass through a number of test stages that mark fragments as culled and may remove the whole quad if all its fragments are marked as culled: a) scissor test, culls the fragments against a rectangular framebuffer region; b) z test compares the fragment depth against the depth value stored in the framebuffer depth (or z) buffer; c) stencil test culls fragments based on a per pixel mask stored in the framebuffer stencil buffer; d) alpha test culls transparent fragments. The alpha test always happens after fragment shading while the scissor test always happens before shading. The z and stencil test are performed in parallel in the same stage and may happen before shading (early z and stencil test) or after shading (in case either alpha testing is enabled or the fragment program uses the kill instruction or modifies the depth value).

Modern GPUs implement a Hierarchical Z buffer [18] stored in on-die memory to reduce the amount of memory BW consumed by z testing. When Hierarchical Z (HZ) is enabled (it may be disabled for some z and stencil modes) the z test is performed in two different phases: In the HZ stage accessing only on-die memory and in the z and stencil test stage accessing GPU memory.

The first statistics that have been gathered from this stage using the ATTILA GPU simulator is the number of fragments per triangle that are generated by the rasterization algorithm and processed in the next stages. TABLE VIII shows the average triangle size in fragments in each of the discussed processing stages of the fragment pipeline: rasterization, z and stencil test, shading and color buffer update and blending. Fig. 7. shows the average triangle size per frame for the three

simulated benchmarks. TABLE IX shows the percentage of quads that are removed in the HZ, z and stencil and alpha test (implemented using a fragment program with the shader kill instruction in ATTILA) and the percentage of quads with the color write mask set to false and those that finally update the color buffer. Blending is always active in the color stage for the three simulated benchmarks. Doom3 and Quake4 use a stencil based technique to create dynamic shadow effects that generates a large amount of fragments with the only purpose of updating the stencil buffer, but not the color buffer, so for the two benchmarks a large percentage of quads reach the color update stage but are immediately removed because of the color write mask set to false.

TABLE VIII

AVERAGE TRIANGLE SIZE (IN FRAGMENTS)

Game/timedemo	Raster	Z&Stencil	Shading	Blending
UT2004/Primeval	652	417	510	411
Doom3/trdemo2	2117	1651	1027	1024
Quake4/demo4	1232	749	411	406

TABLE IX

PERCENTAGE OF REMOVED OR PROCESSED QUADS AT EACH STAGE

Game/timedemo	HZ	Z&Stencil	Alpha	Color Mask	Blending
UT2004/Primeval	37.50%	2.42%	4.15%	0%	55.93%
Doom3/trdemo2	33.95%	13.81%	0.03%	34.48%	17.73%
Quake4/demo4	41.81%	20.57%	0.32%	19.00%	18.30%

The simulated data demonstrate that the HZ stage is removing a large percentage (90% in UT2004, 60% in Doom3 and 50% in Quake4) of the fragments that could be removed because of z test fail. Since the HZ buffer is on-die the corresponding GDDR BW saved is quite significant. However further improvements could be achieved with a better HZ implementation (for example combining stencil into the HZ buffer or a HZ storing maximum and minimum values) or using deferred rendering techniques [19].

The fragment pipeline works on quads. Quads improve memory locality, help in reducing control logic and are a requirement to be able to compute the level of detail (lod) in the texture unit stage. Unfortunately, some quads may have

one or more fragments marked as culled either because the quad corresponds with a triangle border or because they were removed in one of the fragment test stages. TABLE X shows the percentage of complete quads at two different stages of the fragment pipeline, thus showing the efficiency of the quad work unit. The simulated data show higher quad efficiency than results presented in [1], achieving in their case only between 40% and 60%. Their experimented efficiencies are lower because the average triangle size is substantially smaller than in our workloads (their experiments resolutions are smaller and more detailed 3D models are used), and this degrades the percentage of complete quads.

TABLE X
QUAD EFFICIENCY (% COMPLETE QUADS)

Game/timedemo	Raster	Z&Stencil
UT2004/Primeval	91.5%	93.0%
Doom3/trdemo2	93.1%	95.0%
Quake4/demo4	92.0%	92.7%

The algorithm used to generate the final rendered images may require generating and processing multiple fragments for the same framebuffer pixel. The number of fragments that are processed for a given pixel in the final rendered frame is usually called *overdraw*. TABLE XI shows the average overdraw in terms of the number of framebuffer pixels at the benchmarked resolution compared with the total number of fragments generated or processed at each of the discussed stages of the GPU pipeline. The table shows the numbers for rasterized fragments, fragments processed in the z and stencil stage (including the case that the tests are disabled and the fragment bypasses the stage), fragments shaded and fragments that are finally blended into the color buffer.

TABLE XI
AVERAGE OVERDRAW PER PIXEL AND STAGE

Game/timedemo	Raster	Z&Stencil	Shading	Blending
UT2004/Primeval	8.94	5.22	5.52	5.00
Doom3/trdemo2	24.58	16.22	4.38	4.36
Quake4/demo4	24.39	14.12	4.55	4.46

The overdraw at the rasterization and z and stencil stages for the Doom3 and Quake4 benchmarks is significant higher and the reason is, as already commented, the large number of fragments generated to implement the stencil shadows algorithm that both games use. Usually there are two components in the overdraw: overdraw due to hidden surface removal and overdraw due to multipass render algorithms. The first component is an inefficiency of the z buffer algorithm that can not prevent useless updates to the framebuffer when the rendered triangles are not processed in depth order. The second component is because of limitations in the GPU pipeline that force the rendering algorithm to send multiple

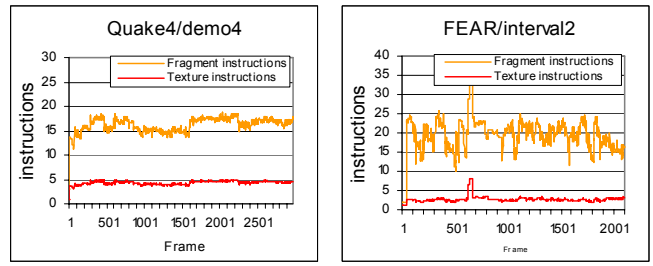


Figure 8 Average fragment program instructions

times the same geometry to generate a visual effect (for example shadows from different light sources when using stencil shadows). For Doom3 and Quake4, though, the first component is non-existent in the shading and blending stages as both games use a first pass to set the depth buffer, and all the shading passes are performed with z test comparison set to equal (kind of a software based deferred rendering).

D.Fragment Shading and Texturing

The main fragment stage in current programmable GPUs is actually the fragment shading stage. TABLE XII shows the average size in instructions (total), texture instructions and ratio between arithmetic and texture instructions used in fragment programs. This statistic has been generated at the API level (OGL or D3D) for all benchmarks described in the previous sections. Figure 8 shows the average fragment program size and texture instructions on a per frame basis for the Quake4 and FEAR benchmarks.

TABLE XII

AVG. INSTRUCTIONS, TEXTURE INSTRUCTIONS AND ALU TO TEXTURE RATIO

Game/timedemo	Instructions	Texture Instructions	ALU to Texture Ratio
UT2004/Primeval ¹	4.63	1.54	2.01
Doom3/trdemo1	12.85	3.98	2.23
Doom3/trdemo2	12.95	3.98	2.25
Quake4/demo4	16.29	4.33	2.76
Quake4/guru5	17.16	4.54	2.78
Riddick/Main Frame	14.64	1.94	6.55
Riddick/Prison Area	13.63	1.83	6.45
FEAR/built-in	21.30	2.79	6.63
FEAR/interval 2	19.31	2.72	6.10
Half Life 2 LC/built-in	19.94	3.88	4.14
Oblivion/Anvil Castle	15.48	1.36	10.38
Splinter Cell 3/first level	4.62	2.13	1.17

1. The same way as in TABLE IV has been used to obtain this data.

TABLE XII shows that the ratio between arithmetic instructions and texture instructions is at least 2 or greater for all but one benchmark and this ratio is actually increasing in

more recent games (Oblivion, FEAR and Quake4). This ratio is important because some GPU architectures already implement a disbalancing between shader ALUs and texture throughput. The ATI's Xenos (Xbox360) [20], RV530 [21] and R580 [22] GPUs have three times the arithmetic processing power compared to the texture throughput. To make an efficient use of the additional shading computing power the ratio between texture processing and shader ALU processing must be greater than 1 and for optimal performance larger than 3.

These API statistics are not enough though to determine the real ratio between shader ALU processing and texture processing. There is a dynamic component in texture processing related with the implementation of the bilinear, trilinear and anisotropy filtering algorithms. The texture processing throughput of modern GPUs is fixed to 1 bilinear sample per cycle and fragment pipeline and better than bilinear filter algorithms take additional throughput cycles to complete (1 more for trilinear, up to 32 more with a 16 sample anisotropy filtering algorithm, see [28]). TABLE XIII shows firstly the average number of bilinear samples required per texture request for the three simulated benchmarks. Then, when combining these data with the corresponding TABLE XII ratios, the ratio between shader ALU processing and texture processing (bilinear requests) is below 1 and therefore the disbalanced architectures would not be able to efficiently use their increased shader processing power.

TABLE XIII

AVERAGE BILINEAR SAMPLES AND ALU TO BILINEAR RATIO

Game/timedemo	Bilinear samples per request	ALU instructions/ bilinear requests
UT2004/Primeval	5.15	0.39
Doom3/trdemo2	4.37	0.52
Quake4/demo4	4.67	0.59

E.Memory

Memory subsystem is a key component of a modern GPU as the rendering algorithm requires multiple accesses in each of the different stages. The main stages that perform memory accesses are the vertex load pipeline that retrieves indices and vertex data from buffers in GPU or system memory, the z and stencil test that access the z and stencil buffer in GPU memory, the texture unit that samples texture data from GPU memory and the color stage that updates and blends the final fragment color with the color buffer. Other consumers are the DAC that outputs the rendered frame to a display device and the Command Processor that parses rendering commands and handles the transfers of data from system to GPU memory.

Due to the cost of high BW, GPU vendors usually put caches at different stages of the pipeline (some of them using compression techniques) to reduce the BW requirements. TABLE XIV shows the configuration and hit rates for the z and stencil, texture and color caches used in the ATTLA architecture. The z and stencil cache implements a fast clear

and z compression algorithm to save BW. The texture cache implements two levels: level 0 stores uncompressed data and level 1 stores compressed data. The color cache implements fast clear and a very simple compression algorithm that only works for blocks pixels with the same color.

TABLE XIV

CACHE CONFIGURATION AND HIT RATE

Cache	Size	Way/Line Size	Doom3 /tr2	Quake4 /d4	UT2004
Z&Stencil	16 KB	64w x 256B	91.0%	93.4%	93.9%
Texture L0	4 KB	64w x 64B	99.2%	99.3%	97.7%
Texture L1	16 KB	16w x 16s x 64B			
Color	16 KB	64w x 256B	93.2%	93.2%	93.7%

Given the presence of caches and compression, TABLE XV shows the average memory BW consumed per frame for the three benchmarks and the percentage of the consumed BW used for read and write transferences. The amount of data read per frame is approximately double of the written data because all stages read from memory but the vertex load, texturing (which is at least a quarter of all the consumed BW) and DAC stages never write to memory.

TABLE XV

AVERAGE MEMORY USAGE PROFILE

Game/timedemo	MB/frame	%Read	%Write	BW@100fps
UT2004/Primeval	81	73%	27%	8 GB/s
Doom3/trdemo2	108	63%	37%	11 GB/s
Quake4/demo4	101	62%	38%	10 GB/s

TABLE XVI shows the percentage of memory BW consumed by each of the described stages for the three simulated benchmarks. Texturing is the main component of the BW requirement and is only surpassed by z and stencil in Doom3 and Quake4 because of the stencil shadows algorithm they implement.

TABLE XVI

MEMORY TRAFFIC DISTRIBUTION PER GPU STAGE

Game/timedemo	Vertex	Z&Stencil	Texture	Color	DAC	CP
UT2004/Primeval	3.9%	15.2%	41.7%	35.2%	3.5%	0.5%
Doom3/trdemo2	2.5%	53.5%	26.1%	14.8%	2.1%	1.1%
Quake4/demo4	4.2%	51.4%	23.0%	17.4%	2.7%	1.3%

Finally TABLE XVII shows the average number of bytes read or written from GPU memory per shaded vertex and per fragment at three different stages: z and stencil test, shading (reading texture data) and color (blending). If no caches or BW reduction techniques were used the rendering algorithm

would require 4 bytes for reading z and stencil data, 4 bytes for writing z and stencil data, 16 bytes for each bilinear sample required for each texture load in the shader and 4 bytes for reading the current color and 4 bytes for writing the new color in the color buffer. The data required for vertices would be one index (one to four bytes) and depending on the number of vertex attributes (up to 16 bytes each) up to 256 bytes.

TABLE XVII
BYTES PER VERTEX AND FRAGMENT

Game/timedemo	Vertex	Z&Stencil	Shaded	Color
UT2004/Primeval	50.18	3.14	7.71	7.40
Doom3/trdemo2	50.88	4.61	8.31	4.60
Quake4/demo4	67.60	4.48	6.68	5.11

Vertexes require quite more data than fragments because they are the first stage of the rendering algorithm and all the geometry based varying data is obtained from the vertex attributes. However as the number of processed vertices is a very small fraction of the processed fragments the impact in the total BW requirements per frame is quite reduced. The z fast clear and compression algorithm is reducing by a half the BW requirements of the z and stencil stage. The combination of using compressed textures (DXT1, DXT3 and DXT5) for most of the texture data in the three benchmarks with the texture cache reduces almost to a tenth the required BW for texture data. The fast color and color compression algorithm fails to work in UT2004 and only significantly reduces the BW consumed in Doom3 and Quake4 because large portions of the final frame remain in shadow.

IV. RELATED WORK

Mitra and Chiueh [1] is the main work about 3D Workload Characterization. It reports important rendering dynamics across image frames, characterizing geometry BW requirements, memory subsystem performance from the point of view of the pixel processing order, and scan conversion involved in the rasterization stage. Their geometry BW analysis is different than the one presented here because, at the time the paper was published, 3D accelerators implemented only the rasterization and subsequent stages of the rendering pipeline. Consequently, the system bus BW requirements were much higher (about 250MBytes/s) to transfer all the projected triangles data (mainly vertex transformed attributes) from CPU to the graphics board, and thus, the benefit of primitive compacted forms as strips or fans was certainly the key point. Also, the presented texture traffic analysis in this paper, using either CPU push or GPU pull transfer mode, is nowadays less relevant because current on-board memory sizes can practically host the entire texture working set for a game (thanks to using compressed textures).

Once clarified this point, our paper shows results not only about the texture traffic for the selected workloads, but also about the memory BW requirements for the different stages of the pipeline, besides affected by the presence of caches and

compression. In addition, our study attaches importance to the dynamic component involved in anisotropic texture requests, as key part of the fragment throughput performance.

The paper [2] describes some static properties of 3D scenes which do not change over the time such as geometry metrics and culling efficiencies. Whereas our study is completely focused on characterization of recent interactive 3D games (we think games will be the most important workload to guide future GPU designs), this work is exclusively focused on static VRML 3D models. The polygon culling analysis includes zero-sized polygons, that is more adequate for renderization of high detailed 3D models in which projected zero-sized triangles are more frequent (in contrast to 3D interactive games, which tend to rasterize larger triangles). It also shows measures of the pixel depth, defined as the number of rasterized polygons, hence neither culled nor clipped, that cover a given pixel. This measure is equivalent to the overdraw statistic at the rasterization stage shown in TABLE XI. Higher pixels depths or overdraw numbers are typical in 3D interactive games rather than in 3D models because games use multipass render algorithms, for example the stencil shadow volume technique used in Doom3 and Quake4.

In [3], authors describe methods for characterizing 3D graphics applications based on requests and traffic between the CPU and the graphics card through the AGP bus. This paper extends the geometry BW analysis in [1], showing interesting reference plots of WinMark executions, that visualize the communication between CPU/AGP and graphics card over time for different frames.

The work [4] is one of the first papers in 3D graphics workload characterization. It attaches importance to the tools needed to characterize 3D workloads. First, a tracing program that intercepts calls at the graphics library level and stores it in a trace file to further replay (allowing to replay exactly the same input several times). Then, a profiler to read trace file and compute workload statistics. And finally, a benchmark tool that runs the trace file over different graphics system in different platforms and measures the execution speed. With these three tools, the authors compare the relative performance between different graphics systems for the characterization metrics proposed (similar to those explained in [2]) evaluating them on a large benchmark set. Our tracing and simulation tools (see Section II) follow to a large extent the tracing methodology of this paper.

Other performance profiling applications for NVIDIA cards are NVPerfHUD [5], a useful developer tool for 3D pipeline bottleneck detection, and NVPerfKit [8], which provides D3D and OGL API instrumentation as well as hardware profiling via performance counters. For ATI cards, PIX plugins are available to gather statistics of ATI boards hardware performance counters [10].

Also, a set of benchmarks is available to stress the different parts of the 3D rendering pipeline: SPECViewPerf [11] is a benchmark set of real selected OGL applications that gives an overall performance mark in frames per second for the tested graphics board. GPUBench [12] is a set of small special-designed tests, each one giving a different measurement like

fillrates, latencies or BWs.

In [14], the authors present a set of 3D graphics workloads more focused on embedded 3D graphics. They show detailed simulation results for the rasterization pipeline and how there is an unbalanced utilization across rasterization pipeline units. They draw some architectural implication conclusions, proposing which rasterization units should be implemented and improved in future embedded generations and which should be implemented in software or with slow and low-power hardware implementations. Again, as with [1], the results only cover the utilization of texture units enabling trilinear filterings, implying a fixed amount of texture lookups, in contrast to the variable number of texture lookups of anisotropic filterings characterized in our work.

V. CONCLUSIONS

The work presented in this paper supposes an analysis of modern 3D games, combining the basic characterization at API call level with the microarchitectural behaviour of specific GPU units in terms of utilization, BW and efficiency. For the games selected, we have instrumented at the API call level and for some supported games we have gathered low-level statistics from ATTILA GPU simulator to complete the API level statistics.

At the API level we have computed the average index BW requirements and the results show that this traffic between CPU and GPU is relatively low and, thus, easily accommodated by current AGP/PCIe BW. We then have turned our attention to the geometry stage, and analyzed the average cost of transforming vertices using vertex programs. The analysis shows the post-shading vertex cache to be a very effective means to reduce the geometry computation requirements (reaching the expected 66% hit rate) and also explains the choice of triangle lists over other primitives such as strips or fans. To close the geometry analysis, the number of indexes, assembled triangles for those indexes and percentage of culled and clipped triangles are shown for the simulated benchmarks. Results indicate that both clipping and culling greatly contribute to reducing the load on the pipeline backend (rasterization and fragment shading).

Our data related to rasterization indicate that the average triangle size is still large in these workloads, with the number of fragments shaded per triangle in the 400 to 1000 range. The analysis of the ratio of fragment ALU instructions to texture instructions (and the associated bilinear sampling throughput) has yielded the very interesting result that the GPU architectures with higher shader processing power than texturing throughput may not be very efficient when running the OpenGL games we simulate. However our API level statistics show that more recent games (Fear, Oblivion) have a more favorable ALU processing to texture processing ratio and this trend is likely to continue in the near future. Optimizations not implemented in ATTILA can also be used to reduce the amount of texture processing required per texture request thus increasing the ALU to texture ratio. Finally we have turned our attention to memory behaviour and found that the various caches (Z, Texture and Color) are highly effective

at reducing overall GPU memory BW demand.

ACKNOWLEDGMENT

This work has been supported by the Spanish Ministry of Education and Science under grant TIN2004-07739-C02-01.

REFERENCES

- [1] T Mitra, T Chiueh, Dynamic 3D Graphics Workload Characterization and the Architectural Implications, 32nd ACM/IEEE Int Symp. on Microarchitecture (MICRO), p.62-71, November 16-18, 1999, Haifa, Israel.
- [2] Tzi-cker Chiueh, Wei-jen Lin, Characterization of static 3D graphics workloads, Proceedings of the 1997 SIGGRAPH/Eurographics workshop on Graphics hardware, p.17-24, August 03-04, 1997, Los Angeles, California, United States.
- [3] Poursepanj, A.; Christie, D., Generation of 3D graphics workload for system performance analysis, Workload Characterization: Methodology and Case Studies, 1998, 29 Nov. 1998 p.36 – 45.
- [4] J. C. Dunwoody and M. Linton, Tracing Interactive 3D Graphics Programs, Computer Graphics (Proc. Symp. Interactive 3D Graphics), 24(2), pp. 155-163, 1990.
- [5] NVPerHUD Home page at NVIDIA developers Site: http://developer.nvidia.com/object/nvperhud_home.html
- [6] Joel McCormack and Robert McNamara. Tiled Polygon Traversal Using Half-Plane Edge Functions. In Proc. Eurographics/SIGGRAPH Workshop on Graphics Hardware, pages 15–21, 2000
- [7] B. Paul. The Mesa 3D Graphics Library. <http://www.mesa3d.org>.
- [8] NVPerfKit 1.1 and Instrumentalized driver for NVIDIA cards: http://developer.nvidia.com/object/nvperkit_home.html.
- [9] V. Moya et al. ATTILA: A Cycle-Level Execution-Driven Simulator for Modern GPU Architectures. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2006) March 19-21, 2006, Austin, Texas.
- [10] ATI plugin for Microsoft's PIX performance analysis tool: <http://www.ati.com/developer/atipix/index.html>
- [11] SPECViewPerf 8.1: <http://spec.unipv.it/gpc/opc.static/vp81info.html>
- [12] GPUBench. How much does your GPU bench?: <http://graphics.stanford.edu/projects/gpubench/>
- [13] Scott Wasson, DOOM 3 high-end graphics comparo - Performance and image quality examined: <http://techreport.com/etc/2004q3/doom3/>
- [14] I. Antochi, B. Juurlink, S. Vassiliadis, and P. Liuha, GraalBench: A 3D Graphics Benchmark Suite for Mobile Phones, in Proceedings of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04), June 2004.
- [15] H. Hoppe. Optimization of mesh locality for transparent vertex caching. In Computer Graphics Proc., Annual Conf. Series (SIGGRAPH '99), ACM Press, , pages 269–276, 1999.
- [16] Thomas McGuire. Doom3 Tweak Guide @ TechSpot: <http://www.techspot.com/tweaks/doom3/>
- [17] M. McCool and C. Wales and K. Moule, Incremental and Hierarchical Hilbert Order Edge Equation Polygon Rasterization, Eurographics/SIGGRAPH, 2001.
- [18] S. Morein. ATI Radeon Hyper-z Technology. In Hot3D Proceedings - Graphics Hardware Workshop, 2000.
- [19] Imagination Technologies, PowerVR Tile-based rendering, GDC 2001 Talk, <http://www.pvrdev.com/pub/PC/doc/f/PowerVR%20Tile%20based%20rendering.htm>
- [20] Beyond 3D - ATI Xenos: XBOX 360 Graphics Demystified: <http://www.beyond3d.com/articles/xenos/>
- [21] Beyond 3D - ATI Radeon X1600 XT and X1300 PRO: <http://www.beyond3d.com/reviews/ati/rv5xx/>
- [22] Beyond 3D - ATI R580 Architecture interview: <http://www.beyond3d.com/reviews/ati/r580/int/>
- [23] DirectX SDK: http://msdn.microsoft.com/library/en-us/directx9_c/dxsdk_tools_performance_pix.asp
- [24] SPEC benchmarks: <http://www.spec.org/cpu2000/>
- [25] TPC benchmarks: <http://www.tpc.org/tpcc/detail.asp>
- [26] MediaBench benchmarks: <http://euler.slu.edu/~fritts/mediabench/>
- [27] Beyond 3D - ATI R520 Architecture review: <http://www.beyond3d.com/reviews/ati/r520/>
- [28] J. McCormack, R. Perry, K. I. Farkas, N. P. Jouppi, Feline: fast elliptical lines for anisotropic texture mapping, Proceedings of the 26th annual conference on Computer graphics and interactive techniques, July 1999.