A Low-Power Implementation of 3D Graphics System for Embedded Mobile Systems

Chanmin Park, Hyunhee Kim and Jihong Kim
School of Computer Science & Engineering, Seoul National University, Seoul, Korea
{cmpark, hh0726, jihong}@davinci.snu.ac.kr

Abstract

For mobile 3D graphics systems, even though performance requirements are met, an efficient power management is even more important for battery-powered mobile devices since they require a large number of arithmetic operations as well as a high frequency of memory accesses.

According to the analysis of the power consumption of mobile 3D graphics pipelines and the slacks across the pipeline stages, we describe intra-frame and inter-frame DVS low-power techniques reducing the power consumption of mobile devices based on a variable voltage processor.

Our implementation on a PDA development board shows that the proposed DVS techniques achieve an energy saving of up to 46% over a non-DVS implementation.

1. Introduction

As mobile devices get more powerful, more desktop PC applications including 3D graphics are moving into the mobile domain. Even though performance requirements are met, reducing power consumption is an important design requirement for battery-powered mobile devices such as PDA and cellular phones. For mobile 3D graphics applications such as 3D games and 3D navigations, an efficient power management is even more important since they require a large number of arithmetic operations as well as a high frequency of memory accesses, making them power-hungry applications.

Previously, low-power 3D graphics have been investigated at different abstraction levels including the circuit level, architecture level and algorithm level. In this paper, we analyze the power consumption of mobile 3D graphics pipelines, and show that there exist imbalances, i.e. slacks, across the pipeline stages based on the dynamic 3D graphics workloads for each 3D graphics application and its characteristics. The slacks can occur in the differences due to the 3D graphics features such as frame rate, the number of primitives, lighting parameters, texture mapping parameters, and the number of fragments, etc.

Based on this observation - energy consumption analysis of typical mobile 3D graphics applications, we propose *intra-frame* and *inter-frame* DVS (Dynamic Voltage Scaling) power saving techniques for mobile 3D graphics systems, which can be useful for mobile devices based on variable voltage processors. The *intra-frame* DVS technique estimates the required workload of each frame based on the characteristics of scene description of a given frame. If the required workload is less than the expected workload, the supply voltage is accordingly adjusted. While the *intra-frame* DVS technique exploits the slack time within the current frame, the *inter-frame* DVS technique takes advantages of unused idle intervals from previous frames.

In order to design and implement our proposed DVS techniques, we develop an energy-efficient *software* implementation of a 3D graphics library, for mobile devices based on a variable-voltage processor. As a specific target platform, we use a prototype PDA system running Linux on Intel's XScale PXA255. Our implementation on the target PDA development board shows that the proposed DVS techniques achieve an energy saving of up to 46% over a power-unaware implementation.

The rest of the paper is organized as follows. In Section 2, we introduce some background about 3D graphics pipeline and Section 3 presents motivational examples and the analysis of power consumption. In Section 4, we describe the system model for DVS. Section 5 explains the proposed DVS scheme. The implementation and experimental results are discussed in Section 6 and Section 7 concludes the paper with future directions.

2. 3D Graphics Pipeline

There are several emerging standard APIs for mobile 3D graphics such as OpenGL ES [3], Java mobile 3D Graphics API (JSR-184), and Direct 3D Mobile, etc.

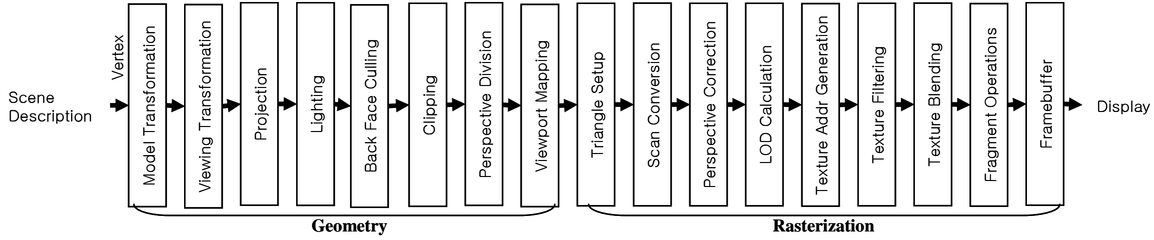


Figure 1: 3D graphics pipeline

OpenGL ES defines a standardized cross-platform API for full-function 2D and 3D graphics on embedded systems, which is a well-defined subset of OpenGL. In this paper, we use several 3D graphics terms limited to OpenGL ES.

The 3D graphics pipeline structure is shown in Figure 1. This pipeline can be broadly divided into two phases, `Geometry` and `Rasterization`. The pipeline stages in the `Geometry` phase require a large number of floating calculations per vertex while the pipeline stages in the `Rasterization` phase need a large number of memory accesses per fragment. The `Triangle Setup` stage comprises per-triangle floating-point operations for `scan conversion` which generate fragments. `Texture mapping` and `Fragment operations` fills the fragments with the appropriate color (via z-test, alpha test, etc.) and `Framebuffer` shows the created image to the display screen.

3. Analysis of Power Consumption

We analyze the power consumption of 3D graphics pipeline using three applications (Figure 2). `Texsub` is a simple texture mapping tutorial scene of OpenGL. `Face model` is a 3D character model, and `Jelly fish` is a 3D shooting game [18]. In order to gather the statistics of applications, we implement a behavioral simulator for OpenGL ES and also its S/W implementation on our target board for physical measurements of energy consumption. The implementation is explained in Section 6.

a. Texsub b. Face model c. Jelly fish

Figure 2: Applications

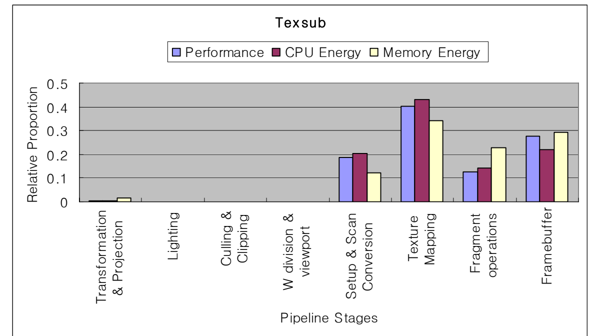
Table 1: The statistics of applications features

Application	Vertex	Triangle	Fragment	Texel access	Time(sec)	Lighting
Texsub	8	4	24388	24388	0.161571	x
Face model	4281	1427	16562	16562	0.806431	o
Jellyfish (average)	9187	3073	47070	47006	0.669926	x

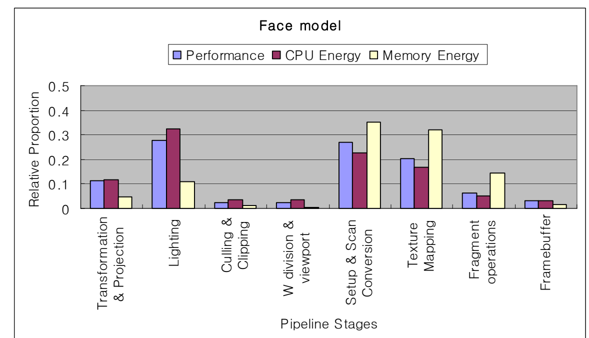
Figure 3 shows the relative proportions of performance (execution time) and energy consumption for each pipeline stage of those applications whose characteristics are summarized in Table 1.

`Texsub` has a small number of vertices and large triangles, spends most its energy during the `Rasterization` phase. On the other hand, `Face model`, which has a relatively large number of vertices with lighting and small triangles, consumes about 52% of the CPU energy on `Geometry` phase. `Jelly fish` has a large number of vertices and a relatively large number of fragments. (However, we note that these analyzed patterns of energy consumption can be also changed depending on the optimization techniques for each pipeline stage.)

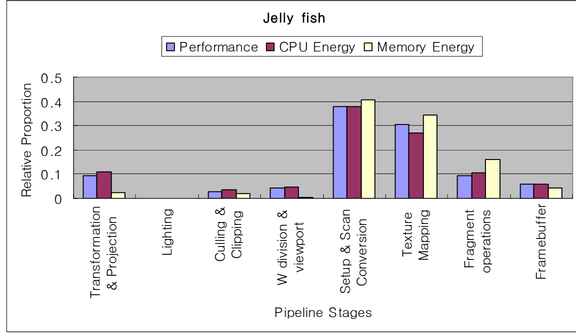
Thus, we propose DVS methods for low-power 3D graphics by using these imbalances (slacks) dependent upon applications' workloads.



a. Texsub



b. Face model



c. Jelly fish

Figure 3: The distribution of energy consumption and performance of three applications

4. System Model

Considering general 3D graphics architecture with any possible optimizations, a pipeline is consist of n stages, which is represented as P = {$p_{stage\text{-}1}$, …, $p_{stage\text{-}n}$}. A 4-tuple {S_i, $P_{th\text{-}i}$, C_i, N_i } is used to represent each pipeline stage $P_{stage\text{-}i}$, where S_i is the state enabled or disabled by graphics feature, $P_{th\text{-}i}$ is the throughput factor which is the parallelism determined in design time and C_i is the worst case execution time (WCET) of the pipeline stage at the maximum processor speed. The graphics features in 3D graphics are the global states determined by the parameters of glEnable (glDisable) command of OpenGL ES, shading model, lighting parameters, texture mapping parameters before the beginning of drawing each scene. N_i is the iteration factor based on the number of primitives, the number of fragments, and depth complexity. Hence, the execution path and execution time of each pipeline stage can be changed depending on these features. Parallelism is defined in design time for optimization and means that how many operations can be done in time, for example, parallel vector operations on SIMD architecture or several pixel-element processors which can process several fragments. Therefore, the execution time of jth frame can be stated as (1):

$$D_j = \sum \frac{C_i S_i N_i}{P_{th-i}}, \quad (1 \le i \le n) \qquad (1)$$

Since C_i and $P_{th\text{-}i}$ are determined in design time, they are fixed values, and the frame deadline and bottleneck come from application's workloads. Even though the pipeline is well optimized, there can exist slack times due to the imbalances occurred in differences from S_i and N_i, depending on applications.

When the execution time B_j of the bottleneck stage of whole pipeline in the frame is (2), the other stages can process more their inputs or have slack times. This means we can have chance to optimize the performance via load-balancing, or slowdown the supply voltage by using these slack times in the system whose the frame rate is 1/max(D_j).

$$B_j = \max\left\{\frac{C_i S_i N_i}{P_{th-i}}\right\}, \qquad (1 \le i \le n) \qquad (2)$$

Usually, while pixel fill-limited applications which have a small number of large triangles tend to be memory-limited (i.e. rasterization-limited), geometry-limited applications which have a large number of small triangles are compute- (or interfacing buffer) limited. The designers of graphics architecture choose fill-rate and fix memory bandwidth based on cost-effective memory technology, and determine triangle rate - processor capability in the design phase. Therefore, in this design phase, the performance goal is selected and then the number of pipeline stages and the capability of each component are defined.

Many researchers have focused on optimizations on the components of each pipeline stage by efficient s/w and h/w design [6-17]. The GPU described in [6, 7] can deactivate the unused components and [7] gives frequency scaling functionality. Texture mapping has been investigated for low-power using DVS based on a human visual perception model [8].

Despite all optimizations, there can exist slack times due to the imbalances occurred in the non-bottleneck stages. They show the analysis of 3D graphics workload depending on several features and propose basic DVS scheme in [4]. However, their history based approach is naive and they do not consider frame-by-frame variations in dynamic workload. In this paper, we specifically focus on DVS techniques for mobile 3D graphics.

5. Intra-Frame and Inter-Frame DVS

Figure 4 shows two levels of DVS for 3D graphics applications. In *inter-frame* DVS, the voltage is adjusted by a frame granularity based on the slack times generated from the previous frame. On the other hand, in *intra-frame* DVS, the voltage is adjusted by an object granularity within a frame.

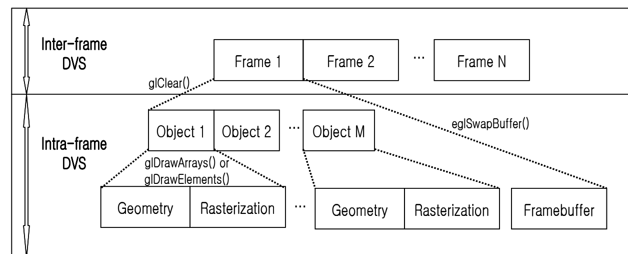


Figure 4: The Inter-frame vs. Intra-frame DVS

In this paper, we assume a frame starts from glClear() call and ends with eglSwapBuffer() call, and each object of a scene is distinguished by glDrawArrays() or glDrawElements() from OpenGL ES applications. A frame deadline is the time period to display one scene with a given frame rate at the highest frequency. An object deadline is the time to finish whole pipeline processing for that object. Since a scene (a frame) can have several (*m*) objects, we restate the equation (1) as (3):

$$D_j = \sum_o \sum_i \frac{C_i S_i^o N_i^o}{P_{th-i}}, \quad (1 \le i \le n, 1 \le o \le m) \quad (3)$$

In *intra-frame* DVS, each object can have static slacks due to (2) or dynamic slacks between objects. The latter uses the slack from the previous object. We assume when the frequency is changed to f_k, the voltage level is also proportionately set to V_k, where the range of scalable voltages and frequencies is 1 to s and $1 \le k \le s$. When we use the slack times for adjusting the supply voltage in *intra-frame* DVS, the frequencies is determined for static (4) or dynamic distribution (5) of the slack.

$$F_o^{static} = \frac{\sum_{i=1}^{n} \frac{C_i S_i^o N_i^o}{P_{th-i}}}{\sum_{i=1}^{n} \frac{C_i S_{all}^o N_i^o}{P_{th-i}}}, \quad (4)$$

where S_{all}^o means all features S_i^o are enabled.

$$F_{o+1}^{dynamic} = \frac{\sum_{j=o+1}^{m} \sum_{i=1}^{n} \frac{C_i S_i^j N_i^j}{P_{th-i}}}{\sum_{j=1}^{m} \sum_{i=1}^{n} \frac{C_i S_{all}^j N_i^j}{P_{th-i}} - \sum_{j=1}^{o} \sum_{i=1}^{n} \frac{C_i S_i^j N_i^j}{P_{th-i}}}, \quad (5)$$

where S_{all}^j means j^{th} object has all enabled features.

$$F_i^o = F_o^{static} \cdot \frac{\left(\frac{C_i S_i N_i}{P_{th-i}}\right)}{B_j} \quad (6)$$

The F_o^{static} in (4) is determined by distributing slack time evenly to each pipeline stage for drawing an object. This static slack-distribution can avoid too frequent voltage scaling. However, it cannot be fair depending on the bottleneck stage (2). So, we also consider frequency for each pipeline stage in (6). Using dynamic distribution (5) of the slack can compensate the misprediction in the previous stage.

Since the execution time of each object can vary according to its dynamic characteristics frame by frame, we construct an object list as a preliminary workload estimator and store the object's characteristics including number of vertices, triangles, fragments and execution time while drawing a scene. When the first frame is rendered, the object list is created and updated frame by frame. When updating the object list, the variations on the characteristics is also stored for the correct prediction of the slack times of the consecutive frames or objects.

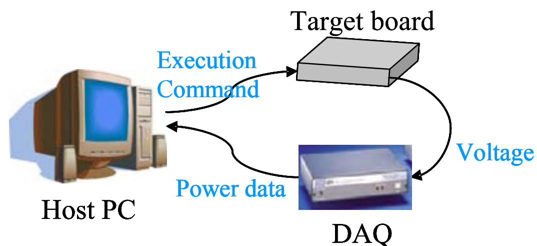
Furthermore, slack times can be made by the variation of the characteristics in a scene or the movement of objects. Since the intra-frame estimation is a conservative approach, it cannot find all the slack times in advance. Such unused dynamic slacks are added to the deadline for the next frame. We call this approach *inter-frame* DVS. In this paper, the slack from the previous frame is used by the first object in the next frame. In *inter-frame* DVS, we assume the frame deadline is a constant for each frame. Determining the frequency for *inter-frame* DVS is similar to (5). In this paper, we implement the proposed DVS techniques at a level of 3D graphics library. If the frame rate is controlled by an application itself, however, the *inter-frame* DVS has no effect on having slack time, since we cannot start processing the next scene earlier at a library.

In order to generate more intra-frame slacks for our implementation, we add vertex caching technique to the object list. This technique can avoid repetitive transformations and lighting calculations of shared vertices, since triangles neighboring with each other usually share vertices in many 3D applications. We construct a vertex cache structure by a binary search tree before `transformation` stage. When a new vertex enters the `transformation` stage, we first search the vertex cache to check whether the same vertex has been processed before. If there is a match, we reuse the previous results for the vertex, thus skipping computationally expensive `transformation` and `lighting` stages. Especially for applications that need a sophisticated lighting model, the vertex cache can keeps away from very heavy lighting calculations many times, since shared vertices possibly have the same averaged normal vectors from neighboring triangles in many applications for smooth shape.

6. Implementation and Experiment

We have implemented the proposed techniques with a software implementation of a 3D graphics library on a prototype PDA system of which CPU and memory are power measurable separately. We have implemented an OpenGL ES 1.1 library running on Embedded Linux as a baseline and modified it to include our proposed DVS techniques. The infrastructure of measurements is shown in Figure 5.

Our system environments are as follows: The main CPU is an Intel PXA255, which can change the clock frequency to one of 7 levels between 100 MHz and 400 MHz, and the target board has a programming core voltage regulator; supply voltage can scale to one of 3 levels between 1.0 V and 1.3 V. The LCD display has a color depth of 16 bits at a 320 x 240 screen resolution, and 3D graphics accelerations are not supported.



a. Physical measurements of power consumption

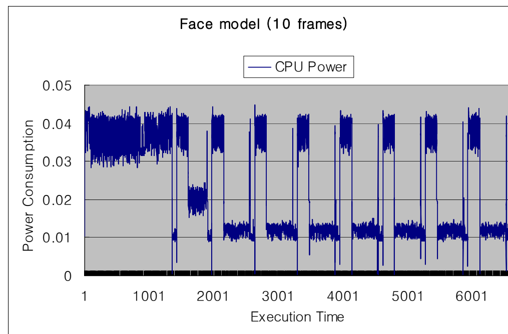
b. Target board and DAQ

Figure 5: Target PDA prototype system

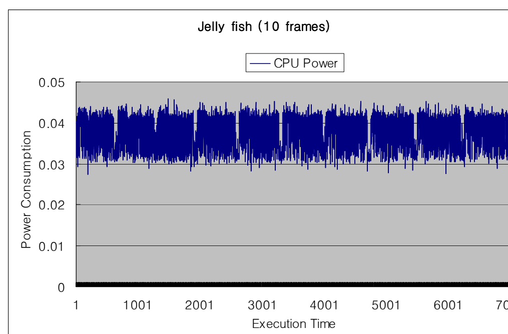
In our experiment, we have evaluated the efficiency of the proposed approach using three applications (Figure 2).

Figure 6 shows the power consumption patterns of the results of applying DVS to `Face model` and `Jelly fish` along their 10 frames, where the measured powers are relative values. The values of power consumption are relative to watt (W) and the values of execution time to milliseconds (ms). The power consumption of original (non-DVS) executions has a pattern in Figure 5(b) and others result from DVS.

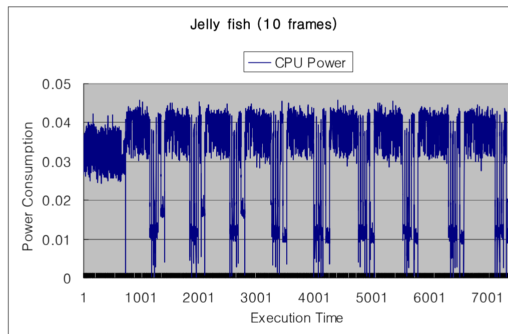
In `Face model`, energy gain was 47% of CPU and 43% of memory, 46% in the total energy consumption. The performance of `Face model` benefited from the vertex caching by 43% and thus the voltage was more effectively adjusted every frame. Since our target platform has discrete level of frequencies available, more fragmented slacks generated in the `geometry` can be added and used in the `rasterization` phase. `Jellyfish` has 40 or more moving objects each frame, and the number of objects varies frame by frame.



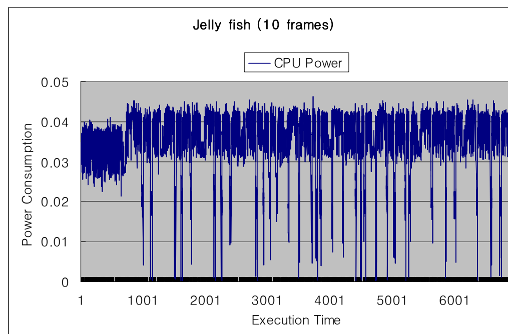
a. Face model: DVS



b. Jelly fish: non-DVS



c. Jelly fish: DVS with performance overhead



d. Jelly fish: DVS without performance overhead

Figure 6: The results of applying DVS

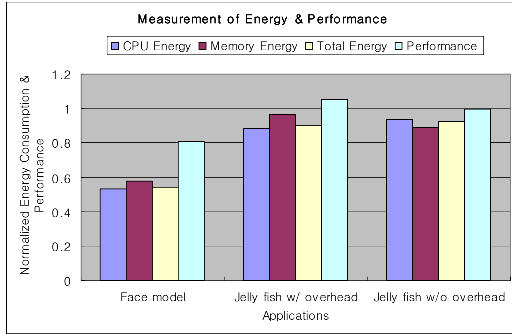


Figure 7: The experimental results

The workload can vary depending on the number of objects and in the experiment, the object list (estimator) has a threshold to validate the estimation of dynamic variations and control the error tolerance of it. Depending on the threshold values, we saved 12% total energy with 5% performance overhead and 8% without performance overhead in `Jellyfish` (Figure 7).

7. Conclusions

We described DVS scheme for general 3D graphics and introduced low-power intra-frame and inter-frame DVS techniques into 3D graphics pipeline stages based on the statistics of applications' features. And we implemented an energy-efficient software implementation of the OpenGL ES 1.1 API, for mobile devices based on a variable-voltage processor.

In our implementation, the object list with vertex cache was useful to make more slacks and manage variations for both intra-frame and inter-frame DVS. Our implementation on a PDA development board shows that the proposed DVS techniques achieve an energy saving of up to 46% over a power-unaware implementation.

Even though our software-only implementation has performance limitations as usual in 3D graphics, we give the feasibility of efficient power-saving DVS schemes applicable to 3D graphics system. And our platform can help flexible simulation to design low-power mobile 3D graphics system. We are evaluating several optimization techniques suitable for low-power mobile 3D graphics. In addition to the DVS techniques, DPM techniques need to be considered when implementing 3D accelerators, since the leakage power is increasing with the CMOS technology generation.

References

[1] Tomas Akenine-Moller, Eric Haines, "Realtime rendering second edition", AK PETERS, 2002.
[2] Dave Shreiner, Jackie Neider, Mason Woo, Tom Davis, "OpenGL Programming Guide 4th edition", Addison-Wesley, 2004.
[3] The Khronos Group, "OpenGL ES Overview", Available at http://www.khronos.org/opengles/index.html.
[4] B.C. Mochocki, K. Lahiri, and S. Cadambi, "Power Analysis of Mobile 3D Graphics", Proceedings of the Design Automation and Test in Europe Conference - DATE '06 pp. 502-508, 2006.
[5] T. Mitra and T. Chiueh. "Dynamic 3D Graphics Workload Characterization and the Architectural Implications", In Proceedings of the 32nd Annual ACM/IEEE International Symposium on icroarchitecture (MICRO), pages 62–71, November 1999.
[6] Masatoshi Kameyama, et al, "3D Graphics LSI Core for Mobile Phone Z3D", Graphics Hardware, The Eurographics Association, 2003.
[7] R. Woo, et al., "A 210mW Graphics LSI Implementing Full 3D Pipeline With 264MTexels/s Texturing for Mobile Multimedia Applications", in Proceedings, IEEE International Solid-State Circuits Conference, February 2003.
[8] J. Euh, J. Chittamuru, and W. Burleson, "A Low-Power Content-Adaptive Texture Mapping Architecture for Real-Time 3D Graphics", 2002 Workshop on Power-Aware Computer Systems (PACS'02).
[9] J. Chittamuru, J. Euh, and W. Burleson, "Dynamic Word length Variation for Low-Power 3D Graphics Texture Mapping", IEEE Workshop on Signal Processing Systems 2003.
[10] Tomas Akenine-Moller, Jacob Strom, "Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones", in Proceeding of ACM SIGGRAPH, ACM Transaction on Graphics 2003.
[11] W. Park, K. Lee, I. Kim, T. Han, S. Yang, "An Effective Pixel Rasterization Pipeline Architecture for 3D Rendering Processors", IEEE Transaction on Computers, Vol. 52, No. 11, pp. 1501-1508, Nov. 2003.
[12] Z.S. Hakura and A. Gupta, "The Design and Analysis of a Cache Architecture for Texture Mapping", Proceedings of the 24th International Symposium on Computer Architecture, pp. 108-120, June 1997.
[13] H. Igehy, M. Eldridge, and K. Proudfoot, "Prefetching in a Texture Cache Architecture", Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware, pp. 133-142, Aug. 1998.
[14] M. Cox, N. Bhandary, and M. Shantz, "Multi-Level Texture Caching for 3D Graphics Hardware", Proceedings of ACM/IEEE International Symposium on Computer Architecture (ISCA), 1998.
[15] N. Greene, M. Kass, and G. Miller, "Hierarchical z-Buffer Visibility", Proc. SIGGRAPH'93, pp. 231-238, Aug. 1998.
[16] J. Bae, et al., "An 11M-Triangles/sec 3D Graphics Clipping Engine for Trianlge Primitives", IEEE international Symposium on Circuits and Systmes, pp. 4570-4573, 2005.
[17] D. Kim, L. S. Kim, "Division-free rasterizer for perspective-correct texture filtering", in Proc. IEEE Int. Circuits and Systems Sump., vol.2, pp 153-15, May, 2004.
[18] The OpenGL ES 1.1 Coding Challenge, http://www.khronos.org/devu/opengles_challenge/