Memory Bandwidth Requirements of Tile-Based Rendering

Iosif Antochi[1], Ben Juurlink[1], Stamatis Vassiliadis[1], and Petri Liuha[2]

[1] Computer Engineering Laboratory,
Electrical Engineering, Mathematics and Computer Science Faculty,
Delft University of Technology,
Mekelweg 4, 2628 CD Delft, The Netherlands
{tkg,benj,stamatis}@ce.et.tudelft.nl
[2] NOKIA Research Center, Visiokatu-1, SF-33720 Tampere, Finland
petri.liuha@nokia.com

Abstract. Because mobile phones are omnipresent and equipped with displays, they are attractive platforms for rendering 3D images. However, because they are powered by batteries, a graphics accelerator for mobile phones should dissipate as little energy as possible. Since external memory accesses consume a significant amount of power, techniques that reduce the amount of external data traffic also reduce the power consumption. A technique that looks promising is tile-based rendering. This technique decomposes a scene into tiles and renders the tiles one by one. This allows the color components and z values of one tile to be stored in small, on-chip buffers, so that only the pixels visible in the final scene need to be stored in the external frame buffer. However, in a tile-based renderer each triangle may need to be sent to the graphics accelerator more than once, since it might overlap more than one tile. In this paper we measure the total amount of external data traffic produced by conventional and tile-based renderers using several representative OpenGL benchmark scenes. The results show that employing a tile size of 32×32 pixels generally yields the best trade-off between the amount of on-chip memory and the amount of external data traffic. In addition, the results show that overall, a tile-based architecture reduces the total amount of external data traffic by a factor of 1.96 compared to a traditional architecture.

1 Introduction

A huge market is foreseen for wireless, interactive 3D graphics applications, in particular games[1]. The resources required to run such applications are usually considerable, while the resources offered by low-power devices are rather limited. Specifically, since such devices are powered by batteries they should consume as little energy as possible. Furthermore, the amount of chip area of such systems is severely limited.

External memory accesses are a major source of power consumption [2, 3]. They often dissipate more energy than the datapaths and the control units. A promising technique to reduce the amount of external data traffic and, hence, the power-consumption is tile-based rendering. This technique decomposes a scene into tiles and renders the tiles one by one. The advantage of this is that the color components and z values of one tile can be stored in small, on-chip buffers, so that only the pixels visible in the final scene need to be stored in the external frame buffer. In other words, tile-based rendering reduces the problem of overdraw. On the other hand, however, a tile-based renderer may require that each triangle to be sent to the graphics accelerator more than once, since a triangle might overlap more than one tile. Moreover, given that the amount of chip area is limited, very little area can be devoted to local memory. So, we are faced with two opposite goals: reduce the amount of external memory traffic as much as possible while, at the same time, using as little internal memory as possible.

In this paper we examine the memory bandwidth requirements for tile-based 3D graphics accelerators. First, we examine how the amount of external data traffic varies with the tile size. The results show that a tile size of $32{\times}32$ pixels yields the best trade-off between the data traffic volume and the amount of area dedicated to on-chip buffers. By increasing the tile size beyond 32×32 pixels, the amount of external data traffic is only marginally reduced. Also, decreasing the tile size under 32×32 pixels, increases the external data traffic substantially. Second, we measure how much external data traffic is saved by a tile-based renderer compared to a traditional renderer. The results show that the tile-based architecture reduces the total amount of external data traffic by a factor of 1.96 compared to the traditional architecture.

This paper is organized as follows. After discussing related work in Section 2, we describe the conventional and tile-based rendering models in Section 3. In Section 4, the results of our experiments are presented. First, we determine the influence of the tile size on the amount of external data traffic. Second, we measure how much external data traffic is saved by employing a tile-based architecture instead of a conventional architecture. Conclusions and future work are given in Section 5.

2 Related Work

Tile-based architectures were initially proposed for high-performance, parallel renderers [4–6]. Since tiles cover non-overlapping parts of a scene, the triangles that intersect with these tiles can be rendered in parallel. In such architectures it is important to balance the load on the parallel renderers [7, 8]. These studies, however, present statistics relevant for high-end parallel graphics, while we consider a low-power architecture in which the tiles are rendered sequentially one-by-one and the available memory bandwidth is rather limited when compared with parallel tile-based rendering.

Low-power tile-based architectures [9, 10] were proposed, but no measurements of the total required bandwidth were presented. For instance, in [9] it is shown only that the tile-based approach reduces the traffic between the renderer and the external memory for various scene sizes. However, there is no indication of how the overall data traffic is affected by the tile size. The work presented in [10] focuses on compatibility problems for tile-based renderers and shows how they can be avoided.

Other papers discussing tile rendering (e.g., [11]) are mainly concerned by the *overlap* (the number of tiles that a primitive covers) of triangles with respect to tile size. Only the traffic between the CPU or main memory to the accelerator was considered. We consider the total data traffic, i.e., not only the traffic from the CPU or main memory to the accelerator but also the traffic between the accelerator and the frame buffer.

The amount of external data traffic can also be reduced by decreasing the amount of transfered texture data. Three techniques to reduce the texturing traffic are texture

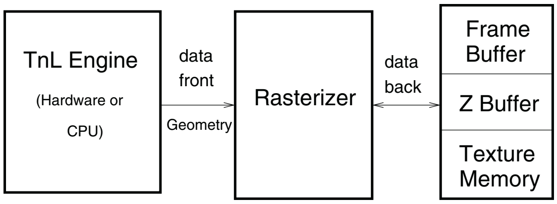


Fig. 1. Organization of a traditional renderer.

caching, texture compression, and texturing mechanism. Texture caching is one of the most efficient techniques to reduce the data traffic generated by a renderer. In [12] it is shown that even a 256-byte, direct-mapped cache can significantly reduce the amount of texture traffic. By using texture compression [13, 14] texture traffic can be even further reduced since less data is transferred from the texture memory to the renderer. Recently Möller and Ström proposed a new texturing mechanism called Pooma [15]. Pooma fetches fewer texels from the texture memory than traditional methods at the expense of slightly lower image quality.

3 Rendering Methods

In this section we describe the basic organization of a traditional and a tile-based renderer and briefly discuss the factors that contribute to the amount of external data traffic.

The organization of a conventional rasterizer is depicted in Fig. 1. The *Transform and Lighting* (*TnL*) *Engine* processes the geometry data at the vertex level. It performs primitives coordinate changes and lighting computations. Modern PC-class graphics accelerators perform (part of) the transform and light stages, but since our target platform is a low-power, low-cost accelerator for mobile phones, we assume they are performed by the host CPU. After the vertices have been processed, they are sent to the rasterizer as primitives such as points, lines, and triangles. After that, the rasterizer scan-converts each triangle into fragments (pixels which may or may not be visible in the final image). It also performs texturing if it is enabled. Finally, for each fragment it is determined if the fragment is obscured by another fragment using, for instance, a z (depth) buffer algorithm. One access to the z buffer is needed to retrieve the old z value and, if the current fragment is closer to the viewpoint than all previous fragments at the same position, the z value of the current fragment is written to the z buffer. Furthermore, if the z test succeeded, the current fragment is written to the frame buffer.

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

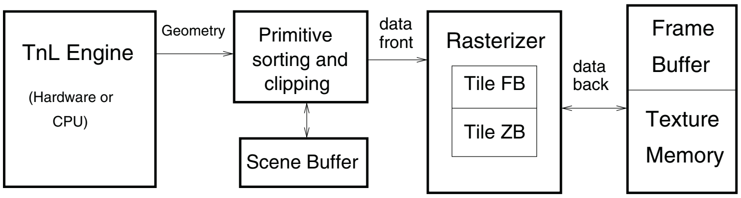


Fig. 2. Organization of a tile-based renderer.

The organization of a basic tile-based renderer is depicted in Fig. 2. First, the primitives are sorted according to the tile partitioning and stored in a so-called scene buffer. This may be performed by the CPU or the rasterizer. In this paper it is assumed that it is performed by the CPU. Furthermore, in our measurements we do not include the data traffic required to sort the primitives because it can be accomplished in many different ways. After the primitives have been sorted, the tiles are rendered one by one. First, the primitives that overlap the current tile are sent to the rasterizer. Thereafter, the data associated with the current tile is fetched from the external frame buffer to the local tile frame buffer. This is necessary for blending if the application does not clear the frame buffer when it starts to render a new frame. Furthermore, the tile z buffer is cleared. If the application clears the frame buffer when it starts to render a new frame, the external z buffer is no longer needed. If not, the old z values have to be fetched from the external z buffer. After all primitives that overlap the current tile have been rendered, the content of the tile frame buffer is written to the external frame buffer.

We now briefly describe the amount of external data traffic generated by each method. As depicted in Fig. 1 and Fig. 2, the data traffic between the graphics chip and other components of the system such as the CPU, main memory, and the frame buffer can be divided into two categories:

1. The data sent from the host CPU (or main memory) to the accelerator. It consists of geometrical data needed to describe the primitives, texture data, and changes to the state of the rasterizer such as enable/disable depth test, change texture wrapping mode, etc. We refer to this component as *data_front*.
2. The data transferred between the accelerator and dedicated graphics memory or memories. Since the frame buffer, z buffer, and texture memory are too large to be placed on-chip, they must be allocated off-chip. This component will be referred to as *data_back*.

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

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

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

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

4 Experimental Results

In this section the experimental results are presented. First, in Section 4.1, the benchmarks, tools, and some simulation parameters are described. Thereafter, in Section 4.2 we study how the amount of external data traffic varies with the tile size. Finally, in Section 4.3, we compare the total amount of off-chip memory traffic produced by a tile-based renderer to the amount of traffic generated by a conventional renderer.

4.1 Experimental Setup

In order to compare the traditional and tile-based architectures we used 6 of the 7 components of the benchmarking suite proposed in [16]: Q3H, Tux, Aw, ANL, GRA, and DIN. The Q3H profile corresponds to a demo of the Quake III 3D FPS game. Tux is a 3D racing game available on Linux platforms. The Aw (Awadvs-04) profile is part of the Viewperf 6.1.2 package. The ANL, GRA, and DIN are 3D VRML models for which "fly-by" scenes were created and traced.

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

Because texturing exhibits high data locality and because small, direct-mapped caches do not require a large amount of area nor consume a significant amount of

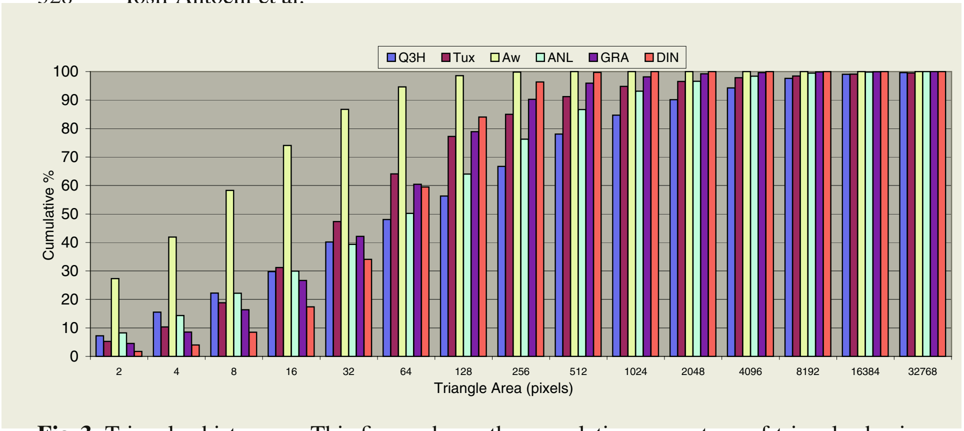


Fig. 3. Triangles histogram. This figure shows the cumulative percentage of triangles having an area lower than a defined size.

power [12, 17], we have employed a tiny (256-byte) direct-mapped, on-chip texture cache. We have used our own trace-driven cache simulator to measure the miss ratio of the texture cache.

In order to simulate the 9 possible tile sizes configurations for all workloads on our rasterizer simulator in an acceptable time interval (several weeks), we have rendered approximatively 60 frames from each workload evenly distributed across the workload. For the skipped frames, however, we have not skipped the state change information so that the appropriate state information was committed to the renderer before each frame was rendered.

4.2 Tile Size vs. External Data Traffic

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

As a first indication of an appropriate tile size, Fig. 3 depicts the cumulative percentage of triangles up to a certain size. It can be seen that by far the most (93%) triangles are smaller than 1024 pixels. Very few triangles (7%) are larger than 1024 pixels. This indicates that more than 93% triangles might fit in a tile size of 1024 (e.g., 32×32) pixels. However, even if most triangles are smaller than say, 32×32 they still can overlap multiple tiles if the tile size is 32×32. Therefore, a better indication of an appropriate tile size is the number of triangles sent to the rasterizer.

Table 1 depicts the number of triangles transferred to the rasterizer for various tile sizes. As explained in Section 3, this data usually dominates the *data_front* component of the total external data. The last row shows the number of triangles transferred if the tile size is equal to the window size. The overlap factor for a specific tile size can, therefore, be obtained by dividing the number of triangles transferred for that tile size by the number given in the last row.

Obviously, if the tile size increases, the number of triangles transferred to the rasterizer decreases, since there is less overlap. However, as remarked before, it is important to use as little internal memory as possible and, therefore, a trade-off needs to be made. It can be seen that using a tile size of less than 32×32 can increase the number of triangles transferred significantly. For example, if we employ a tile size of 16×16 instead of 32×32, the amount of geometrical data sent to the rasterizer increases by a factor of 2.02 for the Q3H benchmark and by a factor of 1.97 for the Tux benchmark. On average, using the geometric mean, a tile size of 16×16 increases the number of triangles sent by a factor of 1.62 compared to 32×32 tiles. On the other hand, employing tiles larger than 32×32 reduces the amount of geometrical data only marginally. For example, the geometric mean of the reduction resulting from employing 64×64 tiles instead of 32×32 tiles is 1.35. This indicates that a tile size of 32×32 is the best trade-off between the number of triangles sent to the rasterizer and the size of the internal buffers.

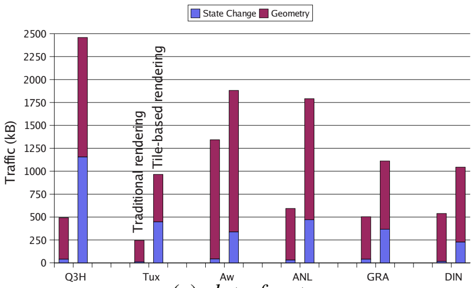
Table 1. Number of triangles transferred as a function of the tile size for each benchmark.

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

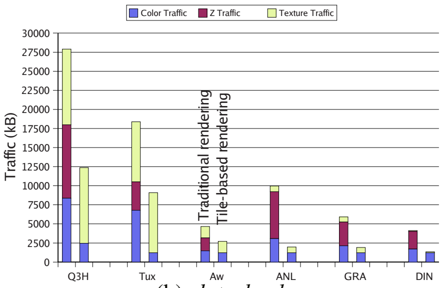
4.3 Tile-Based vs. Conventional Rendering

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

Figure 4(a) presents the amount of data traffic sent from the CPU to the rasterizer (the *data_front* component of the total traffic) for the tile-based as well as the conventional renderer. It also breaks down the *data_front* term into state change data and geometrical data. As expected, the tile-based architecture generates more *data_front* traffic than the traditional architecture. On average, using the geometric mean, the tile-based architecture increases the amount of *data_front* traffic by a factor of 2.66 compared to the conventional renderer. The figure also shows that the amount of *data_front* traffic is dominated by the geometrical data and that the increase is due for a large part to the increase in the amount of geometrical data transferred.



(a) *data_front*



(b) *data_back*

Fig. 4. The front and back data traffic components.

Figure 4(b) depicts the amount of data transferred between the rasterizer and the off-chip color and z buffers and texture memory (the *data_back* term). Furthermore, the *data_back* component has been split into data transferred from/to the color buffer, z-buffer, and texture memory. Due to the fact that our rasterizer simulator is much slower when rendering larger tile sizes due to the fact that some of the used data types, when scaled, can no longer be mapped directly to native data types, and the texture miss ratio changed only marginally for tiles with sizes from 16×16 up to 64×64, we have approximated the texture traffic for a traditional renderer with the texture traffic generated by a 64×64 tile-based rasterizer. On average, the tile-based architecture reduces the *data_back* traffic by a factor of 2.71 compared to the traditional renderer (geometric mean). Furthermore, for the conventional architecture the *data_back* traffic is dominated by the traffic between the rasterizer and the frame/z buffers, whereas in a tile-based renderer this traffic is eliminated almost completely. For a tile-based renderer, the texture traffic is the largest component of the *data_back* traffic.

Finally, Figure 5 depicts the total amount of external data traffic produced by the conventional and the tile-based renderer. The total traffic has been divided into *data_front* and *data_back* traffic. It can be seen that since the amount of *data_back* traffic is much larger than the amount of *data_front* traffic, the tile-based architecture reduces the total amount of external traffic significantly. The geometric mean of the traffic reductions over all benchmarks is a factor of 1.96. However, the advantage of tile-based rendering is workload dependent and the results show that tile-based rendering is more suitable than traditional rendering for workloads with low overlap and high overdraw, while for workloads with high overlap and low overdraw, the traditional rendering should be used. Since the workloads from our benchmark suite do not exhibit high overdraw, the results obtained for the tile-based renderer are not significantly better than traditional rendering.

5 Conclusions and Future Work

In this paper we have presented a comparison of the total amount of external data traffic required by traditional and tile-based renderers. For tile-based renderers, based on the total data traffic variation with respect to the on-chip memory (tile size), a tile size

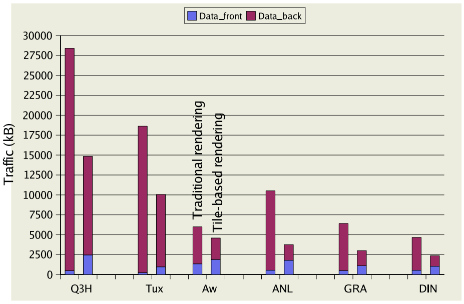


Fig. 5. Total external data transferred (kbytes) per frame for a tile-based and a traditional architecture.

of 32×32 pixels was found to yield the best trade-off between the amount of on-chip memory and the amount of external data traffic. We have also shown that tile-based rendering reduces the total amount of external traffic due to the considerable data traffic reduction between the accelerator and the off-chip memory while maintaining an acceptable increase in data traffic between the CPU and the renderer. Considering that external memory accesses consume a significant amount of power, this indicates that tile-based rendering might be a suitable technique for low-power embedded 3D graphics implementations. We mention, however, that the reduction in bandwidth of tile-based rendering when compared to traditional rendering depends significantly on the workload used. For workloads with a high overlap factor and low overdraw, the traditional rendering can still outperform the tile-based rendering, while for workloads with a low overlap factor and high overdraw, the tile-based rendering is more suitable than traditional rendering.

As future work, we intend to investigate the implications of using multitexturing for tile-based architectures. Furthermore, since the amount of texture traffic is significant for a tile-based renderer, we intend to investigate texture compression techniques. Finally, we are currently investigating the memory bandwidth required for sorting the geometrical primitives into bins corresponding to the tiles.

References

1. ARM Ltd.: ARM 3D Graphics Solutions. Available at http://www.arm.com (2002)
2. Catthoor, F., Franssen, F., Wuytack, S., Nachtergaele, L., Man, H.D.: Global Communication and Memory Optimizing Transformations for Low-Power Signal Processing Systems. In: Proc. VLSI Signal Processing Workshop. (1994)
3. Fromm, R., Perissakis, S., Cardwell, N., Kozyrakis, C., McGaughy, B., Patterson, D., Anderson, T., Yelick, K.: The Energy Efficiency of IRAM Architectures. In: Proc. 24^{th} Annual Int. Symp. on Computer Architecture, ACM Press (1997) 327–337
4. Fuchs, H., Poulton, J., Eyles, J., Greer, T., Goldfeather, J., Ellsworth, D., Molnar, S., G. Turk, B.T., Israel, L.: Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories. Computer Graphics, Vol. 23, No. 3 (July 1989) 79–88
5. Molnar, S., Cox, M., Ellsworth, D., Fuchs, H.: A Sorting Classification of Parallel Rendering. IEEE Comput. Graph. Appl. **14** (1994) 23–32 IEEE Computer Society Press.

6. Humphreys, G., Houston, M., Ng, R., Frank, R., Ahern, S., Kirchner, P.D., Klosowski, J.T.: Chromium: A Stream Processing Framework for Interactive Rendering on Clusters. In: Proc. 29th Annual Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH 2002). (2002) 693–702
7. Mueller, C.: The Sort-First Rendering Architecture for High-Performance Graphics. In: Proc. Symp. on Interactive 3D Graphics, ACM Press (1995) 75–84
8. Chen, M., Stoll, G., Igehy, H., Proudfoot, K., Hanrahan, P.: Simple Models of the Impact of Overlap in Bucket Rendering. In: Proc. ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, Lisbon, Portugal, ACM Press (1998) 105–112
9. PowerVR: 3D Graphical Processing (Tile Based Rendering - The Future of 3D), White Paper.
http://www.beyond3d.com/reviews/videologic/vivid/PowerVR_WhitePaper.pdf (2000)
10. Hsieh, E., Pentkovski, V., Piazza, T.: ZR: A 3D API Transparent Technology for Chunk Rendering. In: Proc. 34th ACM/IEEE Int. Symp. on Microarchitecture MICRO-34. (2001)
11. Cox, M., Bhandari, N.: Architectural Implications of Hardware-Accelerated Bucket Rendering on the PC. In: Proc. 1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware, ACM Press (1997) 25–34
12. Antochi, I., Juurlink, B., Cilio, A., Liuha, P.: Trading Efficiency for Energy in a Texture Cache Architecture. In: Proc. 4th Int. Conf. on Massively Parallel Computing Systems (MPCS'02). (2002)
13. Beers, A.C., Agrawala, M., Chaddha, N.: Rendering from Compressed Textures. In: Proc. 23rd Annual Conf. on Computer Graphics and Interactive Techniques, ACM Press (1996) 373–378
14. Fenney, S.: Texture Compression Using Low-Frequency Signal Modulation. In: Proc. ACM SIGGRAPH/Eurographics Conf. on Graphics Hardware, Eurographics Association (2003) 84–91
15. Akenine-Möller, T., Ström, J.: Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones. ACM Trans. Graph. **22** (2003) 801–808
16. Antochi, I., Juurlink, B., Vassiliadis, S., Liuha, P.: GraalBench: A 3D Graphics Benchmark Suite for Mobile Phones. In: Proc. ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'04). (2004) (to appear)
17. Hakura, Z.S., Gupta, A.: The Design and Analysis of a Cache Architecture for Texture Mapping. In: Proc. 24th Annual Int. Symp. on Computer Architecture. (1997)