Selecting the Optimal Tile Size for Low-Power Tile-Based Rendering

Iosif Antochi, Ben Juurlink, and Stamatis Vassiliadis
Computer Engineering Laboratory, Delft University of Technology
P.O. Box 5031, 2600 GA Delft, The Netherlands
Phone: +31 (0)15 27 83644, Fax: +31 (0)15 27 84898
E-mail: {tkg|benj|stamatis}@ce.et.tudelft.nl

Abstract—**Power-aware graphics architectures are receiving more attention recently. In this paper we analyze rendering techniques suitable for low power devices. One technique that looks promising is *Tile Rendering*. This technique decomposes a scene into tiles and renders each tile independently. For several scenes we compute a tile size that allows most triangles to be rendered without being divided into subtriangles. It is also shown that this technique reduces the bandwidth requirements substantially. Furthermore, different methods for approximating the data traffic to and from a graphics rasterizer are presented.**

Keywords—**graphics architecture, tile based rendering.**

I. Introduction

3D graphics emerges rapidly in consumer electronics. Although lots of approaches have been proposed for PC-based or entertainment platforms, 3D graphics rendering still does not often appear in embedded systems such as PDAs, mobile phones, car navigation systems, etc. With the widespread use of raster scan displays and the increasing desire for faster interactivity, higher image complexity, and higher resolution in displayed images, several techniques have been proposed for rasterizing primitive graphical objects.

Recent graphics systems use triangle rasterization to represent computer generated graphics. In fact, most graphics applications released in the last few years depend almost completely on triangle rasterization performance. Recently, the focus of graphics performance optimization is shifting to bandwidth requirements as well as transformation and lighting computations.

One important concern for mobile devices is the power consumption, thus power-aware architectures are gaining more interest in the last years. Two important sources of power consumption are buses and caches. Since tile rendering architectures reduce the data traffic required by the graphics accelerator by using a small, on-board memory to render a scene instead of an off-chip frame buffer, they seem to be suitable for low-power devices. In this paper we describe several rendering techniques based on the tile rendering model and discuss the trade-offs that can be made to implement these rendering models in low-power graphics devices. An optimal tile size is computed based on data traffic constraints.

This paper is organized as follows. After describing the principles of graphics rendering in Section II, we present the selected rendering methods used in this study in Section III. In Section IV we estimate the amount of data traffic between the CPU and the accelerator as well as the traffic between the accelerator and the frame buffer. Experimental results are presented in Section V, and conclusions and future work are given in Section VI.

II. Background

In order to understand where the rendering process fits in a real-time 3D rendering system, we briefly describe the four major functions of the traditional graphics pipeline [3], [7]: *geometry processing*, *fragment generation*, *hidden surface removal*, and *frame buffer display*. The 3D geometries of scenes are commonly defined in terms of triangles. In the first stage, matrix transformations are applied to the triangle vertices to transform vertices through different coordinate spaces such as the object (model) space, world space, eye space, and projection space. This results, after a perspective division and a viewport transformation, in a perspective mapping of the triangles to the 2D display. The second stage consists of the fragment generation, shading, and texture mapping of fragments. Texture mapping is a technique that adds adding realism to computer generated scenes by mapping 2D images onto 3D objects. The third stage is hidden surface removal that can be accomplished, for instance, by using a z buffer algorithm, and the final stage is the display of the rendered image stored in the frame buffer. Depending on the aggressiveness of an implementation, parts of the graphics pipeline can be implemented in software and others can be mapped in hardware for better performance.

The *rasterizer* performs the second and third stages described above. The input to the rasterizer consists of triangles described by vertices and additional properties of each vertex such as color and texture coordinates. Furthermore, textures must be transferred to the rasterizer or the rasterizer will have to access the textures from the main memory.

Basically, there are two working modes for an accelerator
1. *Immediate Mode* - each primitive is sent to the rasterizer only once, and each primitive is executed immediately after it was received.
2. *Display List Mode* - primitives are stored and sent to the rasterizer as a list afterwards, not necessarily in the same order as they were received.
While the Immediate Mode seems to be faster than the Display List Mode since it is does not require buffering of the primitives, it will be shown that the Display List operation mode allows more optimizations that reduce bandwidth and power consumption on different buses to be applied.

III. Tile Rendering

This section describes the basic organizations of a traditional and tile renderer and discusses several tile rendering variants.

A traditional rasterizer is depicted in Figure 1. The basic functionality of such a rasterizer is as follows. *The Transform and Lighting* (*TnL*) *Engine* processes the geometry data at vertex level (changes in coordinates, lighting computations, etc.). After that, the processed vertices are sent to the rasterizer as primitives such as points, lines, and triangles. The rasterizer scan-converts each triangle into fragments (pixels). It also performs one or more read operations from the texture buffer (memory) if texturing is enabled. Finally, for each pixel it is determined if the pixel is obscured by another pixel using, for instance, a z (depth) buffer algorithm. One access to the z buffer is needed to retrieve the old z value and, eventually, a z buffer write operation is executed if the pixel is not occluded by the previous written pixels. Also a write operation to the frame buffer is performed if the z test succeeded.

Considering that current accelerators are bandwidth limited and current rendered scenes contain a significant amount of pixels rendered but not visible in the final scene (i.e., there is *overdraw*), one possible modification to the above architecture is to use a tiled architecture. This technique decomposes a scene into smaller regions called *tiles* and renders each tile independently. The advantage of this approach is that it allows a smaller, faster memory to be integrated closer to the rasterizer for storing z values and color components of one tile so that the accesses to the frame and z buffer are local, on-chip, accesses. Only after all the primitives of a tile have been rendered, the tile content, color components, and depth values, are sent to the frame buffer.

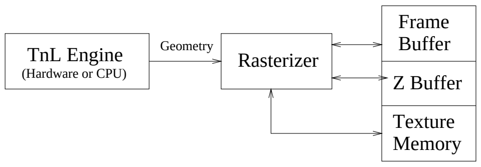

Fig. 1. Traditional Renderer.

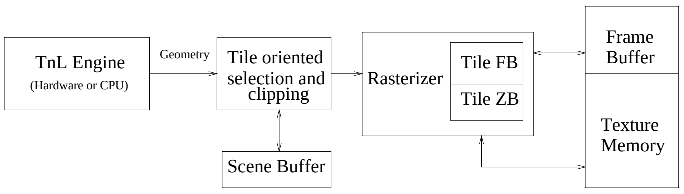

Fig. 2. Tile-Based Renderer.

The basic organization of a tile renderer is depicted in Figure 2. Since in this architecture only parts of the scene are rendered, only the triangles that are (completely or partially) contained within the considered tile need to be sent to the rasterizer. With tile rendering, a triangle might cover more than one tile, so we first need to store all primitives in a so called *scene buffer* and sort them into bins that correspond to the tiles. A tile renderer is, therefore, based on the display list model.

A. Tile Rendering Variants

Tile rendering introduces one additional step in the rendering pipeline that sorts the primitives into bins that correspond to the tiles. Since a primitive can can cover more than one tile, it can be placed in several bins. A basic tile renderer operates as follows. First, it receives the primitives to be rendered. The primitives must be sorted according to their tile repartition. After that, the data associated with each tile is fetched from the external frame buffer. A list of primitives that are relevant to the respective tile is also sent to the rasterizer. After all primitives for the tile have been rendered the content of the tile is sent to the external frame buffer.

An alternative of the above scheme is to perform triangle clipping in software so that triangles are clipped at tile boundaries. The advantage of such an architecture is that the triangle set-up time on the rasterizer can be faster, but the disadvantage is that the triangles fragments are not necessarily triangles, so they should either be splitted into more primitives thereby increasing the amount of data traffic or the rasterizer should be able to render other shapes besides triangles.

These methods, however, have the problem that they render each primitive even though maybe none of the pixels generated for the respective primitive are visible at the end. One possibility to eliminate this problem is to use *multistep rendering* [8]. First, a partial rendering is performed to determine the visible pixels. This requires a supplementary buffer to store a primitive id for the last visible pixels. Thereafter, only the primitives that have visible pixels are fully rendered.

If there is enough memory on-chip to store sufficient information for each pixel so that it can be generated later-on without resending the primitive, then less traffic between the CPU or main memory and the rasterizer can be achieved. In this paper we consider only the basic tile rendering model.

IV. Data Traffic

Some papers that discuss tile rendering (e.g,. [2]), are mainly concerned by the *overlap* of triangles with respect to tile size. Overlap is defined as the number of tiles that a primitive covers. If a primitive covers n tiles then it needs to be sent to the accelerator n times. So only the traffic between the CPU or main memory to the accelerator was considered. To obtain a more accurate estimate of power consumption, 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. In this section we estimate the total amount of data traffic.

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. Geometrical data needed to describe a primitive. In case of a triangle this data consists of triplets, each of them containing:
- coordinates of one point (x, y, z, w),
- one or two color components (r, g, b, a) and, eventually, specular color $(r1, g1, b1)$, and
- the texture coordinates (u, v).

2. Data needed to render a specific primitive:
- texture data (image arrays of colors),
- z buffer data (integer),
- frame buffer pixel color (r, g, b, a) (used for blending), and
- other supplementary data such as stencil values, accumulation values, etc.

Usually, the geometrical data is transferred from the CPU or main memory to the accelerator, while the data needed to render a primitive is mostly transferred between the accelerator and the dedicated graphics memory.

As an example, suppose we have to render t triangles, each of them having the following properties:
- Each coordinate is represented as a 16-bit integer.
- Each color component is 8 bits.
- No specular color is present.
- Each texture is mip-mapped and the initial level is 256 elements wide and 256 elements high. Multiplying this with the element width of 16 bits we obtain that the size of the initial level is $256 \times 256 \times 16$ = 1Mb. Since every next level is a quarter of the size of the level below, the total texture size of all levels is:

$$\sum_{i=0}^{\log_2 256} (\frac{1}{4})^i \times 1\text{Mb} \approx \frac{4}{3} \times 1\text{Mb} \approx 170\text{kB}.$$

Therefore, the total amount of geometry data *geom_data*(t) to transfer from the processor to the graphics chip is

$$\textit{geom\_data}(t) = t \times \textit{tri\_data}, \quad (1)$$

where,

$$\begin{aligned} \textit{tri\_data} &= Vr \times (V \times Cs + C \times CCs + T \times Ts) \\ &= 3 \times (4 \times 2 + 4 \times 1 + 2 \times 2) = 48 \text{ B}, \quad (2) \end{aligned}$$

tri_data	=	number of bytes required to render a triangle
Vr	=	number of vertices
V	=	number of coordinates per vertex
Cs	=	coordinate size (bytes)
C	=	number of color components
CCs	=	color component size (bytes)
T	=	number of texture coordinates
Ts	=	texture coordinate size (bytes)

Thus, there is a linear relation between the number of triangles and the amount of geometric data traffic between the CPU and the accelerator.

For texture data, the situation is different, however. Due to the fact that a texture can be reused inside a frame or across frames, most of the time textures are brought to the dedicated graphics memory from which they can be accessed faster than from the system memory, so actually the texture data traffic is growing slower than linear in the number primitives to be rendered. The total amount of traffic transferred from the processor or main memory to the graphics chips is:

$$\begin{aligned} \textit{data\_front} &= \textit{geom\_data}(t) + \textit{texture\_data} + \\ &\quad \textit{state\_change}, \quad (3) \end{aligned}$$

where the *state_change* term comprises the updates to the state of the rasterizer (e.g. enable/disable depth test, change texture wrapping modes, etc). The amount of *state_change* data is usually negligible compared with the geometry data and the texture data.

The total amount of data transferred to and from the accelerator is:

$$\textit{total\_data} = \textit{data\_front} + \textit{data\_back}, \tag{4}$$

where the *data_back* term of the equation accounts for the data traffic between the accelerator and the off-chip graphics memory and it can be estimated separately for a traditional renderer and a tile-based renderer. In all situations we assume that the texture data is cached due to its high locality properties.

For a frame rendered on a traditional renderer:

$$\begin{aligned}\textit{data\_back} &= Ov \times (Bc + 2 \times Bz + \\ &\quad TMR \times TPP \times Btc) \times W \times H. \end{aligned} \tag{5}$$

where,

Ov	=	the overdraw
Bc	=	number of bytes per color
2	=	accounts for one z read and one z write
Bz	=	bytes per z component
TMR	=	texture miss ratio
TPP	=	texels per pixel
Btc	=	bytes per texture element (color)
W	=	image width
H	=	image height

For a Tile-Based Renderer we have:

$$\begin{aligned}\textit{data\_back} &= (Ov \times TMR \times TPP \times Btc + Bc) \\ &\times W \times H, \end{aligned} \tag{6}$$

since we have no z data traffic, and the color information is written only once from the accelerator (tile buffer) to the off-chip memory.

A data traffic comparison between a traditional renderer and a tile-based renderer will be presented in Section V-A.

V. Results

We used as a benchmark a widespread game, namely "Quake III Arena"(Q3). This game is almost a standard in benchmarking 3D (OpenGL [9]) performance of a graphics accelerator. Even though this benchmark is no longer the most stressing application for current (PC) accelerators, it can be used as a reference for a low-power accelerator. We also used, AWadvs-04 that belongs to the *viewperf* graphics benchmark suite [6].Depending on the benchmark configuration (i.e., realism level), different results can be obtained. For instance, an accelerator that supports *multitexturing* can reduce by about half or more the number of triangles received from the host processor and also the fragments sent to the frame buffer are reduced with the same factor, due to the internal blending of textures. On the other hand, supporting multitexturing increases the computational complexity of the rasterizer since when more than one texture may need to be accessed and combined for each pixel. This approach has the advantage that it reduces the bandwidth from the rasterizer to the frame buffer, but on the other hand it might affect the locality of data in a texture cache [5], [1] used for accelerating the accesses to the texture memory.

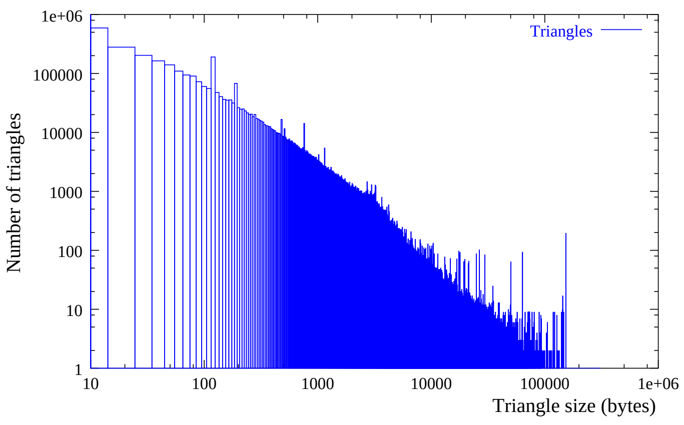

Fig. 3. Triangles histogram. This figure shows for each triangle size, the number of triangles having that size.

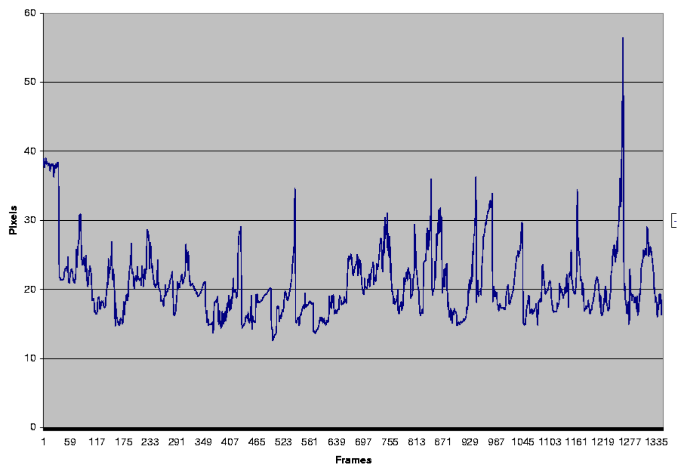

Fig. 4. Average height and width of a triangle (computed from average area).

A. Optimal Tile Size

An optimal tile size can be determined from the average triangle size, so that most triangles can be rendered without being divided into sub-triangles. Given P triangles of average pixel area A, a total of $P \times A$ pixels must be accessed to render the scene. Assuming the scene covers a

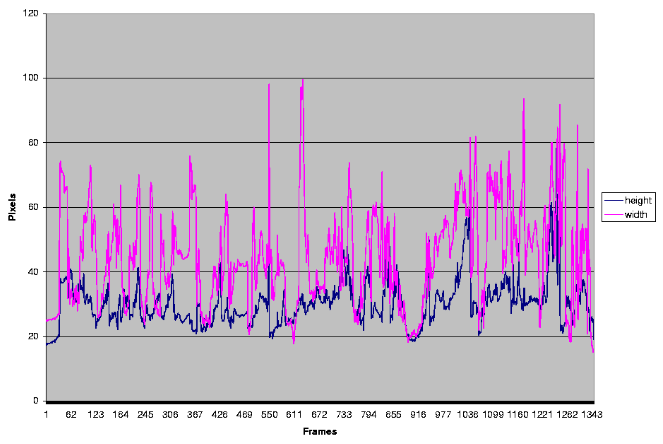

Fig. 5. Average separate height and width of a triangle.

screen of N pixels and that each triangles are overlaid in D layers everywhere, we have [4]:

$$P \times A = N \times D \tag{7}$$

where D is the depth complexity. The average height or width of a triangle is proportional to

$$\mathit{Average\_width} = \mathit{Average\_height} = \sqrt{A} = \sqrt{\frac{N \times D}{P}} \tag{8}$$

Figure 4 depicts the average triangle widths and heights, computed using Equation (8), for the frames we have studied. Some similar results are presented in [5].

On the other hand, if a bounding box for each triangle is used then the results are more accurate since the ratio of the bounding box width and height is taken into account instead of assuming equal width and height. Figure 5 presents the average height and width computed separately. Table I presents the percentages of triangles that fit into a given tile size, for different tile sizes. If we consider the ratio of the increase in tile size and the percentage of triangles covered for that tile size, the the optimum tile size is 32×32.

Tile	Tile Width			
Height	16	32	48	64
16	52.31	59.1	61.46	62.81
32	60.75	70.05	73.71	75.79
48	62.65	73.01	77.42	79.96
64	63.81	74.68	79.48	82.39

TABLE I
PERCENTAGES OF TRIANGLES THAT FIT IN A GIVEN TILE SIZE (PIXELS).

Still this is only an estimation of the optimal tile size, since some triangles even though are smaller than a tile can cover several tiles. A more relevant result is obtained by computing the actual number of triangles transferred for each tile. In Table II and Table III the number of triangles transferred from the CPU (or main memory) to the rasterizer is depicted for Q3 and AWadvs-04, respectively. The initial number of triangles to be rendered was 6,037,311 for Q3 and 14,020,076 for AWadvs-04. The number frames was 1346 for the Q3 benchmark, and 600 for the AWadvs-04 benchmark. Furthermore, the resolution used for rendering was 640×480 pixels for Q3 and 1260×938 pixels (the default) for AWadvs-04. Comparing the result from Tables II and III again for ratio of the number of triangles that are sent to a tile and the tile size, a tile size of 32×32 is optimal.

Tile	Tile Width			
Height	16	32	48	64
16	32,815	23,393	19,776	18,589
32	24,638	16,673	13,896	13,178
48	21,629	14,767	12,243	11,586
64	20,118	13,643	11,256	10,630

TABLE II
NUMBER OF KILOTRIANGLES TRANSFERRED AS A FUNCTION OF TILE SIZE (PIXELS), FOR Q3.

Tile	Tile Width (Pixels)			
Height	16	32	48	64
16	22,372	20,372	19,731	19,445
32	20,877	18,502	17,696	17,386
48	20,510	17,959	17,077	16,726
64	20,363	17,705	16,783	16,403

TABLE III
NUMBER OF KILOTRIANGLES TRANSFERRED AS A FUNCTION OF TILE SIZE (PIXELS), FOR AWADVS-04.

It can be expected that the results obtained for AWadvs-04 are better than those obtained for the Q3, since AWadvs-04 is a high-end benchmark that uses, by default, a higher resolution than Q3. It also has a larger percentage of small triangles. If we would have scaled the resolution of AWadvs-04 to 640×480, since we are concerned with low-power devices with small consoles, then the results obtained for AWadvs-04 would have been even better.

The second part of the data traffic is the traffic between the accelerator and the local graphics memory that is not on-chip. For the traditional architecture this traffic is directly proportional to the number of rendered pixels. For a tile based architecture this traffic is almost independent of the amount of rendered pixels since it involves only the transfer of tile data to the frame buffer and in rare cases from the frame buffer to the tile buffer.

The amount of data traffic between the accelerator and the off-chip graphics memory for one frame, can be determined analytically using the overdraw factor and the texture miss ratio. For Q3 we determined that without multitexturing enabled the overdraw is about 5.7 and a texture miss ratio of 10% even when using a small [1] (256B) texture cache. For the traditional approach, the data transferred between the accelerator and the off-chip memory can be approximated using Equation (5). So if the image size is 640×480, the amount of data is $5.7 \times (4 + (1 + 1) \times 4 + 0.1 \times 3 \times 4) \times 640 \times 480 \approx$ 22MB. For the tile-based renderer, using Equation (6), only $(4 + 5.7 \times 0.1 \times 3 \times 4) \times 640 \times 480 \approx 3.17$MB. So the data traffic from the accelerator to the off-chip memory in case of the tile-based rendering is reduced approximately by a factor of 6.94. On the other hand the data transferred from the CPU or main memory to the accelerator can be computed using Equation (1). For the traditional renderer : $6,037,311 \div 1346 \times 48 = 210.2$kB, and for a tile-based renderer with a tile size of 32×32 pixels: $16,672,896 \div 1346 \times 48 = 580.6$kB.

Thus at the cost of increasing the data traffic on the bus from the main memory to the accelerator, a larger data traffic is saved on the bus that connects the accelerator with the local memory.

VI. Conclusions

An overview of the current used rasterizing methods has been presented. A tile size of 32×32 pixels, considering actual applications complexity and realistic level, appears to be optimal. We have also shown that tile-based rendering reduces considerably the amount of traffic between the accelerator and the off-chip memory while the increase in data traffic from the CPU or main memory to the accelerator is only by a factor of 2.76. Considering that current rasterizers are more bandwidth limited than computational speed, we believe that tile rendering is a suitable technique for low- power devices since it provides faster on-chip access to the tiled frame and z buffers. The tile rendering method also provides an elegant solution to the ever increasing depth complexity of the rendered scenes.

References

[1] Iosif Antochi, Ben Juurlink, Andrea Cilio, and Petri Liuha. Trading Efficiency for Energy in a Texture Cache Architecture. In *Proceedings of the 4th International Conference on Massively Parallel Computing Systems (to appear)*, 2002.

[2] Michael Cox and Narendra Bhandari. Architectural implications of hardware-accelerated bucket rendering on the PC. In *Proceedings of the 1997 SIGGRAPH/Eurographics workshop on Graphics hardware*, pages 25–34. ACM Press, 1997.

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

[4] N. Gharachorloo, S. Gupta, R. F. Sproull, and I. E. Sutherland. A Characterization of Ten Rasterization Techniques. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, pages 355–368, 1989.

[5] Ziyad S. Hakura and Anoop Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. In 24^{th} *Annual International Symposium on Computer Architecture*, 1997.

[6] Tulika Mitra and Tzi-Cker Chiueh. Dynamic 3D Graphics Workload Characterization and the Architectural Implications. MICRO-32. Proceedings. 32nd Annual International Symposium on Microarchitecture, 1999.

[7] Tulika Mitra and Tzi-Cker Chiueh. Three-Dimensional Computer Graphics Architecture. Computer Science Dept., State University of New York at Stony Brook, 2000.

[8] PowerVR. 3D Graphical Processing (Tile Based Rendering - The Future of 3D), White Paper. http://www.powervr.com/Downloads.asp, 2000.

[9] M. Segal and K. Akeley. The OpenGL TM Graphics System: A Specification. Silicon Graphics, 2001.