Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones

Tomas Akenine-Möller
Chalmers University of Technology
Ericsson Mobile Platforms

Jacob Ström
Ericsson Research

Abstract

The mobile phone is one of the most widespread devices with rendering capabilities. Those capabilities have been very limited because the resources on such devices are extremely scarce; small amounts of memory, little bandwidth, little chip area dedicated for special purposes, and limited power consumption. The small display resolutions present a further challenge; the angle subtended by a pixel is relatively large, and therefore reasonably high quality rendering is needed to generate high fidelity images.

To increase the mobile rendering capabilities, we propose a new hardware architecture for rasterizing textured triangles. Our architecture focuses on saving memory bandwidth, since an external memory access typically is one of the most energy-consuming operations, and because mobile phones need to use as little power as possible. Therefore, our system includes three new key innovations: I) an inexpensive multisampling scheme that gives relatively high quality at the same cost of previous inexpensive schemes, II) a texture minification system, including texture compression, which gives quality relatively close to trilinear mipmapping at the cost of 1.33 32-bit memory accesses on average, III) a scanline-based culling scheme that avoids a significant amount of z-buffer reads, and that only requires one context. Software simulations show that these three innovations together significantly reduce the memory bandwidth, and thus also the power consumption.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processors I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Texture

Keywords: graphics hardware, mobile phone, culling, texture filtering, multisampling

1 Introduction

Mobile phones are used all over the world, and since they are equipped with displays, it is also possible to render images on these devices. It is very likely that this makes the mobile phone the most widespread rendering platform today. However, this type of rendering has mostly been limited to very simple two-dimensional graphics, and it is only recently that three-dimensional graphics has seen the light in this context. Increased interest in mobile graphics can be seen in the activities of upcoming standards, such as Java Specification Request 184, and OpenGL ES (for embedded systems). Applications that are likely to use three-dimensional graphics include man-machine interfaces (MMIs), screen savers, maps, animated messages, and, naturally, games.

Mobile phones inherently exhibit two characteristics that are drastically different from, for example, PC systems with graphics cards. First of all, they have very small displays, and second, they have very small amounts of resources for rendering. These characteristics will be briefly discussed next. Existing mobile phones with color displays have limited resolution. Common sizes are QCIF (176×144), and QVGA (320×240). Larger resolutions are more expensive than small ones, and also consume more energy which decreases use time on battery charge. It is therefore likely that lower resolutions, such as QCIF and QVGA, will dominate for all but the high end market. In terms of the number of colors of the display, anything from 256 to 65,536 is common. In addition to low resolution, the user often holds the display close to the eyes, which makes the average eye-to-pixel angle large in comparison to that of a PC system. Our measurements show that this angle is between 2–4 times larger on a mobile phone than on a PC. These display conditions implies that every pixel on a mobile phone should ultimately be rendered with higher quality than on a PC system.

There are several reasons for a mobile phone to have limited resources. Since they are powered by batteries, any type of rendering needs to use as little energy as possible. External memory accesses are often the operation in a computer system that uses the most energy [Fromm et al. 1997]. In low-power processes, an off-chip memory access consumes more than an order of magnitude more energy than an access to a small on-chip SRAM memory. This means that bandwidth resources should be used with great care, and that peak bandwidth is extremely limited in the first place. Little available bandwidth, little amount of chip area, and small amounts of memory all help in keeping the price per device low, and also, in the majority of the cases, contribute to using less energy than a system with more resources. Typical examples of real mobile phone data is: 1) one 32-bit memory access per clock cycle (10-100MHz), 2) CPUs with 10-200 MHz, and 3) 1-16 MB of reasonably fast memory.

The demand for high-quality rendering and the requirement of using little resources are contradictory. To ameliorate this conflict, we present a hardware architecture for rasterizing textured triangles with three new, key innovations:

1. a multisampling scheme that only generates two new samples per pixel, and that achieves more effective levels of shades for critical edges than previous schemes,
2. texture filtering, using texture compression, which costs 1.33 32-bit memory accesses on average for minification, with quality relatively close to trilinear mipmapping, and
3. a simple culling scheme that operates on a scanline basis, making the implementation inexpensive, and saving bandwidth and thus also energy.

Figure 1: One frame from our ScreenSaver benchmark rendered with different schemes at 176×144 pixels. Bandwidth without clears in MB/frame in brackets. OSSP=one sample per pixel. Left to right: nearest neighbor texture sampling with OSSP (0.43), POOMA with z_{min}-culling with OSSP (0.42), POOMA with FLIPQUAD multisampling and z_{min}-culling (0.62), and trilinear mipmapping with OSSP (0.91).

We have simulated our architecture in software and counted the number of memory accesses, which we believe is the most important performance measure for our target platform, the mobile phone. Our methods are ad-hoc rather than based on theoretical insights. Still, compared to trilinear mipmapping with one sample per pixel, our architecture gives relatively high quality texture filtering at 53% less memory bandwidth, and multisampling at 32% less bandwidth. Figure 1 contains some results.

This paper is organized as follows. First, some previous work is reviewed, followed by a discussion about the system assumptions. Then the actual architecture is presented, with focus on the three previously mentioned innovations. After that, the implementation is discussed, followed by results and evaluation. Finally, a conclusion and some thoughts on future work are offered.

2 Previous Work

In general, there has not been much published on low-cost architectures where the entire system is described. Two notable exceptions are Neon [McCormack et al. 1999], and the PixelVision architecture [Kelleher 1998]. However, these are not on the extreme side of the cost spectrum. The Kyro architecture described in [Akenine-Möller and Haines 2002], on the other hand, has been scaled down and adapted to handheld devices. In this form it is called MBX. This architecture is tile-based, and uses deferred shading. A significant portion of memory bandwidth is used since geometry sorting is needed before rendering can start. It has not been documented on how energy-efficient this architecture is. A low power rendering engine is presented by Woo et al. [2002], where several hardware techniques are used to minimize power usage. Their solution delivers high performance, and uses a lot of resources. Therefore, it cannot be considered to be of low cost, as opposed to our proposed system. In the rest of this section, we review research that has focused on extreme low-cost, or that closely relates to our own work.

Low Memory Bandwidth Texturing

There are mainly two ways described in the literature for reducing the cost of texturing, namely, texture compression and texture caching. A simple texture compression scheme has been incorporated in DirectX [McCabe and Brothers 1998]. This was developed by S3, and initially called S3TC, and later renamed to DXTC. A 4×4 block of texels is compressed into a fixed size using the following strategy: First, two reference colors, encoded in 16 bits each, are found, and in between these, two more colors are computed by linear interpolation. Each texel in the block can then index into these four colors, and requires thus only two bits each. This gives a compression ratio of 6:1. For blocks where the colors in the block are not close to lying on a line in RGB space, the quality may degrade severely. However, for photographs and similar images, it most often gives surprisingly good results. One of the main advantages is that each block is compressed to a fixed size, which simplifies hardware implementation greatly.

Using a cache for texels is a clever strategy since already fetched texels that are needed later can be accessed at a greatly reduced cost [Hakura and Gupta 1997]. Prefetching in an architecture with texture caches can be used to hide latency [Igehy et al. 1998].

Traversal and Occlusion Culling

For polygon rasterization with subpixel accuracy, one can use a modified Bresenham algorithm [Lathrop et al. 1990]. An often used alternative involves edge functions [Pineda 1988], where the region inside a triangle is described as the logical intersection of the positive half-spaces of the triangle edges. Different strategies, called traversals, can be used to find the pixels inside the triangle. Depending on how complex these are, different numbers of contexts are required during traversal. A context stores interpolation parameters for a particular pixel, and can be used to traverse to neighboring pixels using only additions. Each context costs considerable in terms of gates on-chip.

To increase coherence utilization, and for simple occlusion culling algorithms, graphics hardware often traverses the pixels that a triangle covers in a tiled fashion [Kelleher 1998; McCormack and McNamara 2000; Morein 2000]. This means that all pixels inside a tile, say an 8×8 region, are visited before moving on to another tile. Different traversal strategies are needed for this, and these cost in terms of numbers of contexts that must be stored. For example, McCormack and McNamara describe a tiled traversal algorithm that requires one more context than the corresponding non-tiled traversal. In total, they need four contexts for the tiled version.

The hierarchical z-buffer algorithm uses a z-pyramid to cull groups of geometry that are occluded by already rendered objects [Greene et al. 1993]. This algorithm is highly efficient when implemented in software, but there still does not exist a full-blown hardware implementation. Commodity graphics hardware often has a simpler form of occlusion culling, where each tile stores the maximum, z_{max}, of the z-values inside a tile [Morein 2000]. A tile can be e.g., 8×8 pixels. During traversal of a triangle, a test is performed when a new tile is visited that determines if the "smallest" z-value of the triangle is larger than z_{max} of the corresponding tile. If so, that tile is skipped, and otherwise that tile is rendered as usual. Note that, to update z_{max}, all the z-values of the tile must be read, which can be expensive. This update operation is needed if a z-value is overwritten that was equal to z_{max}.

Low-Cost Screen Antialiasing

In terms of low-cost antialiasing schemes for rasterization hardware, there is not much previous published work. Our view of low-cost schemes is that they should have reasonably regular sampling patterns and generate few samples per pixel, in order to simplify implementation and generate few memory accesses. Note that edge antialiasing is most needed for near-vertical and near-horizontal edges. In our experience, during rotation, edges near 45 degrees are the next to most distracting. In Figure 2, some common low-cost sampling patterns are illustrated. Many graphics cards support brute-force supersampling schemes, where the scene

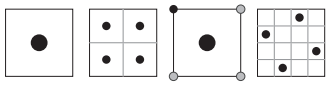

Figure 2: Simple, low-cost sampling patterns (left to right): one sample per pixel (no antialiasing), 2×2 super sampling (brute force), Quincunx sampling pattern (gray samples are borrowed from neighboring pixels), and rotated grid supersampling (RGSS).

is rendered at twice the horizontal and vertical resolution. When the scene has been rendered, a filtering pass starts, which averages 2×2 regions down to a single pixel. These schemes use four times as much memory, and this sampling pattern does not much increase the number of shades for near-horizontal and near-vertical edges. The most apparent advantage is that it is simple to implement. The *Quincunx* scheme is less expensive [NVIDIA 2001]. It generates only two samples per pixel, but uses five samples to generate a color for a pixel. The sampling pattern is that of the "five" on a dice as seen in Figure 2. Each pixel generates one sample in the middle of the pixel, and one sample in one of the corners. The other three corner samples come from neighboring pixels' corner samples. This makes the scheme inexpensive, simple to implement, and with increased quality over the brute force approach. The weights for the samples in Quincunx are 0.5 for the middle sample, and 0.125 for each of the corner samples. Another clever scheme is the N-rooks sampling scheme [Shirley 1990]. One instantiation of this is the *rotated grid supersampling* (RGSS) scheme, as shown to the right in Figure 2. The pixel is divided into a 4×4 grid, and four samples are placed so that every column and every row has exactly one sample. This increases the average number of shades on near-horizontal and near-vertical edges. Still, it uses four samples, and thus as much memory as the brute force 2×2 scheme.

3 System Assumptions

In this section we will present the assumptions that we have made for our system, and some background for those assumptions.

In order to keep power usage low, the available memory bandwidth is assumed to be 32 bits per clock. Texture accesses in graphics memory are completed in one clock cycle, while a texture access in main memory may incur a latency of more than one clock cycle. The power constraint is likely to limit display resolution to QCIF or QVGA for some years to come. Furthermore, we assume that the depth complexity per pixel will be at most four, which enables reasonably complex scenes. Colors and depth values in the z-buffer are stored in 16 bits each. Furthermore, colors are encoded as 5-6-5 bits for red, green, and blue. Next, we will discuss the implications of some of these limits on resources.

In general, the number of memory accesses, m, for a standard fragment that is visible is often assumed to be [Morein 2000]:

$$m = z_R + z_W + c_W + t_R. \tag{1}$$

z_R (Z read), z_W (Z write), and c_W (color write) are often 32 bits=4 bytes memory accesses in PC systems, while t_R (texture read) can be significantly more due to texture filtering. Using, for example, trilinear mipmapping [Williams 1983] and no texture caching, t_R can cost 3 bytes times 8 = 24 bytes. In our system, z_R, z_W, and c_W cost 2 bytes each, and a color in a texture is stored as 16 bits. The average overdraw, $o(n)$, per pixel is given by the equation below [Cox and Hanrahan 1993], where n is the depth complexity:

$$o(n) = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} \tag{2}$$

For an average pixel, this gives:

$$m = z_R + \frac{o(n)}{n}(z_W + c_W + t_R). \tag{3}$$

A target depth complexity of $n = 4$ gives $o(n) \approx 2$. Therefore, the following holds for an average fragment in our architecture:

$$m = z_R + \frac{2}{4}(z_W + c_W + t_R). \tag{4}$$

This is true if the depth test (z_R) is done before the other operations (z_W, c_W, t_R). Assume that z_R's, z_W's and c_W's can be queued up so that two values are read/written in one 32-bit memory access, and that we count in terms of 32-bit memory operations. This gives:

$$m = 0.5 + \frac{2}{4}(0.5 + 0.5 + 6) = 4. \tag{5}$$

The last figure (6), comes from the fact that texels are stored in 16 bits, which means that reading a 2×2 texel block takes either two accesses (if we are lucky, probability 0.5) or four (if unlucky, probability 0.5). Thus, the eight texels needed for trilinear mipmapping need six 32-bit accesses on average. Since the depth complexity is 4, this implies that the required bandwidth per pixel is $4 \times 4 = 16$ words of 32 bits each.

4 Architecture

Our rasterization system is mostly a classical pipeline, and so need not be described in detail. However, it includes three key innovations that will be described here. These increase image quality and/or lower memory bandwidth usage, and thus also power consumption. Our design choices are often taken in order to create an affordable solution that is also energy-efficient. At the same time, we desire high rendering quality and performance. Therefore, we seek a balanced architecture that satisfies those constraints and desires in a reasonable way.

4.1 FLIPQUAD Multisampling

As implied in Section 1, our target platform needs reasonably high rendering quality, and hence, we have decided to include some sort of multisampling scheme. This scheme should be as cheap as possible and still reduce edge antialiasing significantly. Therefore, our approach is to combine the good features of the Quincunx sampling pattern and RGSS (see Section 2) into what we call the *FLIPQUAD* multisampling scheme. The desired features of Quincunx are that it generates only two samples per pixel, and that sample sharing from neighboring pixels occurs. The advantage of RGSS is that each row and column has exactly one sample, which increases the number of shades in edge antialiasing.

We start with the RGSS sampling pattern, and move the sample locations so that they are positioned at the pixel borders. This is shown to the left in Figure 3. Placing samples on the border of

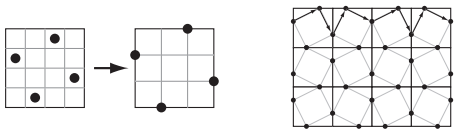

Figure 3: Left: the RGSS pattern is adapted so that sample sharing between neighbors can potentially occur. Right: the sample pattern has to be reflected when moving to neighboring pixels. This results in the FLIPQUAD sampling pattern. The top row pixels to the right also shows a possible traversal order (moving to the right).

the pixel opens up for simple sample sharing between neighboring pixels. However, the resulting pattern cannot be replicated onto every pixel on the screen, because this does not make sample sharing

occur. The simple solution to this is to flip (reflect) every other pattern. This is shown to the right in Figure 3. This sampling pattern has the desired features: only two samples generated per pixel, sample sharing occurs, and samples are unique on rows and columns. The final color of a pixel is obtained by averaging the four samples with equal weights (0.25), and this causes a texture shift towards the upper left.

For all pixels except the bottom and right edges, only two samples are generated. Assume that these two samples are the ones located on the left and the top edge of the pixel. Texture samples could either be fetched for both these sample locations, or alternatively, they can share a single texture lookup. We take the latter, less expensive approach, and we also let these samples share the same interpolated color. This technique is also used by Quincunx. However, each sample computes its own depth value in order to avoid incorrect surface intersections. To simplify hardware, the following coordinates are used for shared computations. Assume that the coordinates of the upper left corner of the pixel is $(0,0)$, and that the lower right corner is $(1,1)$. The average of the two sample locations could be used, but since the sample configuration changes for every other pixel, we use the average of the two samples for both configurations: (two for the left edge, two for the upper edge) $[(0,1/3)+(0,2/3)+(1/3,0)+(2/3,0)]/4 = (1/4,1/4)$.

4.2 The POOMA Texturing System

In this section, we will describe the texturing subsystem of our architecture. It is called "poor man's" texturing system (POOMA for short), since it makes as much as possible out of the limited resources. Our starting point is trilinear mipmapping, which accesses six 32-bit words on average (see Section 3). Using the techniques presented in this section—*bilinear-average mipmapping*, *texture compression* and *block overlapping*—this can be reduced to as low as 1.33 memory accesses on average. Note that we have omitted the use of texture caches as those incur a significant cost to our system, and also because our system only can access 32-bits per cycle, which limits the latency problem.

Bilinear-Average Mipmapping: Trilinear mipmapping requires access to the 2×2 neighborhood in two levels of the mipmap, and in each level bilinear filtering should be performed. Finally, those two filtered values are linearly blended to create the final filtered color. Our simplified filtering method uses only the texels of the higher resolution mipmap level, and so avoids half of the memory accesses: The four texels in the higher resolution level are read, and bilinearly filtered to produce the first filtered value. Then the average of the same four texels is computed, and used as the second filtered value. These two filtered values are linearly combined, to provide the final filtered color. Roughly, this gives us a filtering scheme that computes bilinear filtering in level n, and nearest-neighbor filtering in level $n+1$, and then linear filtering between those values. However, our scheme often performs a little bit better than that. The reason for this is illustrated to the right in Figure 4. Assume that the texture sample point is located in the gray 2×2 region. This region does not coincide with the 2×2 blocks used for computing averages for level $n+1$. Therefore, the average of the texels in the gray region does not exist in level $n+1$. Still, it is used in our scheme, and it is more correct than the corresponding nearest-neighbor sample. Since only the higher resolution mipmap level is read, the number of memory accesses is halved from six to three.

Texture Compression: To increase the number of texels obtained in one memory access, we use texture compression. Each 3×2 region is compressed into 32 bits in the following way. Just as in the DXTC algorithm (Section 2), two reference colors are stored, but only 11 bits are used per color (4-4-3). A third reference color is calculated as the mean of these two reference colors. Each texel

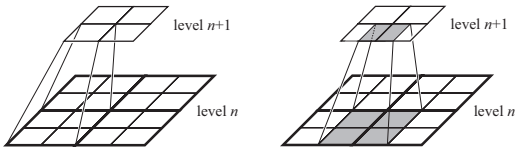

Figure 4: Left: part of a mipmap pyramid with two levels shown. A texel in level $n+1$ is often computed as some average of four texels in level n. Right: If the four gray texels constitute the 2×2 neighborhood of the sample, our filtering technique (bilinear-average mipmapping) uses the average of those as a filtered value from level $n+1$. Note, however, that this value does not exist in the real level $n+1$, and therefore our filtering technique finds a better value than doing nearest neighbor sampling in level $n+1$.

in the block can refer to one of the three reference colors. Thus a ternary variable is needed for each texel, instead of two bits as used by DXTC. Three ternary variables give rise to $3^3 = 27$ different combinations, and can thus be encoded into a 5-bit number, that we call a texel mapping value. Two such texel mapping values are sufficient to encode the full 3×2 texel block. The texel mapping value can be decoded using the following table, where *a*, *b* and *c* represent the first, second and third reference color, respectively.

0: aaa	4: abb	8: acc	12: bba	16: bcb	20: cac	24: cca	28: -
1: aab	5: abc	9: bba	13: bbb	17: bcc	21: cba	25: ccb	29: -
2: aac	6: aca	10: bab	14: bbc	18: caa	22: cbb	26: ccc	30: -
3: aba	7: acb	11: bac	15: bca	19: cab	23: cbc	27: -	31: -

For instance, if a texel mapping value of 18 is stored, the three texels should be *c*, *a*, *a*. Using this compression scheme, a 3×2-block can be stored in $11+11+5+5 = 32$ bits. Originally, a 3×2 block requires 2 bytes$\times 3\times 2 = 12$ bytes of storage. Using the proposed compression scheme, the same block requires only 4 bytes. This means that we achieve a compression ratio of 3:1. The texture compression scheme also lowers memory bandwidth. In $1/3$ of the cases, all 2×2 texels can be found in the same block, and only one memory access is needed. In half of the cases, two blocks are needed, and in $1/6$ of the cases, four blocks are needed. Thus the average number of memory accesses is lowered from three to $1/3\times 1+1/2\times 2+1/6\times 4 = 2$.

Block Overlapping: While a texture cache can decrease the number of memory accesses needed, its cost in terms of on-chip memory lead us down a different path. Instead of caching texels, we can increase the probability that the entire 2×2 neighborhood around the texture sample point can be found inside a block. This is done by introducing redundant blocks. Each 3×2 block is allowed to overlap with its neighbor in the *y*-direction, as shown in Figure 5. Thus, the entire 2×2 neighborhood will be inside a block with

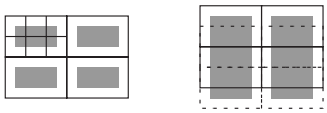

Figure 5: Left: If a texture sample point is inside the gray region, then all four texels needed for bilinear filtering are available from the same 3×2 block. Right: The dotted blocks overlap with the solid blocks, and the gray area covers more of the space.

probability $2/3$, and one memory access will suffice. In the rest of the cases, two blocks are needed, which means that the number of memory accesses has been lowered from six to $2/3\times 1+1/3\times 2$ = 1.33. One obvious disadvantage here is that the memory requirements increase. Roughly speaking, the increase will be a factor

of 2, but since the previously presented compression scheme compressed at a ratio of 3:1, our scheme will still decrease memory requirements by a factor of 2/3. A more subtle disadvantage is that the same texel will be represented twice, and since the compression is lossy, with two slightly different values. During minification this can increase flickering slightly, but as we will show, this effect is surprisingly mild. During magnification, however, this will lead to discontinuities when traversing block boundaries in the *y*-direction. Therefore, the redundant blocks are not used during magnification, and two memory accesses are thus required. The actual average number of memory access per pixel will thus vary with how great a percentage of the pixels are magnified, but will end up somewhere between 1.33 and 2.

4.3 Scanline-based z_{min}-Culling

Our goal of constructing an energy-efficient architecture can be translated into minimizing the cost of the terms in Equation 3 because memory accesses use the majority of the energy [Fromm et al. 1997]. Since c_W's and z_W's are hard to avoid, and because our POOMA texturing system reduced the cost of t_R significantly, it is natural to attempt to reduce the cost of z_R. It is interesting to note that if the depth test is performed before texturing, then z_{max}-based occlusion culling algorithms [Morein 2000] can only avoid z_R's, in terms of memory bandwidth. The following rewrite of Equation 3 is key to our algorithm:

$$m = \alpha z_R + \beta(z_R + z_W + c_W + t_R), \tag{6}$$

where $\alpha = 1 - o(n)/n$ and $\beta = o(n)/n$. If no occlusion culling algorithm is used, then α is the cost for occluded fragments in terms of z_R and β is the cost for non-occluded fragments. Thus, αz_R is what can maximally be avoided in terms of bandwidth if a z_{max}-based occlusion culling algorithm is used. For $n = 4$ we can see that $\alpha \approx \beta \approx 2/4 = 0.5$ (Section 3). When n is smaller than 4, then $\alpha < \beta$. This lead us to investigate whether it is possible to reduce the cost associated with β instead of that of α. Therefore, we propose that the minimum *z*-value, z_{min}, of each tile is stored (off chip). When a new a tile is encountered, a maximum *z*-value, z_{tri}, for the triangle in the tile is computed. This z_{tri}-value can be, e.g., the maximum of the triangle vertices' *z*-values, or the maximum *z*-value of the triangle plane inside the tile. If $z_{min} > z_{tri}$, then all fragments inside the tile for that triangle are visible, and all *z*-reads can be avoided for that tile. Put another way, triangles that are definitely in front of all previously rendered geometry do not need to read the *z*-buffer. From now on, we refer to this algorithm as z_{min}-based culling. In the same spirit, Morein uses a fast clear algorithm that also avoids the first z-read operation [2000]. There are two major advantages of z_{min}-based culling over z_{max}-based culling. First, z_{min} for a tile is trivial to update: as soon as we write a *z*-value that is smaller than the tile's current z_{min}, the z_{min} value must be updated. Thus, there is no need to read all *z*-values of the tile in order to update z_{min}. Second, for depth complexities up to $n = 4$, there is potentially more to gain by using the proposed algorithm since $\beta > \alpha$. This is because the z_{min}-based culling algorithm starts to pay off from the first rendered triangle, while for z_{max}-based algorithms, it takes a while to build up a z_{max}-value that can actually cull something. As a background, we first explain z_{min}-culling for a system that traverses pixels tile-by-tile. We will then go through how a scanline based version works. In both cases, we need to store the z_{min}-value of a tile off-chip, and when rendering of a tile starts, an off-chip memory access is always needed.

With tile-based traversal, we start rasterizing a tile by reading z_{min} from the off-chip memory. We then calculate z_{tri} for that tile. If $z_{min} > z_{tri}$, all pixels in the tile are visible, and we can render the entire tile without reading the *z*-buffer. The z_{min}-value is copied to on-chip memory, where it is called z_{min}^{onchip} and is continuously updated during pixel write so that it holds the minimum of the *z*-buffer for that tile. After the rasterization of the tile, we write back the z_{min}^{onchip}-value to the off-chip z_{min} storage for that tile.

Note that the algorithm reads and writes to the off-chip z_{min} storage even if no *z*-reads can be avoided. In the worst case, this can make the total bandwidth increase. However, as seen in the results, the reduction in bandwidth due to tiles which can be trivially accepted generally more than outweighs this.

In order to decrease the number of contexts (interpolation parameters etc), we use the zigzag traversal scheme [Pineda 1988] (left part of Figure 6), which is scanline based and only uses one context. Reading and writing the off-chip z_{min}-value eight times per tile (once for each scanline) is too expensive. For an efficient z_{min}-culling implementation for scanline traversal, we store temporary information about the current row of tiles on-chip. For each tile on the current row, we store a z_{min}^{onchip}-value, a *visited*-bit and a *visible*-bit.

When we switch from one row of tiles to the next (e.g., going from $y = 7$ to $y = 8$ for 8×8 tiles), we invalidate all *visited* bits in the on-chip memory (right part of Figure 6). This indicates that the information stored in z_{min}^{onchip} and *visible* is invalid. The first time we

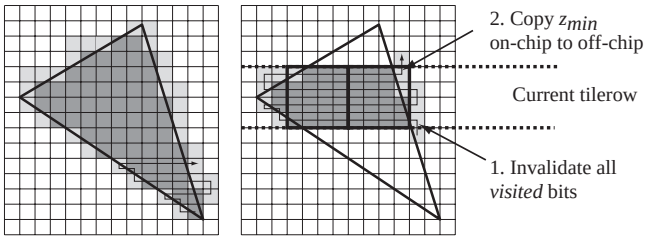

Figure 6: Left: zigzag traversal. Dark gray pixels are inside the triangle, and light gray pixels are outside the triangle, but still visited by the traversal algorithm. Right: tiled zigzag traversal.

enter a tile, we validate the *visited* bit, copy z_{min} from the off-chip memory to z_{min}^{onchip}, calculate z_{tri} and set *visible* to true if $z_{min}^{onchip} > z_{tri}$, and false otherwise. When we enter a tile for which the *visited* bit is set, we simply read *visible* to see if we need to read the *z*-buffer or not. When writing inside a tile, we update the z_{min}^{onchip}-value appropriately. Finally, before we go to the next row of tiles (going from $y = 15$ to $y = 16$ in our case), we write back all z_{min}^{onchip}-values for all the tiles where *visited* is true.

If a stencil or alpha test kills the pixel, the z_{min}^{onchip}-value for that tile will not be updated for that pixel. This is possible since updating of z_{min}^{onchip} is done right before *z*-write and *c*-write, at the end of the pixel pipeline. Note also that no hazards can occur for the updates of z_{min} since our architecture only renders one triangle at a time.

5 Implementation

We have implemented a subset of OpenGL in software, and focused mainly on the rasterization stage, since it is there our new algorithms reside. All texture filtering modes of OpenGL have been implemented, in addition to a new texture filtering mode for texture minification to support the POOMA texturing system. We also support bilinear-average filtering with and without compression. The z_{min}-culling scheme was implemented inside the traversal loop that used zigzag-order. The FLIPQUAD multisampling scheme is toggled through a simple API call as well. Due to our energy focus, we have mainly concentrated on gathering statistics concerning memory accesses. This is important since it reveals a lot about a possible hardware implementation in terms of energy-efficiency.

6 Results and Evaluation

In this section, we show visual results from our algorithms, as well as data gathered during software simulations of our architecture. For many of our tests, we have used three benchmark scenes: ScreenSaver, Man-Machine Interface (MMI), and Game. Screen

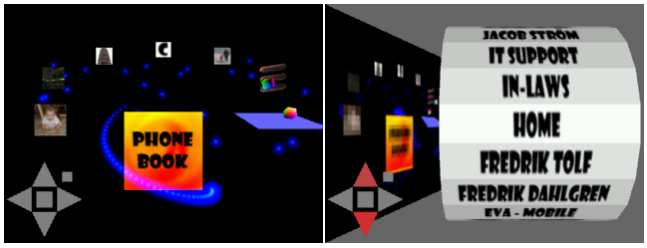

Figure 7: Screen shots from our MMI benchmark scene. To the left the main 3D user interface is shown. A number of menu icons are located on a circle, and the user can navigate using the left and right keys on the mobile phone. A blue particle system shows the currently active icon. To the right, the phone book icon has been activated and the user can scroll in his phone book.

shots from these can be seen in Figures 1, 7, and 10. ScreenSaver is a simple screen saver, and MMI is a three-dimensional user interface. Finally, Game is a walkthrough in a game level. We believe these three scenes represent typical applications that will be used on mobile phones. We have chosen not to include frame buffer clears in our statistics, since our paper does not focus on reducing these costs.

6.1 FLIPQUAD Multisampling

In this subsection, our proposed multisampling scheme will be evaluated. In Figure 8, a white triangle has been rendered on a black

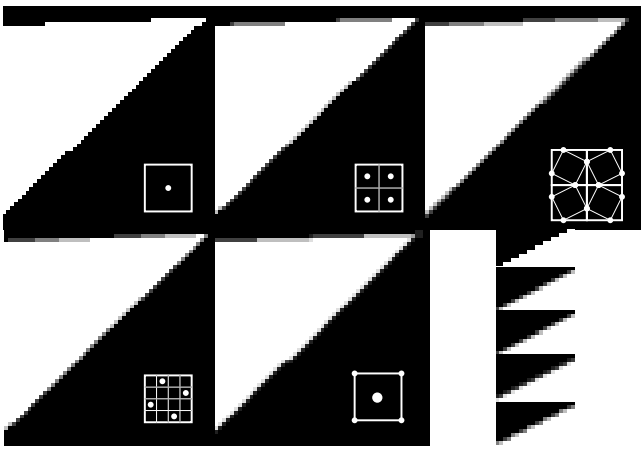

Figure 8: Different sampling schemes for a white triangle on black background. From left to right, top to bottom: One sample per pixel, 2×2 samples, FLIPQUAD (our scheme), RGSS and Quincunx. Right bottom: the worst case edge for FLIPQUAD. Top to bottom: one sample, 2×2, RGSS, Quincunx, and FLIPQUAD.

background, to show how well the schemes perform for edge antialiasing. The best scheme here is RGSS, but it costs 4 samples per pixel. FLIPQUAD performs almost as good at half the cost. Notice also, that Quincunx, with the same cost as FLIPQUAD, creates fewer effective shades on the edges, and fails to remove the jerk on the edge that is close to 45 degrees. In the bottom right part of that figure, one of the worst case edges for our scheme is evaluated. This occurs when the triangle edge coincides with two of the sample locations. As can be seen, our scheme produces a slightly jerky result, but we believe that near-horizontal, near-vertical, and near-45° are the types of edges in most need of antialiasing. Figure 9 shows an evaluation of how the schemes perform on rendering thin features. As can be seen, our scheme again performs almost as well as RGSS.

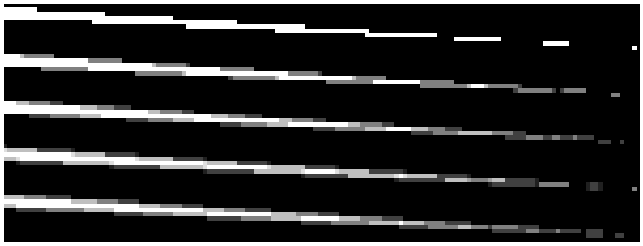

Figure 9: Evaluation of sampling schemes for thin features. From top to bottom: one sample per pixel, 2×2 samples (brute force), RGSS, Quincunx, and FLIPQUAD.

This is somewhat surprising, since the distance between the samples in FLIPQUAD is relatively large. We believe that one reason why it still performs well, is that the sampling pattern is reflected for every other pixel, and this break-up of the symmetry helps in getting a better result. Compared to Quincunx, a major advantage is that the weights for our scheme is the same for all four samples. This results in the following weight transition for FLIPQUAD: 0, 0.25, 0.5, 0.75, 1.0 (from 0 to 4 samples). For Quincunx, the corresponding transition is: 0, 0.125, 0.25, 0.75, 0.875, 1.0. As can be seen, there is one more number, but the transition is not linear. Also, for near-vertical or near-horizontal edges, the effective transition often becomes: 0, 0.25, 0.75, 1.0, which is clearly worse than that of FLIPQUAD. In Figure 10, our scheme is compared against Quincunx for a real scene. We choose to concentrate on Quincunx,

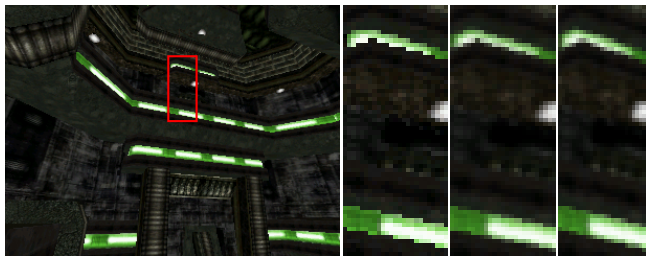

Figure 10: A screen shot from our Game benchmark scene using (left to right): one sample per pixel, Quincunx, and FLIPQUAD. In all images, we also used the POOMA texturing system.

since it has the same memory requirements as our scheme. As can be seen, FLIPQUAD reduces edge aliasing a bit more than Quincunx, and both FLIPQUAD and Quincunx produce good results in comparison to using just one sample per pixel. We have not observed any artifacts stemming from the variation in the sampling pattern, however, a more formal analysis is left for future work.

The triangle setup for using FLIPQUAD needs to compute $1/3$'s of the deltas, and then during stepping from one sample to another, one needs to step with these deltas, or two times these deltas. Also, to be certain that a pixel is completely outside a triangle, one either needs to test up to three (one for left, top, and right pixel edge) sample locations for inclusion in the triangle. Alternatively, one can test geometrically whether the square of the pixel overlaps the

triangle. To summarize, our scheme is a bit more complex to implement than the Quincunx scheme, but we believe it is worth the extra complexity because of the increase in quality.

6.2 The POOMA Texturing System

In this section, we will show individual results for each of the innovations of the POOMA texturing system; bilinear-average mipmapping, texture compression and overlapping. We will also show results from the combination of these three techniques.

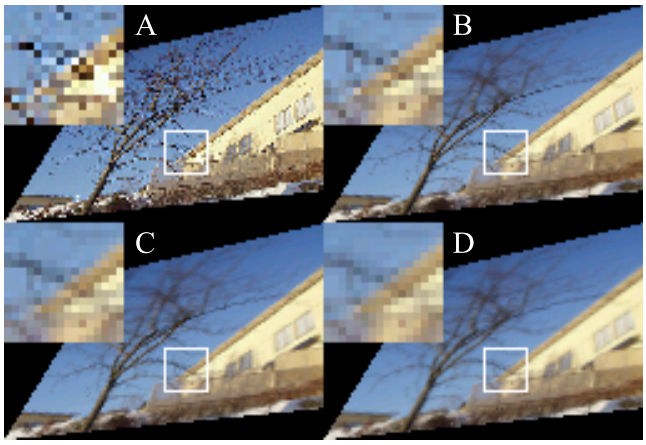

Figure 11: A: nearest neighbor, 1 memory accesses per pixel (MAPP). B: bilinear mipmapping, 3 MAPP. C: bilinear-average mipmapping, the proposed algorithm, 3 MAPP. D: trilinear mipmapping, 6 MAPP.

Bilinear-Average Mipmapping: Figure 11 shows a 16-bit image of a tree and a house textured using nearest neighbor (1 memory access, top left), bilinear mipmapping (3 memory accesses, top right), bilinear-average texturing as proposed here (3 memory accesses, bottom left) and trilinear mipmapping (6 memory accesses, bottom right). As we can see, bilinear-average mipmapping produces less aliasing than bilinear mipmapping, at the same cost in terms of memory accesses—half of that of trilinear mipmapping. Since the proposed scheme does not interpolate between mipmap levels, it cannot be used for, e.g., art maps for non-photorealistic rendering [Klein et al. 2001], and other similar techniques. This also implies that each level in the mipmap must be a lowpass filtered version of the previous level. When the level of detail is increased, a filtering transition will occur from bilinear interpolation in the higher resolution level to something that is similar to nearest neighbor mipmapping in the lower resolution level. Here, underblurring can create a slight aliasing effect, as in the checker board image in Figure 13. However, this effect is less pronounced than for bilinear mipmapping. It is our experience that the bilinear-average texturing proposed here is better than all other schemes in this study bar trilinear mipmapping, which is considerably more costly.

Texture Compression: Figure 12 shows some results from our texture compression scheme. The top row shows the uncompressed 100×100 images in 16 bits (5-6-5). The middle row shows the same images, but this time compressed with the proposed scheme, to $1/3$ of the number of bytes. In these two images, the 2×3 blocks are rotated 90° compared to the illustration in Figure 5. The bottom row shows the images compressed using the S3TC method. When starting from 24 bits, the S3TC method compresses a factor 6:1, but since we start out with 16 bit originals, the compression factor is 4:1. It is clear that the images compressed with the proposed technique have some artifacts, due to the quantization to 11 bits (4-4-3), and to the fact that only three colors can be represented in each

Figure 12: Top: Uncompressed 100×100. Middle: Compressed 3:1 using the proposed scheme, PSNR 24.4 dB and 28.3 dB. Bottom: Compressed 4:1 using S3TC, PSNR 23.7 dB and 31.3 dB.

block. The compression can also "disturb" the dithering pattern that is needed due to the 11 bit quantization. This is especially a problem for smooth gradients of blue (as seen in the blue sky of Figure 12), since only 3 bits are allotted to the blue component. S3TC on the other hand gives smoother colors, but also more block artifacts as can be seen in the left image.

Block Overlapping: By using block overlapping, it is possible to go down from 2 to 1.33 memory accesses per pixel for minification. As discussed in Section 4.2, this means that there will be two different representations of the texture. The two leftmost images in Figure 13 show two such representations. This will give rise to some flickering, since a pixel can be drawn using the one represen-

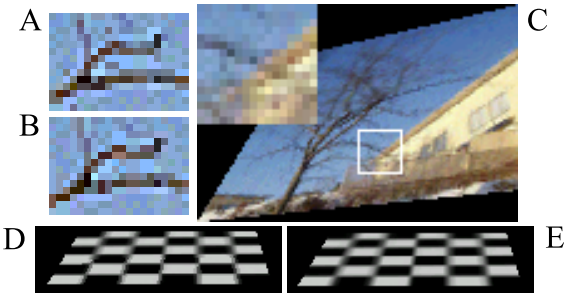

Figure 13: A, B: Two overlapping representations of a texture. C: The combined results of bilinear-average mipmapping, texture compression and block overlapping. D: Checker pattern, bilinear-average mipmapping. E: Checker pattern, trilinear mipmapping.

tation in one frame, and the other representation in the next frame. However, during minification this flickering effect is reduced by the bilinear filtering that is part of the bilinear-average texturing scheme. Our experience is that bilinear-average texturing and block overlapping together produce slightly more flickering/aliasing than just bilinear-average texturing by itself, but the result is still much

better than, for instance, nearest neighbor texturing, which requires only slightly fewer memory accesses. Also, even with block overlapping and mipmapping, our scheme requires less memory storage than nearest neighbor without mipmapping. The rightmost image in Figure 13 shows the combination of bilinear-average texturing, texture compression and block overlapping.

The POOMA texturing system may not be appropriate in all situations. For render-to-textures, for instance, compressed textures might not be feasible due to the extra time needed for compression. Bilinear-average filtering, however, might still be appropriate. Other times, standard trilinear mipmapping is the way to go. We see the features of POOMA texturing as a complement to existing filtering methods, rather than as a replacement.

6.3 Scanline-based z_{min}-Culling

To evaluate our culling algorithm we used the three scenes presented in the beginning of this section. The table below shows how much z_{min}-culling, with tile size 8×8, reduced the number of z-reads, and how much the total bandwidth (BW), i.e., external memory accesses, was lowered. The second column shows the average depth complexity, while the last shows increase due to z_{min}-culling.

	num. of frames	*avg. DC*	*reduction of z-reads*	*tot. BW change*	*on-chip BW*
ScreenSaver	1599	2.5	69 %	-10 %	+6 %
MMI	1807	0.65	84 %	-14 %	+24 %
Game	2666	1.5	49 %	-8 %	+28 %

As can be seen, the reduction of z-reads is quite drastic, ranging from 49 up to 84 percent. In terms of total bandwidth, z_{min}-culling offers reduction between 8–14 percent. Note that off-chip memory accesses are more than an order of magnitude more expensive in power consumption than an on-chip access, and therefore our scheme is advantageous. The on-chip requirements are $176/8 \times 18$ bits $= 49.5$ bytes for QCIF, and 90 bytes for QVGA, where 18 bits comes from the fact that 2 bytes are used to store z_{min}, and 2 bits are needed for the *visited* and *visible* flags. We believe that these modest requirements and the gains in bandwidth are enough to justify implementation of this algorithm.

6.4 Entire Architecture

To evaluate our entire architecture, our three benchmark scenes were used. The table below reports bandwidth usage to external memory in megabytes (MB). The scenes were rendered in QVGA resolution.

	nearest 1 sample	*POOMA+z_{min} 1 sample*	*POOMA+z_{min} FLIPQUAD*	*trilinear 1 sample*
ScreenSaver	1.15	1.29	1.87	2.24
MMI	0.52	0.51	0.70	1.10
Game	0.98	1.09	1.58	2.80

Comparing the first two columns, we see that POOMA with z_{min}-culling is quite near the results of nearest-neighbor sampling. This is remarkable, since the quality increase of our method is substantial. Texture storage is also saved: with overlapping and mipmap levels it is about $8/9$th of that needed for a non-mipmapped uncompressed texture. Compared to trilinear mipmapping, POOMA with z_{min}-culling is considerably cheaper — a reduction of 53% on average. Perhaps even more striking is a comparison of the two last columns. It reveals that our scheme can provide nice texture filtering with multisampling at a lower cost (32% lower on average) than single sample trilinear mipmapping. Figure 1 shows visual results.

7 Conclusion and Future Work

We have proposed an architecture for hardware rasterizing of textured triangles on mobile phones. The architechture has been implemented in software. One of our goals was to reduce the number of memory accesses since these consume much of the total energy of the system, and our texturing system and culling can reduce the used memory bandwidth by 53% with similar quality. Another goal was to keep the rendering quality high, and therefore we have proposed a new, inexpensive multisampling scheme. Together with our texturing system and culling, it provides multisampling at 32% fewer memory accesses than single sample trilinear mipmapping. We believe that the tradeoffs that we have made can be justified in our context, i.e., rendering on a mobile phone powered by a rechargeable battery. In the future, we would like to combine our POOMA texturing system with texture caches for even lower memory bandwidth consumption. Furthermore, we would like to combine our z_{min}-culling with the z_{max}-culling algorithm.

Acknowledgements: Thanks to Peter Svedberg and Peter Gomez for help with the video. Many thanks to Eric Haines, Fredrik Dahlgren and Erik Ledfelt, for proofreading. Thanks to `www.gametutorials.com` for letting us use the scene in Figure 10.

References

AKENINE-MÖLLER, T., AND HAINES, E. 2002. *Real-Time Rendering*. AK Peters Ltd.

COX, M., AND HANRAHAN, P. 1993. Pixel Merging for Object-Parallel Rendering: a Distributed Snooping Algorithm. In *Symposium on Parallel Rendering*, ACM SIGGRAPH, 49–56.

FROMM, R., PERISSAKIS, S., CARDWELL, N., KOZYRAKIS, C., MCCAUGHY, B., PATTERSON, D., ANDERSON, T., AND YELICK, K. 1997. The Energy Efficiency of IRAM Architectures. In *24th Annual International Symposium on Computer Arhchitecture*, ACM/IEEE, 327–337.

GREENE, N., KASS, M., AND MILLER, G. 1993. Hierarchical Z-Buffer Visibility. In *Proceedings of ACM SIGGRAPH 93*, ACM Press/ACM SIGGRAPH, New York, J. Kajiya, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 231–238.

HAKURA, Z. S., AND GUPTA, A. 1997. The Design and Analysis of a Cache Architecture for Texture Mapping. In *24th International Symposium of Computer Architecture*, ACM/IEEE, 108–120.

IGEHY, H., ELDRIDGE, M., AND PROUDFOOT, K. 1998. Prefetching in a Texture Cache Architecture. In *Workshop on Graphics Hardware*, ACM SIGGRAPH/Eurographics.

KELLEHER, B. 1998. PixelVision Architecture. Tech. rep., Digital Systems Research Center, no. 1998-013, October.

KLEIN, A., LI, W., KAZHDAN, M., CORRÊA, W., FINKELSTEIN, A., AND FUNKHOUSER, T. 2001. Non-Photorealistic Virtual Environments. In *Proceedings of SIGGRAPH 2000*, ACM Press/ACM SIGGRAPH, New York, E. Fiume, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 527–534.

LATHROP, O., KIRK, D., AND VOORHIES, D. 1990. Accurate Rendering by Subpixel Addressing. *IEEE Computer Graphics and Applications 10*, 5 (September), 45–53.

MCCABE, D., AND BROTHERS, J. 1998. DirectX 6 Texture Map Compression. *Game Developer Magazine 5*, 8 (August), 42–46.

MCCORMACK, J., AND MCNAMARA, R. 2000. Tiled Polygon Traversal Using Half-Plane Edge Functions. In *Workshop on Graphics Hardware*, ACM SIGGRAPH/Eurographics.

MCCORMACK, J., MCNAMARA, B., GIANOS, C., SEILER, L., JOUPPI, N. P., CORELL, K., DUTTON, T., AND ZURAWSKI, J. 1999. Implementing Neon: A 256-Bit Graphics Accelerator. *IEEE Micro 19*, 2 (March/April), 58–69.

MOREIN, S. 2000. ATI Radeon HyperZ Technology. In *Workshop on Graphics Hardware, Hot3D Proceedings*, ACM SIGGRAPH/Eurographics.

NVIDIA. 2001. HRAA: High-Resolution Antialiasing Through Multisampling. Tech. rep.

PINEDA, J. 1988. A Parallel Algorithm for Polygon Rasterization. In *Computer Graphics (Proceedings of ACM SIGGRAPH 88)*, ACM, 17–20.

SHIRLEY, P. 1990. *Physically Based Lighting Calculations for Computer Graphics*. PhD thesis, University of Illinois at Urbana Champaign.

WILLIAMS, L. 1983. Pyramidal Parametrics. In *Computer Graphics (Proceedings of ACM SIGGRAPH 83)*, ACM, 1–11.

WOO, R., YOON, C., KOOK, J., LEE, S., AND YOO, H. 2002. A 120-mW 3-D Rendering Engine With a 6-Mb Embedded DRAM and 3.2 GB/s Runtime Reconfigurable Bus for PDA Chip. *IEEE Journal of Solid-State Circuits 37*, 19 (October), 1352–1355.