INVITED PAPER

Graphics Processing Units for Handhelds

In handheld devices that display 3-D images, lower-power consumption for graphics hardware can be achieved by algorithms that reduce bus traffic with system memory and provide more efficient rendering.

By Tomas Akenine-Möller and Jacob Ström

ABSTRACT | During the past few years, mobile phones and other handheld devices have gone from only handling dull text-based menu systems to, on an increasing number of models, being able to render high-quality three-dimensional graphics at high frame rates. This paper is a survey of the special considerations that must be taken when designing graphics processing units (GPUs) on such devices. Starting off by introducing desktop GPUs as a reference, the paper discusses how mobile GPUs are designed, often with power consumption rather than performance as the primary goal. Lowering the bus traffic between the GPU and the memory is an efficient way of reducing power consumption, and therefore some high-level algorithms for bandwidth reduction are presented. In addition, an overview of the different APIs that are used in the handheld market to handle both two-dimensional and three-dimensional graphics is provided. Finally, we present our outlook for the future and discuss directions of future research on handheld GPUs.

KEYWORDS | Computer graphics; graphics processing units; mobile devices; rasterization

I. INTRODUCTION

In 2006, about 800 million mobile phones were sold all over the world; according to World Cellular Information Service, this figure was expected to grow to more than 1 billion during 2007. The mobile phone has evolved into a powerful tool that can be used not only as a phone but also as a music and video player, calendar, TV, radio, and still-image and video camera, as well as for gaming and surfing the Internet. In particular, the display technology has changed dramatically over the past ten years, and today resolutions of 320 $\times$ 240 are common (and increasing) with 65 thousand to 16.8 million possible colors per pixel. Hence the mobile phone can be considered the most widespread device with graphics capabilities [1]. Today, the visual content and its quality is a key differentiating factor for mobiles, and it is therefore of uttermost importance to do this part of the platform well in order to create a successful mobile phone. To create three-dimensional graphics, specialized hardware called *graphics processing units* (GPUs) are used to render high-quality images. Most phones today do not yet have a GPU, but we are in the middle of a major change in which more and more phones are being equipped with them. We believe that the most important application will be to increase the experience in user interfaces (UIs) and games. User interfaces can indeed be the killer application. This is illustrated by devices such as the iPhone, where advanced graphics have been included mostly for the user interface. See Fig. 1 for an example of user interfaces for mobile phones.

It should also be noted that the Android platform, managed by Google, for mobile devices also has extensive support for graphics, with support for OpenGL ES 1.0. Other examples of possible applications are navigation in three-dimensional maps (e.g., Google Earth), E-commerce (to look at products from different viewpoints), screen savers, augmented reality, avatars, messaging, and advertising. For professional users such as firefighters and rescue workers, there could be further applications in visualization such as three-dimensional blueprints of buildings, three-dimensional maps with overlay graphics, etc.

This survey will investigate the implications of graphics rendering when using these limited devices and present a high-level overview of algorithmic improvements for

Manuscript received June 5, 2007; revised November 20, 2007.
T. Akenine-Möller is with the Department of Computer Science, Lund University, 221 00 Lund, Sweden (e-mail: tam@cs.lth.se).
J. Ström is with Ericsson Research, 164 80 Stockholm, Sweden (e-mail: jacob.strom@ericsson.com).

Digital Object Identifier: 10.1109/JPROC.2008.917719

0018-9219/$25.00 ©2008 IEEE

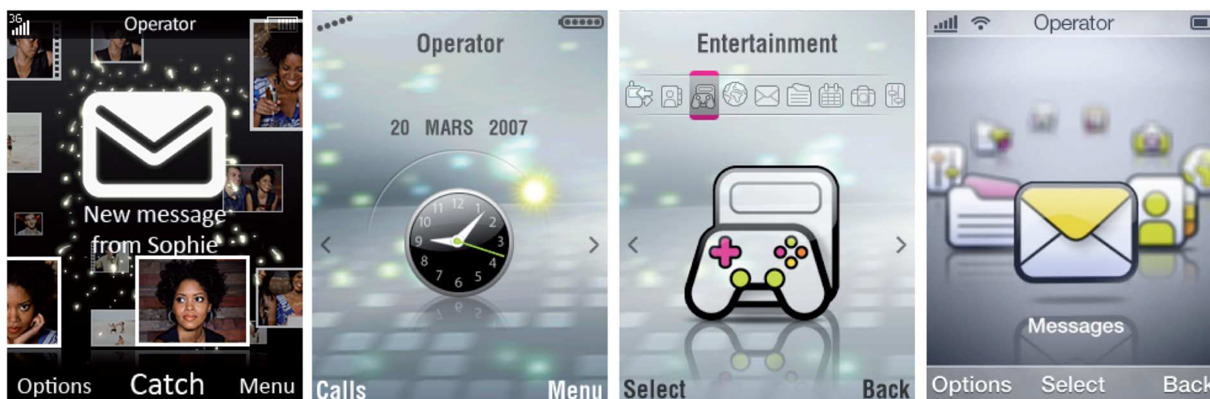

Fig. 1. ***Examples of possible user interfaces for mobile phones. The UI to the right uses three-dimensional graphics to be able to fit more icons on screen. (Images courtesy of TAT AB, http://www.tat.se.)***

graphics hardware designed primarily for battery-driven platforms.

II. CONCEPTUAL OVERVIEW OF GPU ARCHITECTURE

In this section, we will give a quick overview of a GPU for a desktop PC, i.e., a GPU that has not been developed for a mobile phone. First, we note that the memory subsystem is a massive design and provides enormous amounts of bandwidth to different memories. This is illustrated in Fig. 2. As can be seen, there is a dedicated graphics memory and also a separate system memory, which both the CPU and the GPU can use. In the GeForce 8800, which is a recent GPU from NVIDIA, the peak bandwidth to the graphics memory is 86.4 gigabytes per second (GB/s) [20]. Originally, GPUs were designed only to render triangles, lines, and points. This has changed, and they are now used for general-purpose computations for performance-critical algorithms that can be efficiently expressed in the streaming nature of the GPU architecture. Here, the focus will be mostly on the graphics aspect of GPUs, with concentration on rendering triangles, which are the "rendering atoms" of real-time graphics. A conceptual overview of the rendering pipeline when drawing triangles is shown in Fig. 3.

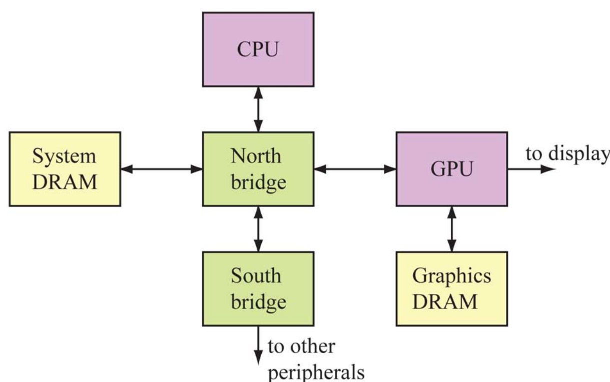

Fig. 2. ***A typical memory architecture for a nonmobile GPU, where the buses between the units have capabilities of many tens or hundreds of gigabytes per second. (Illustration after Kilgariff and Fernando [15].)***

An application, for example, a game, is running on the CPU and feeds the GPU with triangles to be rendered and states indicating where, and how, these should be rendered. To transform the vertices of a triangle into the desired positions, the vertices are processed in a unit called "Vertex Processing." Here, different types of per-vertex computations are performed. Then a unit called "setup" assembles triangles from three vertices and computes data that are constant over the triangle. After that, the pixels that are inside the triangle are found, and for each such pixel, per-pixel processing commences. This includes computing the color of the pixel using a variety of techniques to obtain the desired visual result. At the end of the GPU processing, different types of frame buffer operations take place. This includes resolving visibility (making sure that the triangles closest to the viewer are visible) and blending, for example. Finally, the result (if any) may be written to memory. From a general-purpose computation perspective, the rendering of a triangle can be seen as an implicit **for**-loop over the pixels inside the triangle. For more details about GPU architectures, we refer to Kilgariff and Fernando [15].

Both the vertex processing and pixel processing stages are fully programmable using high-level C-like languages. One such example is the Cg language [18]. These programs running on the GPU are often called *shaders*. The instruction sets for vertex processing and pixel processing have converged over the years and are now identical. Hence, the conceptual diagram above can be implemented in a *unified shader architecture*. This includes recent graphics cards, such as the Geforce 8800 series [20] and the Xbox 360 from AMD. It should be noted that there are also geometry shaders [5], which can "create" new

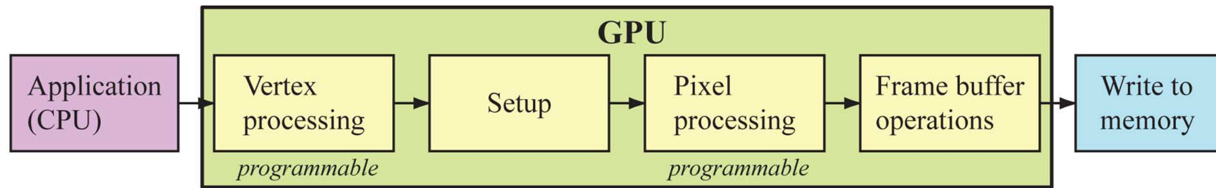

Fig. 3. *Conceptual overview of the rendering pipeline. The application is running on the CPU and is responsible for sending, for example, triangles to be rendered to the GPU. The GPU first processes the vertices of the triangles by executing a user-supplied program (aka shader) for each vertex. Then the setup unit gathers vertices to form a triangle and computes various constants used for rasterization. For each pixel inside the triangle, a program is executed in the pixel processing unit. The major task of this program is to compute the color of the pixel. Several different frame buffer operations may then be performed, and finally, pixel data may be written to memory.*

geometry inside the GPU. The idea of unified shader architectures is to duplicate many (e.g., 128) unified shader units, which can execute vertex, pixel, and geometry shader programs, within the GPU. As illustrated in Fig. 4, a single unified shader core can execute any type of shader and forward the result to another shader core (or even to itself), until the entire chain of shaders has been executed.

An important feature of unified shader architectures is that they provide load balancing. For example, a highly detailed character composed of tiny triangles with simple lighting may utilize the full computing resources of the GPU by allocating more shader units for vertex processing and fewer for per-pixel processing.

It is interesting to compare the computing peak performance of CPUs and GPUs. The NVDIA GeForce 8800 GT provides 346/520 Gflops [20], which should be compared to a quad-core Intel Xeon at 3 GHz with SSE2, providing a peak performance of 96 Gflops. It should be noted that GPUs for desktops make heavy use of both caching and compression techniques.

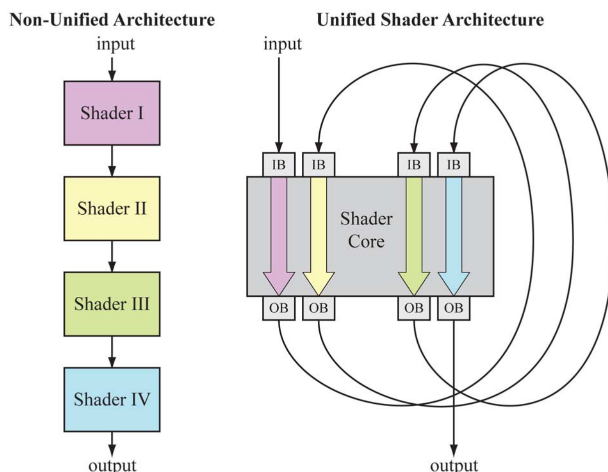

Fig. 4. *A nonunified architecture versus a unified shader architecture. The advantage of unified approach is that one can have several shader cores and use them for any type of shader (I–IV in this example). This gives better load balancing. IB and OB are input and output buffers.*

III. LIMITATIONS AND CONSIDERATIONS OF MOBILE PHONES

On a functional level, GPUs for mobile devices are very similar to GPUs for desktops, in general. However, under the hood, many differences can be found. One example is shown in Fig. 5. The physical size of the mobile phones limits how much technology can fit in the device. The key differencing factors for a mobile phone are that they:

1) are powered by batteries;
2) have limited CPU instruction set;
3) use a low clock frequency;
4) have limited memory and memory bandwidth;
5) do not have fans or other cooling technology.

In the following, we discuss these a bit more in detail.

A *mobile* device is by definition powered with batteries and is also small in order to be portable. Most or all limitations stem from these two constraints: battery-driven and small size. To provide long use-time on the battery, it is important to make sure that the system of the mobile phone uses as little energy as possible. There are many power-reduction techniques for this, some of which will be discussed in this paper. The increase in storage capacity per unit volume for batteries is increasing at a relatively slow rate of about 5–10% per year [21], [25], which further increases the difficulty of the situation. Even if batteries would suddenly become much more powerful, power consumption could not be increased infinitely since the small size of the mobile phone means that it would need to dissipate large amounts of heat from a small area, and thus become too hot to handle [21]. The instruction set for the CPU is often reduced. For example, sometimes the division instruction is missing, and often floating-point support is not available. The power consumption is an increasing function of the clock frequency, and hence it is kept rather low as well.

In many systems for mobile phones, the memory architecture is quite different from that of desktop systems, as can be seen in Fig. 6. A flash memory is often used as a "hard

Fig. 5. *The small physical size of a mobile device creates limitations that are not present for desktops.*

disk" in USB sticks, for example, but also in mobile phones. This type of memory is nonvolatile (it does not go away when you turn the power off), but it is not a disk per se. In addition, there is a rather small system RAM that is located "off chip." Since reading from flash memory in general is much faster than writing, music, video, executables, etc., are often stored there and loaded into the RAM when needed. The mobile GPU can be located either on the same chip as the CPU or in a separate circuit. In many cases, there is no dedicated graphics memory and no separate bus for graphics-related memory accesses. In addition, the width of the bus can be as small as 16 bits; however the trend is towards wider buses such as 32 or perhaps even 64 bits. In order to save power consumption, the transistors that make

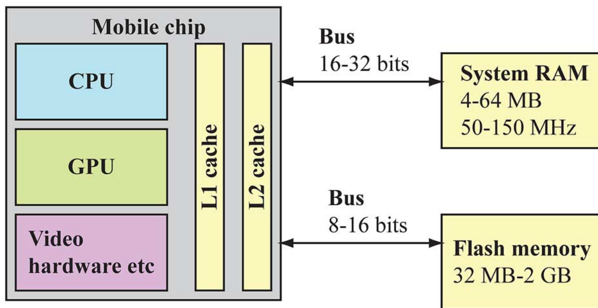

Fig. 6. *The memory system in a mobile phone is quite different from a desktop PC system (compare to Fig. 2). Here, it may be the case that both the CPU and the GPU and other hardware share the same chip (though these can be separate chips as well). In addition, there is usually a limited amount of system RAM and some kind of flash memory (e.g., NAND flash), which is used as a "hard drive." Note that an L2 cache is seldom seen in today's mobile devices but is something that will be incorporated in the near future.*

up the CPU and the GPU are created using low-power processes, which make them leak very little energy. However, in order to access off-chip memory (such as system RAM), one needs to drive high capacitances for the buses, and this costs a lot in terms of energy. Hence, memory accesses are very expensive in terms of energy compared to computation [10], and this observation will be used for a variety of algorithms later in this paper.

The resolution of mobile phone displays has increased steadily in recent years. Fig. 7 plots a diagram for a set of models from Sony Ericsson (T68m, T610, K600, K800i, and W880i), where the angle between two pixels to one point on the retina is plotted against the year the handset was first sold. PlayStation Portable (PSP), iPod Video (leftmost Apple data point) and the iPhone (rightmost Apple data point) are also shown. A viewing distance of 30 cm was used for all portable units, and 60 cm was used for the 24-in desktop display of 1920 $\times$ 1200. As can be seen in the diagram, while at first mobile phones had vastly inferior resolution, they are now starting to surpass that of desktop displays. Naturally, it is not meaningful to increase resolution further than the resolution of the eye. While the resolution of the eye depends on contrast and viewing conditions, it is on the order of 1 arc minute (one-sixtieth of a degree, plotted in the diagram in dashed black) [8].

It should be noted that the display is a major power consumer in a mobile device, and it has been reported that about 30% of the power in a laptop is consumed by the display [17], mainly because inefficient backlighting is needed. Exactly how much is consumed by graphics-intensive applications, such as games, is not well known, but it can be expected to be a high percentage of the remaining

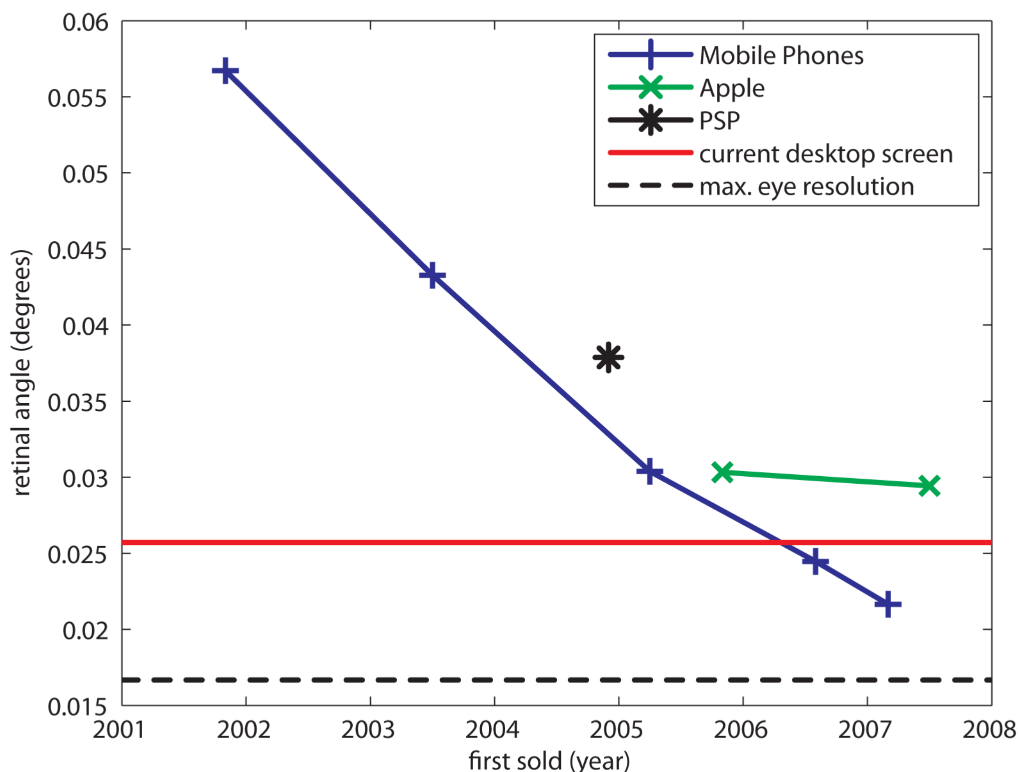

Fig. 7. *The resolution of mobile phone displays has increased rapidly, and their corresponding retinal angle has surpassed or is on par with that of large desktop displays, even considering the fact that they are viewed at half the distance.*

consumption. In this paper, we do not consider algorithms or technology for reducing power consumption in the display. However, it should be noted that using light-emitting diodes (LEDs) offers promise for highly energy-efficient displays. Each pixel would then "consist" of several LEDs, and the issue of backlighting would be avoided altogether. Examples include organic LEDs [26] and nano-LEDs [3].

In the following, we will first discuss application programming interfaces (APIs) specifically developed for mobile devices, and why this is so. Then we will discuss high-level algorithmic improvements for handheld GPUs.

IV. APIs SPECIFICALLY FOR MOBILE DEVICES

In the desktop world, APIs for three-dimensional real-time graphics are dominated by DirectX and OpenGL. Interestingly, these APIs are used more and more to accelerate the rendering of the windows manager in Mac OS X and in Windows Vista, and can also be used to accelerate rendering of PDF files.

When three-dimensional graphics entered the mobile market, a plethora of proprietary APIs surfaced. However, to reduce market fragmentation, there was a need for standardized APIs, both for the Java and the native level. This need was met by the Java standard *Mobile 3D Graphics* (M3G), standardized by JCP, and the native standard OpenGL ES (ES is short for embedded system), standardized by Khronos. On the native side, OpenGL ES 1.0 was more or less created as a subset of OpenGL 1.3, where parts that were not useful for mobile phones were removed, such as computer-aided design support. On the Java side, one major constraint has been the speed at which Java code can be executed. Writing scene manipulation structures such as a scene graph in Java code would thus be very inefficient. Therefore, a scene graph is included in M3G so that its functions can be implemented in fast native code, using just a few Java calls. However, low-level manipulation of rendering primitives is also possible in M3G, making it both a high- and low-level API. In contrast, OpenGL ES is a purely low-level API, and the application writers usually write their own scene-manipulating structures. Since these can be adapted to the problem at hand (a scene-manipulating structure for a car game may be quite different from that of a fighting game, say), they can be more efficient than a general scene graph. The mobile APIs have been playing a rapid catchup game with the desktop APIs. Whereas version 1.0 of OpenGL ES was mostly intended for software renderers, 1.1 targeted hardware renderers, and the latest version, 2.0, allows for programmable shaders. A programmable version of M3G 2.0 is also under way. OpenGL ES is also used outside the mobile world; Sony uses a variant of OpenGL ES 2.0 in their PlayStation 3 game console.

Both M3G and OpenGL ES are APIs for three-dimensional graphics. However, two-dimensional vector graphics is being revived for mobile devices. The major advantage of vector graphics over traditional bitmap, or raster, graphics is that it is scalable. In bitmapped graphics, an image is defined by specifying the color for a number of pixels directly. Zooming in a bitmapped image thus means interpolating values between pixels, and edges will not stay sharp. In contrast, in vector graphics, the content is defined by specifying the colors of geometrical primitives such as lines, curves, and polygons. An image can then be rendered from any distance, and thus turned into pixel colors. A zoomed-in vector graphics image can thus preserve sharp edges, which is why it is called "scalable." This means that a user interface, for example, can be made only once and then rendered at different display resolutions with nice appearance. With bitmap graphics, one bitmap per resolution would be required. Applications of vector graphics include graphical user interfaces, animated messages, simple games, high-quality text (e.g., for reading books), and two-dimensional maps. See the tiger in Fig. 8 for a clear example of the usefulness of scalable two-dimensional graphics.

In the desktop world, there is no standardized hardware accelerated API for two-dimensional vector graphics similar to Direct3D or OpenGL. One reason for this may be that software implementations of, for instance, the Flash player are considered fast enough even if run on a CPU. On a mobile device with a weaker CPU, software implementations may not be fast enough. Furthermore, a hardware implementation is not only faster but also more

Fig. 8. ***A tiger rendered with OpenVG at two different scales. Notice that details becomes visible at the larger scale. This model contains Bézier curves, color ramps, etc. Rendered with Hybrid Graphics' OpenVG reference implementation.***

efficient in terms of power consumption. Therefore, Khronos developed OpenVG, which is a hardware-friendly low-level API of the same kind that OpenGL ES is for three-dimensional graphics. Higher level APIs are also available; *Flash Light* is the mobile version of the desktop Flash standard, and *SVG Tiny* is ditto for SVG. A high-level API such as SVG could be implemented on top of a low-level one, such as OpenVG. The APIs typically support antialiased polygons, lines, curves, and strokes. They also include effects such as color gradients, textures, and text rendering. The major difference compared to three-dimensional APIs is the support for curved primtives, such as Bézier curves. This enables zooming on a curve without any tessellation artifacts.

It is possible to use a graphics architecture created for three-dimensional graphics to accelerate vector graphics such as OpenVG. The two-dimensional primitives are then tessellated to triangles before being rendered. This can be an attractive solution if there is a need for both three-dimensional and two-dimensional vector graphics. However, on systems that only require vector graphics, the graphics hardware can be made smaller in terms of surface area and be made to consume less power. It should be noted that many of the enticing "three-dimensional" effects used in user interfaces, such as warping the image to make it "fly away" from the screen, can be realized using two-dimensional APIs such as OpenVG. If no true three-dimensional effects are needed, such as objects that occlude each other in complex ways, it is possible to create a low-power small-surface-area system using OpenVG-only-capable hardware.

Whereas native applications are standard on desktops, most phones sold today (denoted "feature phones") do not have the capability for the user to install new native software. Instead, these phones rely on Java for downloadable applications. Lately, the number of "smart phones," i.e., phones that allow the user to install new native software, has been soaring. This has created a need for standardizing part of the native interface, so that applications can be ported more easily between different models of mobile phones. Khronos' OpenKODE standard is a response to this need, standardizing file system, data types, etc., as well as the interaction between other media APIs such as OpenGL ES and OpenVG.

V. TECHNIQUES FOR MOBILE GPU ARCHITECTURES

There are several ways on the circuit level in which power consumption can be lowered in a mobile GPU. In the first category, we find, for example, clock gating, which means temporarily shutting down parts of the chip that are currently not used. It is also common to use low-power processes when designing the chip, which means that transistors will leak less power than on desktop GPUs. Both clock gating and the use of low-power processes are standard practices on mobile phone chips. Clearly, there are many more low-level energy conservation techniques. This includes, for example, energy-efficient arithmetic units and voltage scaling techniques. Such techniques are out of the scope of this paper, as they are not directly related to graphics.

Another important way to reduce power consumption has to do with high-level algorithmic changes in the GPU. As argued before, memory accesses are expensive in terms of power, and therefore there is a significant amount of research dealing with ways to lower memory bandwidth usage. On desktops, memory bandwidth-saving techniques have mostly been used to increase the performance for a certain level of memory traffic. On mobile phones, however, it may be equally important to reduce the bandwidth (and hence the power consumption) for a certain level of performance. Using high-level algorithmic changes in the GPU, bandwidth can often be traded for computation. In the next sections, we will discuss three such techniques: tiling architectures, buffer compression, and texture compression. However, computation also carries a power cost, and Moore points out that transistor leakage is increasing at a very rapid pace [19]. This implies that the number of transistors should be kept down, since this will reduce leaking. Also in the case of reducing computation, high-level algorithmic changes in the GPU can provide substantial savings. As an example to this, the recently introduced programmable culling unit (PCU) [14] can reduce the number of pixel shader instructions to be executed by down to 50%.

A. Tiling Architectures

We have already observed that there are several major differences between GPUs on handhelds and for desktops. In terms of architectures for GPUs, there is yet another difference. On handhelds, there is a particular type of GPU, which we will call a *tiling architecture*. The first example of this is the Pixel-Planes 5 graphics system [9], and some architectures like this have existed even for the desktop market. At the moment, however, there is no tiling architecture on the desktop market, but there are several targeting handhelds.

As a high-level overview of "standard" rendering, a triangle is first transformed to its correct position, and per-vertex computations take place. Then the triangle is sent down the graphics pipeline for further per-pixel processing. A tiling architecture, on the other hand, renders only to one tile at a time. A *tile* is typically a rectangular block of pixels, say, 16 × 16 pixels. These architectures need to send all the geometry of the entire frame to the graphics card, which then sorts the triangles into lists. The entire rendering area is divided into tiles, and for each tile that a triangle overlaps after transformations, a pointer to that triangle is stored. Hence, after this has been done for all triangles, each tile has a list with all the triangles that overlap that particular tile.

At this point, rendering of the first tile commences. The triangles of that tile are thus rendered into this first tile. The major advantage is that the depth and color buffers and other buffers only need to be as big as the tile, for example, 16 × 16 pixels. This amounts to a small piece of memory, and this can be stored on-chip. We call this the *tile buffer*. Therefore, off-chip memory accesses to external (off-chip) memory are avoided to a large extent. When all triangles for that tile have been rendered, the buffers can be written out to external buffers. Another useful feature is that once the sorting has been done, per-pixel rendering to different tiles can be done in parallel. However, this requires multiple hardware units for per-pixel processing.

For antialiasing on tiling architectures, there is an advantage that is of particular interest on mobile devices. Assume that an antialiasing scheme uses four samples per pixel. For a conventional architecture, the frame buffer (including, e.g., color and depth) needs to be four times as large, and this buffer is stored in system RAM or video RAM. For a tiling architecture, either the on-chip memory for a tile buffer needs to be four times as large or the tile size is reduced to a fourth of the initial tile size (e.g., going from 16 × 16 pixels to 8 × 8 pixels). In the latter case, the on-chip requirements do not increase but the tile size decreases, and so more tile lists are needed and four times as many tiles need to be processed. In the former case, only the on-chip memory requirements increase, which may be reasonable in some cases.

There is a clear advantage for tiling architectures since off-chip accesses can be avoided in many cases. However, the memory bandwidth increases in other places in the pipeline. For example, the triangle/tile sorting needs to be done, and creating the triangle lists increases bandwidth usage a bit. At this point, there is an ongoing debate on which architecture is the best, and we simply note that this is clearly scene-dependent. This means that there will be three-dimensional scenes where a tiling architecture performs much better than a standard architecture, but the opposite is also true. Unfortunately, there is no academic study analyzing the advantages and disadvantages in terms of hardware implementation and memory bandwidth usage. The currently available information can be found on company Web sites with tiling architecture products, but we avoid citing those documents as there is no description of how the comparisons were made.

Examples of existing tiling architectures for handhelds are the Mali series from ARM and the MBX and SGX architectures from Imagination Technologies.

B. Compression and Decompression Techniques

One way to reduce the number of memory accesses is to compress the data. In Fig. 9, a high-level rendering architecture is illustrated to the left. Standard components in such a system include a depth buffer, which handles occlusion by storing the distance from the viewer for each rendered pixel, and textures, which are images that are glued onto surfaces in order to make them appear more realistic. The textures and the depth buffer are stored in memory, and in order to render a scene, the GPU must access this data over the bus. For instance, in order to render one pixel using standard quality filtering (called trilinear mipmapping), the GPU must read eight texels (texture pixels) from the texture. Caches certainly help to lower the number of bus accesses, but even with efficient caching, memory bandwidth is usually a performance-limiting and/or power-consuming factor.

Buffer Compression: To lower the bus bandwidth usage, one can deploy buffer compression. The buffer, such as the depth buffer, is then compressed and stored in memory in compressed form. When accessed by the GPU, the compressed data are transferred over the bus and decompressed in real-time in the GPU. This is shown to the right in Fig. 9. If the data need to be written, such as in the case with the depth buffer, it must be compressed on-the-fly before being transferred to the memory.

For some buffers, such as the depth buffer, it is important that the compression not destroy the data; the compression

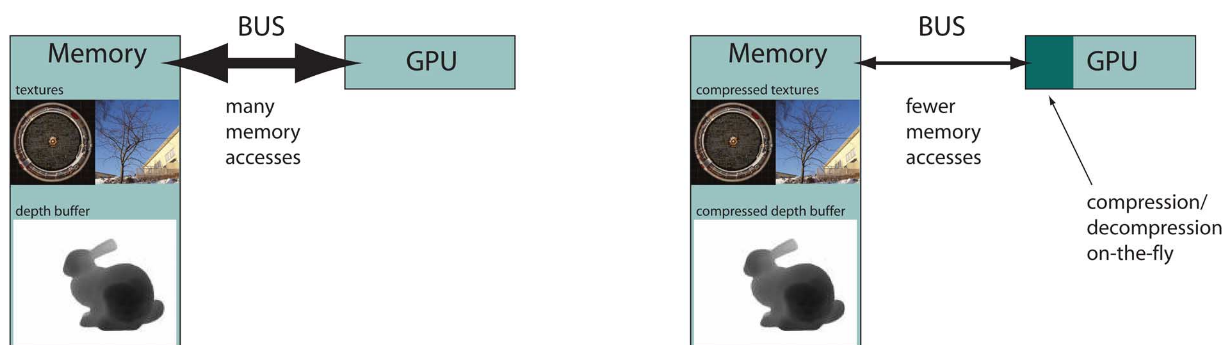

Fig. 9. ***Left: the bus traffic is intensive between the memory, where geometry, textures and the depth buffer are stored, and the GPU. Right: by compressing textures and buffers, and decompressing them on-the-fly, bus traffic can be greatly reduced.***

must be *lossless*. In that case it is not always possible to compress the data at all. Therefore, depth compression algorithms reserve the same amount of memory as if the data were not compressed at all, e.g., 384 bits per 4×4 block. The GPU keeps a bit on-chip to indicate whether the block was compressed. If compressed, only a few of the 384 bits are transferred, and decompression takes place in the GPU. Otherwise, all bits are transferred, and the decompression is bypassed. Thus depth buffer compression does not save memory space, only bus bandwidth. It should be noted that the same algorithms for buffer compression can be used for mobile devices and for desktop GPUs. For surveys on depth buffer compression and color buffer compression, we refer to Hasselgren and Akenine-Möller's paper [13] and Rasmusson *et al.*'s paper [23]. Finally, we note that there is no substantial difference in buffer compression algorithms between handheld GPUs and desktop GPUs, except that the implementation should be kept to a minimum on the handheld.

Texture Compression: Textures form a special case of accessing memory, since they are usually not written to during rendering. This means that they can be compressed off-line when the application is created, and therefore the compression does not need to be fast. Decompression however must happen on-the-fly during rendering and still needs to be fast. Moreover, the slight degradation in image quality obtained by lossy compression is usually not possible for the human visual system to detect, and due to this fact, it is possible to compress the textures more efficiently than if lossless compression was used. Since the texels in a texture can be accessed in an arbitrary order, it is important that the GPU can easily calculate where in memory a certain texel resides. One way to solve this is to divide the texture into blocks of, say, 4×4 pixels and compress each block to a fixed number of bits, normally 64 bits. On desktop machines, DXTC is the de facto standard in texture compression [16]. However, on mobile phones, Ericsson Texture Compression (ETC) [27] is becoming popular since it is standardized in OpenGL ES. ETC works by specifying a color per 2×4 pixel area and then modulating the luminance in every pixel additively, as shown in Fig. 10.

VI. OUTLOOK

Hopefully, it is clear from this paper that there are many similarities between desktop GPUs and mobile GPUs. However, with the introduction of mobile three-dimensional graphics, many new algorithms have been invented by necessity due to the limitations of the device. It is likely that this type of development would not have happened in the absence of mobile graphics. The major reason to use a GPU compared to doing all the rendering on the CPU is that the GPU can be made much more energy-efficient.

To reach better power utilization, we believe that *approximate* rendering will become more important. Currently, there are lossy algorithms in many units and algorithms in graphics. This includes all work where functions are represented using wavelets or spherical harmonics, where only the most important coefficients are stored and used for computing lighting. Texture compression is by design also lossy. However, all buffers are currently compressed without loss. Note that for television, we put up with pretty poor quality, and so it makes sense to compress with loss even for color buffers, for example. In such a case, the error must be under precise control so that it does not grow too big. At this point, some initial research has been done on lossy color buffer compression [23]. In addition, there has been recent work [14] on avoiding pixel shader executions. The idea is to conservatively estimate per tile whether the contribution of pixel shader executions inside the tile gives a contribution greater than zero. If so, per-pixel executions follows. Otherwise, pixel shader executions can be avoided. This can be extended into a lossy approach: if the contribution is less than a small threshold, we avoid the work. If a slight degradation in image quality comes with a longer use time on the battery, we believe that these types of algorithms can have great impact for mobile GPUs. However, it is probably best if the user of the phone gets access to turn the "knob" of image quality versus battery time.

Another approach would be to adapt the concept of *frameless rendering* [4], [7] to GPUs. This means that only a subset of the pixels on the display are written to every frame. For example, we can decide to write to only every fourth pixel, and hence the pixels would be divided into four pseudorandom subsets that together include all pixels.

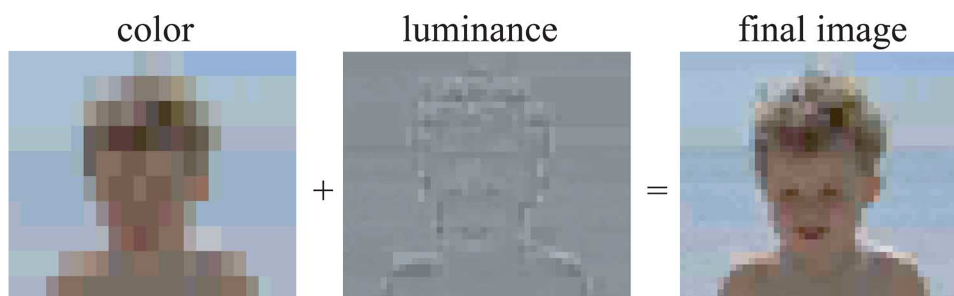

Fig. 10. *ETC works by specifying a color in every 2×4 pixel area and then modifying the luminance in every pixel.*

Fig. 11. ***Example on how quickly the mobile GPUs are catching up with desktop GPUs. Remember the fans of desktop GPUs grow every year, but the mobile phones become smaller. (Image appears courtesy of ARM.)***

Fig. 12. ***This image was rendered using OpenGL ES 2.0, and 14 different vertex and pixel shader programs were used. Antialiasing using four samples per pixel was used for higher quality. (Image appears courtesy of AMD.)***

Therefore, if nothing is moving, then the image will converge after four frames. When objects are moving, we will get a poor motion blur effect. Preferably, the rendering pattern should be pseudorandom to hide artifacts as much as possible. There are also possibilities to save bandwidth from the frame buffer to the display, since only 25% of the buffer changes per frame, and only this part needs to be sent to the display. The display controller needs some logic to handle this though. However, there is quite a bit of research to be done before this type of algorithm will make it to GPUs. Some initial work on stochastic rasterization has been done [2], where a moving triangle is sampled stochastically within a frame. This can be used to render motion blur, depth of field, or glossy reflections. In general, motion blur is used to hide jerkyness for feature-film rendering, and it may be possible that a lower frame rate can be used on the handheld with this type of algorithm in place. We note that this could be another "knob" for the user.

If autostereoscopic displays, i.e., displays where the user can see stereo without any extra peripherals, break through in terms of usage, then efficient rendering to these displays may become a hot research topic. There has been some initial research on this topic [11], but the target was not GPUs. More recently, it has been shown that a lot of the contents in a texture cache can be exploited for all views given a specialized rasterization order [12]. It is clear that there is a lot of inherent coherency in the left and right images for stereo, and it may be possible to utilize that in a novel architecture.

There is much to be said about rendering on handhelds, but one thing is certain: mobile graphics will affect all of our daily lives, and soon every handheld is likely to have specialized hardware for two-dimensional and/or three-dimensional graphics. See Figs. 11 and 12 on what to expect. ■

Acknowledgment

The authors wish to thank P. Nordlund, AMD, and V. Miettinen and E. Sørgård, ARM.

REFERENCES

[1] T. Akenine-Möller and J. Ström, "Graphics for the masses: A hardware rasterization architecture for mobile phones," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 801–808, 2003.

[2] T. Akenine-Möller, J. Munkberg, and J. Hasselgren, "Stochastic rasterization using time-continuous triangles," in *Graph. Hardware*, Aug. 2007, pp. 7–16.

[3] D. Appell, "Nanotechnology: Wired for success," *Nature*, vol. 419, no. 6907, pp. 553–555, 2002.

[4] G. Bishop, H. Fuchs, L. McMillan, and E. J. Scher Zagier, "Frameless rendering: Double buffering considered harmful," in *Proc. ACM SIGGRAPH 1994*, 1994, pp. 175–176.

[5] D. Blythe, "The DirectX 10 system," *ACM Trans. Graph.*, vol. 25, no. 3, pp. 724–734, 2006.

[6] J. X. Chen and E. J. Wegman, *Foundations of 3D Graphics Programming: Using JOGL and Java3D*. Berlin, Germany: Springer-Verlag, 2006.

[7] A. Dayal, C. Woolley, B. Watson, and D. Luebke, "Adaptive frameless rendering," in *Proc. Eurograph. Symp. Render.*, 2005, pp. 265–275.

[8] M. Deering, "The limits of human vision," in *Proc. 2nd Int. Immersive Projection Technol. Workshop*, 1998.

[9] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel, "Pixel-planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories," in *Proc. ACM SIGGRAPH*, 1989, pp. 79–88.

[10] R. Fromm, S. Perissakis, N. Cardwell, C. Kozyrakis, B. McCaughy, D. Patterson, T. Anderson, and K. Yelick, "The energy efficiency of IRAM architectures," in *Proc. 24th Annu. Int. Symp. Comput. Architect.*, 1997, pp. 327–337.

[11] M. Halle, "Multiple viewpoint rendering," in *Proc. ACM SIGGRAPH*, 1998, pp. 243–254.

[12] J. Hasselgren and T. Akenine-Möller, "An efficient multi-view rasterization architecture," in *Proc. Eurograph. Symp. Render.*, 2006, pp. 61–72.

[13] J. Hasselgren and T. Akenine-Möller, "Efficient depth buffer compression," in *Proc. Graph. Hardware*, Sep. 2006, pp. 103–110.

[14] J. Hasselgren and T. Akenine-Möller, "PCU: The programmable culling unit," *ACM Trans. Graph.*, vol. 26, no. 3, pp. 92.1–92.10, 2007.

[15] E. Kilgariff and R. Fernando, "The GeForce 6 series GPU architecture," in *GPU Gems 2*, M. Pharr, Ed. Reading, MA: Addison-Wesley, 2005, pp. 471–491.

[16] K. I. Iourcha, K. S. Nayak, and Z. Hong, "System and method for fixed-rate block-based image compression with inferred pixel values," U.S. Patent 5 956 431, Sep. 21, 1999.

[17] C. B. Margi, K. Obraczka, and R. Manduchi, "Characterizing system level energy consumption in mobile computing platforms," *Wireless Netw., Commun. Mobile Comput.*, vol. 2, pp. 1142–1147, 2005.

[18] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, "Cg: A system for programming graphics hardware in a C-like language," *ACM Trans. Graphics*, vol. 22, no. 3, pp. 896–907, 2003.

[19] G. E. Moore, "No exponential is forever: But 'forever' can be delayed," in *Proc. Int. Solid State Circuits Conf.*, 2003.

[20] NVIDIA, GeForce 8800 GPU architecture overview, Nov. 2006. Tech. Brief.

[21] K. Pulli, T. Aarnio, K. Roimela, and J. Vaarala, "Designing programming interfaces for mobile devices," in *Proc. IEEE Comput. Graph. Applicat.*, Nov./Dec. 2005.

[22] K. Pulli, T. Aarnio, V. Miettinen, K. Roimela, and J. Vaarala, *Mobile 3D Graphics: With OpenGL ES and M3G*. San Mateo, CA: Morgan Kaufmann, 2007.

[23] J. Rasmusson, J. Hasselgren, and T. Akenine-Möller, "Exact and error-bounded approximate color buffer compression and decompression," in *Proc. Graph. Hardware*, Aug. 2007, pp. 41–48.

[24] J. Owens, "Streaming architectures and technology trends," in *GPU Gems 2*, M. Pharr, Ed. Reading, MA: Addison-Wesley, 2005, pp. 457–470.

[25] K. Takamoro, J. Tabuchi, M. Watanabe, and Y. Hirota, "Development of battery pack for mobile phones," *NEC Res. Develop.*, vol. 44, no. 4, pp. 315–320, Oct. 2003.

[26] J. Shinar, Ed., *Organic Light-Emitting Devices: A Survey*. Berlin, Germany: Springer-Verlag, 2004.

[27] J. Ström and T. Akenine-Möller, "iPACKMAN: High-quality, low-complexity texture compression for mobile phones," in *Proc. Graph. Hardware*, 2005, pp. 63–70.

ABOUT THE AUTHORS

Tomas Akenine-Möller received the M.Sc. degree in computer science and engineering from Lund University, Sweden, in 1995 and the Ph.D. degree in graphics from Chalmers University of Technology, Sweden, in 1998. He is now a Professor in the Department of Computer Science, Lund University.

He has worked on shadow generation, mobile graphics, wavelets, high-quality rendering, and collision detection. He has several papers published at the ACM SIGGRAPH conference, including on the pioneering topic of mobile graphics (with J. Ström) in 2003. He is a coauthor (with E. Haines) of *Real-Time Rendering* (Wellesley, MA: AK Peters, 2002). He is currently active part-time in a startup company (Swiftfoot Graphics) for novel graphics hardware algorithms. His current research interests are in graphics hardware for mobile devices and desktops, new computing architectures, collision detection, and high-quality rapid rendering techniques.

Dr. Akenine-Möller received the Best Paper Award at Graphics Hardware 2005 (with J. Ström) for the ETC texture compression scheme, which is now part of the OpenGL ES API.

Jacob Ström received the M.Sc. degree in computer science and engineering from Lund University, Sweden, in 1995 and the Ph.D. degree in image coding from Linköping University, Sweden, in 2002.

In 1998 he received a Fulbright grant and spent one year with the Massachusetts Institute of Technology Media Lab, Boston, as a visiting Ph.D. student. He is currently a Senior Specialist in graphics and image processing with Ericsson Research, Stockholm, Sweden. He has worked with mobile graphics, and together with T. Akenine-Möller wrote the first SIGGRAPH publication on mobile graphics in 2003. His current interests include compression techniques for graphics.

Dr. Ström received the Best Paper Award at Graphics Hardware 2005 (with T. Akenine-Möller) for the ETC texture compression scheme, which is now part of the OpenGL ES API.