IEEE Computer Graphics and Applications
www.computer.org/cga

Making Augmented Reality Practical on Mobile Phones, Part 2

Daniel Wagner and Dieter Schmalstieg

Vol. 29, No. 4
July/Aug. 2009

Projects in VR

Editor: Lawrence Rosenblum and Simon Julier

Making Augmented Reality Practical on Mobile Phones, Part 2

Daniel Wagner and Dieter Schmalstieg, ***Graz University of Technology***

In part 1, we introduced a software environment for augmented reality (AR) on mobile phones, discussed development and debugging strategies, and showed how to execute several tasks of a common AR system in parallel on a mobile device. Here, we discuss how to overcome the most severe limitations, such as memory, rendering speed, and computational power. We analyze in detail where an optimized mobile phone AR application spends most of its processing time and give an outlook on what to expect in the next few years.

Conservative Memory Bandwidth Usage

After the CPU's processing capabilities, memory bandwidth is typically the next most limiting factor in computationally expensive phone applications. Today's mobile phones use a unified memory architecture: CPU, GPU, camera, and video share a single pool of general-purpose memory, which makes memory bandwidth scarce. Unfortunately, owing to the execution units' parallel nature, benchmarking memory-intensive code is often difficult.

To save battery life, memory is usually not only small but also slow. A cache miss is therefore even more expensive than on a PC. To make matters worse, misses in the comparably small caches (typically 32 Kbytes) are more likely if data isn't carefully structured.

Explicitly limiting bandwidth usage is therefore essential for fast application performance. For example, compact pixel formats such as RGB565 (16 bits) are preferable to RGB888 (24 bits) or even RGBX8888 (32 bits). This is especially important for implementing video backgrounds, which require uploading a large texture every frame. On devices with very slow texture upload, downscaling to half-size beforehand can produce considerably faster overall performance.

When an application requires alpha support, RGBA5551 or RGBA4444 is preferable. You can save even more bandwidth by using compressed texture formats. Owing to the complex compression schemes used by today's GPUs, this approach is viable only for static textures that can be compressed offline.

Not only rendering but also computer vision tasks benefit from representing images in compact pixel formats. Many mobile phone cameras provide images in YUV format, which can be converted to grayscale with minimal computation. Because most computer vision trackers use grayscale internally, YUV is preferable to RGB variants, which require more memory and internal format conversions.

To improve cache usage in the renderer's vertex stage, vertex data should be passed in an interleaved format. For static meshes, using vertex buffer objects can increase speedup. Although the driver typically won't be able to store vertex data in special memory owing to the unified-memory approach, it can reorder and preprocess vertex data for optimal performance.

Scene-graph traversals should be kept to a minimum. Ideally, a scene graph should be traversed only once per frame, because the graph's structure (including nodes containing 3D mesh data) typically doesn't fit in the cache.

Rendering

The rendering step requires more device-specific optimization than any other step of the AR pipeline. PCs typically have only one configuration: OpenGL or Direct3D abstracting the graphics card. On mobile phones, several more different APIs exist, of which usually only one is available or provides optimal performance. OpenGL ES 1.x (ES stands for "embedded systems") supports only a fixed-function pipeline and can be programmed in fixed-point and floating-point data formats. OpenGL ES 2.x doesn't support the fixed-point data format or a fixed-function pipeline; instead, everything must be done using shaders.

Additionally, OpenGL ES 1.x can run in software or on accelerated hardware. Exploiting each configuration is crucial for optimal performance. Hardware rendering doesn't permit direct frame buffer access. So, the application must draw the

 Published by the IEEE Computer Society

video background using texturing. Software rendering is especially slow in texturing but allows direct frame buffer access. Hence, it's important to render the video background by directly copying it into the frame buffer to achieve meaningful performance.

Another topic to consider is a floating-point versus a fixed-point data format for geometry. Depending on the device-specific implementation, either one can lead to better performance. Using vertex buffer objects is usually preferable because it allows converting static data into whatever format the renderer prefers internally.

Fixed-Point Mathematics

Many numerical algorithms for computer vision or graphics intensively use floating-point operations. PC CPUs have dedicated floating-point units and can even interleave floating-point and integer operations so that algorithms can execute faster when using both floating-point and integer units than when using integers only.

In contrast, most mobile phone CPUs don't have floating-point units. Instead, the compiler must insert floating-point emulation code where required. Consequently, floating-point calculations are typically approximately 40 times slower than their integer variants. Avoiding floating-point operations is therefore essential to achieve high performance. Unfortunately, this often means rewriting critical code sections. This implies that careful dynamic code profiling is necessary to determine an algorithm's most time-critical sections and concentrating optimization efforts there.

As a quick first improvement, replacing double-precision floating-point types with single-precision types alleviates arithmetic units and preserves memory bandwidth. However, major speedup will occur only if you replace floating-point code with integer or fixed-point variants. ARM CPUs can perform barrel (bit) shifting as an integral part of most operands, which means that fixed-point and integer math perform almost identically.

Two general problems with fixed-point usage are reduced precision and numeric range. Development of fast, high-precision basic functions such as sines, cosines, or square roots can require much effort. It's usually more efficient to determine the precision requirements and then implement trigonometric and other basic functions using lookup tables and possibly interpolation. You can implement scalar inversion (1/x) and inverse square roots (and hence also regular square roots) using Newton-Raphson iteration. Although these primitives are often part of open source packages for embedded platforms, advanced primitives such as SVD (singular value decomposition) or Cholesky factorization (both required for 3D computer vision) typically aren't available in fixed-point format.

Although fixed-point programming is tedious, it's currently a key factor in achieving fast performance of typical research applications on mobile phones. A fixed-point implementation alone might be less attractive on a fully featured CPU with a floating-point unit. However, we've found that fixed-point algorithms tend to be generally more optimized with respect to cache coherence and memory bandwidth, and are therefore often faster than their conventional counterparts.

Sporadic, Low-Bandwidth Networking

The low processing capabilities of mobile phones and similar embedded platforms naturally raise the idea of outsourcing computationally intensive tasks to a powerful server. However, with mobile computing, connectivity can change over time. In contrast to stationary computing, for mobile phones, loss of connection isn't an exception but a rule. For example, a user might deliberately turn off networking support or enter an area without connectivity. So, you should treat networking as an option to enable special features or boost performance.

Although fixed-point programming is tedious, it's currently a key factor in achieving fast performance of typical research applications on mobile phones.

Past research tried thin-client approaches to outsource tasks such as image processing and rendering to wirelessly connected servers. With these approaches, bandwidth quickly becomes the limiting factor. Data compression can't overcome this problem because it consumes too much processing power or—for lossy image compression—degrades quality too severely. One of our early experiments in phone tracking involved sending thresholded, run-length compressed images (typically 2 Kbytes) to a server for image analysis. Although bandwidth didn't pose a serious problem, the network round-trip time induced so much latency that we instead used faster local image processing on the phone.

Another networking challenge arises from massive multiuser applications. Theoretically, peer-to-peer architectures scale better with the number of users, but this assumes continuous connectivity for keeping replicated data in sync. In a client-server

Table 1. Smartphones used in the benchmark tests.

Smartphone	CPU	MHz	GPU
HTC Tornado	TI OMAP850	200	None
HTC Excalibur	TI OMAP850	200	None
Palm Treo700W	Intel XScale	312/520*	None
Motorola Q	Intel XScale	312	None
Motorola Q9	TI OMAP2420	330	On the CPU

** We overclocked the CPU to 520 MHz to compare the clock rate's influence.*

(a) (b) (c)

Figure 1. Test models rendered for benchmarking: (a) a textured cube, (b) a Venus of Willendorf model, and (c) a model of a car. The cube represents a minimal workload, the Venus represents a typical workload, and the car represents a maximal workload.

architecture, setting up server operation takes additional effort, but you can compensate for this by storing important state at the server and letting sporadically connected clients synchronize with the server-side state. The server can perform housekeeping operations for application-specific simulation without relying on clients. Moreover, you can use the server to provide static data such as textures or sound files on demand.

Testing AR Performance on Current Phones

To estimate how each stage of an AR pipeline for mobile phones affects performance in practical terms, we performed three benchmarks on five low- to high-end smartphones (see Table 1). The benchmarks (see Figure 1) were

- a textured cube, which represented a minimal workload application;
- a textured model of the Venus of Willendorf model, which had 2,625 triangles and represented a typical high-quality 3D model; and
- an (untextured) CAD model of a car, which had 25,652 triangles.

Although the third model is clearly too big for most phones, we used it to estimate next-generation devices' performance.

We ran the benchmarks in seven configurations, each successively adding more steps of the AR pipeline and increasing the overall workload. Each new step depends on the previous steps' outcomes and slightly influences the following steps' performance owing to different cache and memory loads. So, we decided to not treat each step independently but to evaluate each new step together with its pipeline predecessors.

Figure 2 shows the results for all seven configurations. Test configuration 1 ran an empty main loop, which rendered an empty (black) rectangle on the screen. All the phones performed extremely well with the empty main loop. They had enough processing power to blit the screen at 150 frames per second or more. (A blit copies the screen content into video

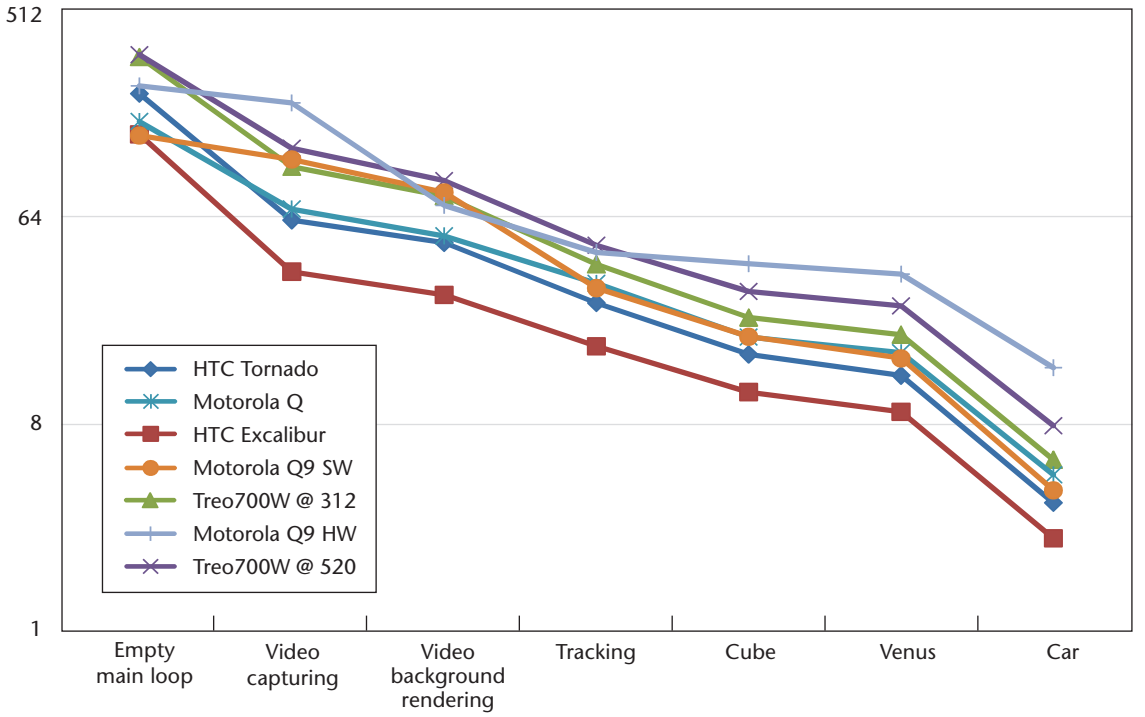

Figure 2. Benchmark results in logarithmic scale. Each successive test increased the workload on the mobile phones.

memory, making it visible.) So, this step no longer posed a noticeable bottleneck. The AR framework's overhead is negligible: even on the slowest phones, it added up to less than 7 milliseconds.

Test 2 added video capturing, which introduced a highly different workload on the various devices we used in our tests. Although the Q9 hardly lost any performance, all other devices suffered enormously. The main reason is that the Q9 delivers images in RGB565 directly, whereas the other devices must convert YUV12 to RGB565, which demands much processing power. Although our software contains optimized code for this conversion, it still slowed down overall performance considerably. Yet some phones such as the Excalibur lost more performance than others, possibly because of memory bandwidth limitations.

Test 3 added video background rendering. This test produced results inverse to those of image capture: the Q9 lost the most performance. In particular, the hardware-accelerated version performed worse than the version using software rendering. This performance loss is no surprise because the Q9 must upload the video into a texture and render that texture. The other devices, running software rendering only, can directly copy the camera image into the frame buffer instead. So, the Treo outperformed the Q9, even when not overclocked.

Test 4 included tracking, therefore forming a complete AR pipeline except for 3D rendering. Adding tracking affected all phones similarly. Nevertheless, the Q9 and Treo performed far above the rest. Yet even the Excalibur still performed at 17.7 frames per second. Taking the difference in timing between this and the previous test, we estimate that tracking takes 22.3 milliseconds on the slow Excalibur and only 9.8 milliseconds on the overclocked Treo. As we've previously pointed out, tracking scales almost linearly with the CPU's clock rate, independent of the CPU manufacturer.[1]

Tests 5 through 7 also rendered the three objects in Figure 1, thereby putting different workloads on the rendering subsystem. On these tests, all phones performed similarly on the cube and the Venus, despite the latter's considerably larger polygon count. Obviously the 870 vertices, grouped efficiently in a single triangle strip, created no big bottleneck for the software-implemented vertex stage.

However, the car model created a noticeable burden on all phones, including the hardware-accelerated Q9. Whereas the Excalibur and Tornado ran at frame rates below what can be considered real time in AR, the other devices maintained interactive rates. Only the Q9 ran at 15 frames per second, its maximum frame rate.

In this article's two parts, we've presented guidelines and best practices gained from developing AR applications on mobile phones over six years. We strongly believe that owing to fundamentally different design goals, mobile phones will remain a device class separate from desktop and notebook computers in the foreseeable future. So, many of the general restrictions we've discussed will remain valid. Unless an unexpected major breakthrough occurs in embedded-circuit or battery research, mobile phone hardware will remain an order of magnitude less capable than PCs. Instead, small form factors and optimized battery usage will probably continue to drive mobile phone design. This implies that techniques for real-time processing on mobile devices will remain different from PCs despite advances on both platforms.

Today, only few high-end phones have built-in hardware floating-point units. In a few years, most mobile phones will likely ship with hardware support for OpenGL ES 2.0. Unlike OpenGL ES 1.x, it doesn't support fixed-point operations, which makes floating-point units mandatory for meaningful operation on all mobile devices and might significantly accelerate many algorithms.

However, the processing power of the mobile phones' main CPU core probably won't increase dramatically, owing to power limitations. It seems more likely that CPUs will acquire more special-purpose units, such as for graphics, video processing, and digital signal processing. The introduction of cross-platform standard APIs such as OpenKODE (www.khronos.org/openkode) or OpenCL (www.khronos.org/opencl) will allow simplified access to these features.

Reference

1. D. Wagner and D. Schmalstieg, "ARToolKitPlus for Pose Tracking on Mobile Devices," *Proc. 12th Computer Vision Winter Workshop* (CVWW 07), Verlag der Technischen Universität Graz, 2007, pp. 139–146; www.icg.tu-graz.ac.at/Members/daniel/Publications/ARToolKitPlus.

***Daniel Wagner** is a postdoctoral researcher at the Graz University of Technology and deputy director of the Christian Doppler Laboratory for Handheld Augmented Reality. Contact him at wagner@icg.tugraz.at.*

***Dieter Schmalstieg** is a professor of virtual reality and computer graphics at the Graz University of Technology, where he directs the Studierstube research project on augmented reality. Contact him at schmalstieg@icg.tugraz.at.*