Standards & Emerging Technologies

Editor: Sumi Helal ■ University of Florida ■ helal@cise.ufl.edu

Smart Camera Phones: Limits and Applications

Jonatan Tierno and Celeste Campo

Camera phones have been in existence almost five years now, and as their popularity increases, users are demanding better image capturing capabilities and more functionality. Fortunately, the processing capabilities of current smart phones can support new applications such as image processing, movement detection, pattern recognition, color detection for the visually impaired, and augmented reality games. To support these new applications, some smart phones use the Mobile Media API, which lets developers build multimedia applications—including video recording and image capture—into phones from different manufacturers.

Here, we review MMAPI technology and explore how people might use their phones' cameras as more than just conventional cameras. We performed our study on a typical European high-end GSM smart phone, the Nokia 6600, and compared some measures with the Nokia 3650. Although our results apply to these specific terminals, you could generalize them for other smart phones. Our goal was to understand this technology and its limitations and to clarify what we can (and can't) achieve with smart camera phones.

THE MOBILE MEDIA API

J2ME grants access to a phone's camera through the optional library MMAPI,[1] which is used mainly to retrieve, open, and play multimedia contents from any source—local or remote. The MMAPI manages the camera just like any other multimedia source, in a straightforward manner. In mid 2003, only the Nokia 3650 supported MMAPI, but now at least 30 models from different manufacturers support it, and we expect more models will soon follow. (Manufacturers supporting MMAPI include Alcatel, Misubishi Electric, Nokia, Siemens, and Sony Ericsson; for specific model information, see http://developers.sun.com/techtopics/mobility/device/device). Unfortunately, not all J2ME phones (nor the Mobile Information Device Profile 1.0 or 2.0) have this library, even when the terminal includes a built-in camera.

The processing capabilities of current smart phones can support new applications such as image processing, movement detection, and pattern recognition.

The camera is considered a multimedia source, so the locator `capture://video` can identify it. With the `System.getProperty(String key)` call, we can determine whether the phone supports video and audio recording and in what formats. The video capture is equivalent to a reproduction (you use the same API for both playing and recording), and the programmer manages it using a `Player` created with the `createPlayer(String locator)` call in the `Manager` class. We can access the camera's images as a video stream and then post them on the screen. Using the `VideoControl.getSnapshot(String imageType)` method, we can take a photograph—in this case, an individual video frame—and present it in a range of formats, sizes, and bit depths.

To process data, we need to take the RGB pixel values from the image in a compressible standard image format. The most straightforward way to accomplish this is to use the `Image.getRGB()` method, which stores the RGB values as an array of integers. However, this method is only available in MIDP 2.0. We can also use the `DirectGraphics.getPixels` method in the Nokia UI proprietary API,[2] but not all Java-enabled phones have this library. The hardest way to achieve this would be to parse the image file to access the pixels. We adopted the straightforward solution, because the Nokia 6600 phone is MIDP 2.0 enabled. We also processed data using the Nokia 3650 (which supports MIDP 1.0) by parsing a PNG file. This was simple because the phone produced uncompressed PNG files, so we just had to remove the headers.

CAMERA ACCESS PERFORMANCE IN J2ME

Phones like the Nokia 6600 usually have a VGA camera chipset; in other words, the phone captures photographs of 640 × 480 pixels—the same as what the phone's native camera application can achieve. We measured the acquisition time and file size for the three available formats in the Nokia 6600 phone: PNG, BMP, and JPEG.

Published by the IEEE CS and IEEE ComSoc ■ 1536-1268/05/$20.00 © 2005 IEEE

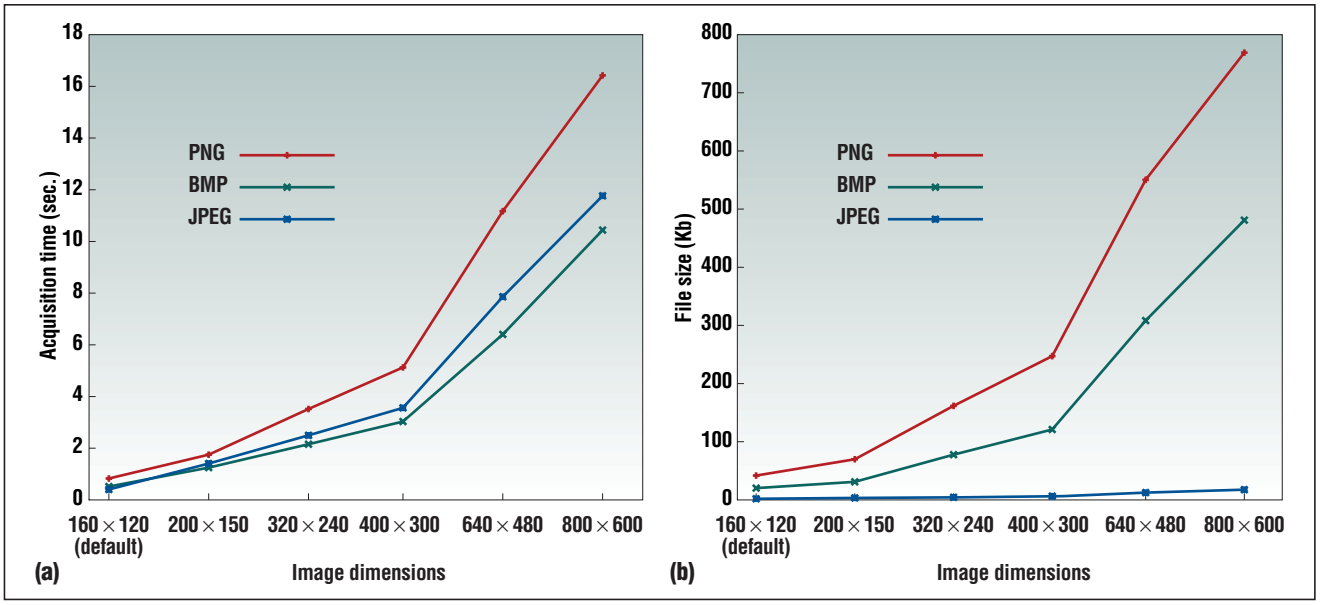


Figure 1. Comparison among PNG, BMP, and JPEG format images: (a) acquisition time and (b) file size.

We use the `getSnapshot()` method as the function's parameters. The default format is a 160 × 120 PNG image. We took three measures for each size and format. Figure 1 presents the average results. We also consider how image complexity influences file size. We took a photograph first of a white wall and then of a scene with people and objects. Table 1 presents the results.

When the file was too big for the phone to process, the application either blocked, so the camera wasn't available until we rebooted the phone, or it (more frequently) aborted with an error message. Also, the phone always used the default image size (160 × 120) and then rescaled the image if the width and height parameters weren't the default, so the image quality didn't improve.

For real-time applications, such as those requiring movement detection, the phone never achieved a high processing speed—at best, it came close to two images per second. JPEG was the fastest format, but BMP was a close second.

Why are JPEGs faster, even though they require a more complex process? A typical camera chipset can work using several formats: uncompressed (YCrCb) or compressed (JPEG) images. Thus, the Kilo Virtual Machine would need to process PNG or BMP images, but hardware creates JPEG images. We have performed tests on the Nokia 3650, which has reached approximately seven frames per second.

JPEG images are the smallest, and the file size is less dependent on image dimensions, owing to the lossy compression used. PNG image files are larger but the image quality is better. The BMP format is uncompressed, so the file size is proportional to the image width and length. The image quality is quite low, though, because a 256-color palette is used. A higher bit depth would increase the file size too much.

Using Symbian native code instead of J2ME (see www.symbian.com), native applications can access the camera in terms of independent images and not video (as with MMAPI).[3] The viewfinder isn't provided, so the application must take photographs and show them to the user periodically. The images are obtained asynchronously using *active object* and are offered in a Symbian-specific format, which the application can later convert to other formats. One of the parameters is image quality, which can be high or low.

A low-quality image (called a *snapshot*) is quarter VGA—that is, 160 × 120 pixels, and 4,096 colors (12 bits). A high-quality image is VGA (640 × 480) and 16 million colors (24 bits). Typi-

TABLE 1

Comparing file size and acquisition time among formats based on image complexity.

File format	Image complexity	File size (bytes)	Acquisition methods		
			getSnapshot (ms)	createImage* (ms)	getRBG (ms)
PNG	Simple	28,953	828	219	0
	Complex	44,248	875	219	16
BMP	Simple	20,278	532	141	16
	Complex	20,278	500	156	16
JPEG	Simple	1,223	610	203	15
	Complex	3,469	641	219	16

*To manage an image, MIDP uses the class Image, created with the static method Image.createImage().

STANDARDS & EMERGING TECHNOLOGIES

TABLE 2
Duration of a 10,000,000 loop and of a single operation (average of 10 measures).

Operation	Loop time (ms)	Single operation (ns)
Empty loop	1,372.0	0.0
Array extraction	2,062.7	69.1
Increment	1,573.3	20.1
Sum	1,565.9	19.4
Shift	1,562.5	19.1
Multiply	1,862.6	49.1
Divide	12,396.7	1,102.5
Less or equal	2,375.1	100.3
Less	2,351.4	97.9
Equal	2,554.8	118.3

cally, snapshots are used for video and viewfinder, and high-quality images for the actual photograph. It's also possible to switch between illumination profiles (*night* and *day*), which modify image brightness by adjusting the white balance.

MMAPI, on the other hand, creates a video stream by putting together snapshots taken periodically from the camera. An image obtained with `getSnapshot()` is simply one of these snapshots, rescaled and reformatted when necessary.

IMAGE PROCESSING PERFORMANCE IN J2ME

If we want to implement applications that process images, our device must be able to run heavy algorithms very fast, sometimes even in real time. We'll also probably need to use a lot of data at the same time from one or more images. So, two important features that we measured were operation speed and the amount of dynamic memory available for a MIDlet.

We measured the speed of the processor's basic operations to determine the overall speed and to identify the fastest and slowest applications. We refer only to low-level arithmetic operations, because those are the only math operations available in the J2ME API.

Operation speed

To measure time, we used the `System.currentTimeMillis()` method, which has a precision of milliseconds. We repeated each operation inside a loop a certain number of times and measured the duration of the loop. Then we did the same with an empty loop. The duration of a single operation is that of the first loop, minus the empty loop, over the number of iterations. With this method, we measured how long it took to

- extract a variable from an array,
- increment a variable,
- add two variables and store the result,
- multiply and divide in powers of 2 using bit shift,
- multiply and divide a variable and a constant, and
- compare two variables (equal, less or equal, less).

We implemented an application that executes this method with each of these operations. The results obtained were distorted by other processes running concurrently on the phone, such as phone service management and other operating system tasks. In fact, the operations took much longer when we pressed the keys or navigated the menus. These tasks will be always present in the terminal, so we used the average measure.

The results in Table 2 reveal which operations we should avoid in heavy computations to make our applications more efficient. Division is the slowest operation—almost two orders of magnitude slower than an addition. Surprisingly, compare operations are also several times slower than others, including multiplication. An array extraction is slower than an addition or a shift because it requires indirection.

Of course, the algorithms determine which operations we need to use, but, using this information, we can minimize the number of divisions inside a loop to make it faster or we can replace the division with a shift. Note that the operations should use integer numbers; if decimal numbers are needed, the *fixed-point arithmetic* technique allows decimal calculations using integer types.

Because the Nokia 6600 uses a 104-MHz processor, and a simple operation typically takes two clock periods, it's clear that the fastest operations in our results executed approximately at processor speed. This might suggest that a MIDlet is as fast as a Symbian application, but this is only partially true. The Nokia 6600 uses a virtual machine called Monty 1.0 VM,[4] which features a dynamic compilation scheme, based on the Java Hotspot Virtual Machine for J2SE. So the virtual machine has both an *interpreter*, which executes Java bytecodes at runtime, and a *compiler*, which turns bytecodes into native code at runtime. Native code is an order of magnitude faster than interpreted code but takes up additional memory space. The Monty VM solves this using the *profiler*, a third block of the virtual machine. The profiler uses statistic techniques to identify at runtime parts of the code, called *hotspots*, that the MIDlet runs repeatedly. Then the Monty VM compiles the hotspots and interprets the rest of the code, which it executes rarely or only once.

This enhancement assumes that an application spends most of its runtime executing a small portion of the code. The current test MIDlet made this assumption, since all it does is repeat a loop thousands of times. The profiler identifies this loop as a hotspot and compiles it, running at processor speed. This assumption holds true for all the applications we study here. Typically,

an image-processing algorithm will iterate on an array containing pixels of an image and repeat the process on subsequent images over time or on different parts of an image. Thus we can expect good behavior from our MIDlets.

Available dynamic memory

The volatile memory that a device uses to execute Java applications is called *heap*. According to the Nokia 6600 specifications, the heap is 3 Mbytes. To check this, we used a Symbian application *FExplorer* (http://users.skynet.be/domi/download.html) and found that the free dynamic memory, regardless of the data in the memory card or the flash memory, is about 10 Mbytes. However, the heap could be smaller, since it's the memory available only to MIDlets.

To take actual measures, we implemented a simple MIDlet that tries to allocate a byte array of a given size. We performed several tests with different array lengths. We learned that the memory available for a MIDlet isn't limited by the heap (3 Mbytes) but by the volatile memory (10 Mbytes). We also learned that the memory that the KVM can assign to a MIDlet is dynamic; initially, it allows a small amount, but when the memory actually used nears its limit, it assigns more memory to the MIDlet.

WHAT CAN SMART CAMERA PHONES ACHIEVE?

Our experiments reveal that when implementing image processing MIDlets on mobile devices, we must first identify which device resources are available from J2ME. Then we can determine whether these resources offer the same characteristics to a MIDlet as to a native application. This might not be obvious and often can be answered only by performing tests or through trial and error on an actual device.

We also learned that the main restrictions are those imposed by the camera or, more precisely, by the access to the camera that J2ME grants. The size of the available images, which is much smaller than what the camera can reach, and the need for a minimum distance between camera and object, makes implementing applications such as barcode readers or text recognition difficult. Symbian provides the camera's full resolution but doesn't address the focal distance problem. We might address this issue using an add-on lens for close photographs, such as the lens for the Nokia 3650 terminal (Nokia CC-49, www.nokia.com/nokia/0,4879,5813,00.html).

If our application requires real-time processing, then frame speed, which is often quite slow, is the most important feature. Symbian looks like a good alternative, but we have yet to measure this.

Applications that require pattern recognition, such as face recognition, will find their bottleneck in the phone's processing speed and in the reduced API that MIDP offers for math operations. In Symbian, the API is far more complete, but the best way to deal with heavy algorithms in a limited device is to use basic arithmetic operations with integer data, using techniques such as fixed-point arithmetic and lookup tables. We shouldn't expect a dramatic increase in application speed if we move to native Symbian, because the most critical parts of our MIDlet will already run on native code.

Our experiments reveal that when implementing image processing MIDlets on mobile devices, we must first identify which device resources are available from J2ME.

Applications obtaining information from the image's color or brightness will be limited mainly by the camera's automatic white balance, which the programmer can't control. The same process occurs in Symbian, although it might help to adjust the camera's lighting profile.

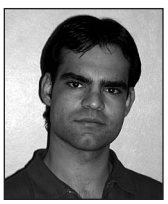
To test the possibilities of J2ME image processing on a mobile phone, we implemented two applications. The first is a color reader designed to help blind people. It detects the colors present on the image and announces them. The second is a simple surveillance application that reviews time images taken by a camera in real time and sounds an alarm when movement is detected.

Eventually, new terminals with better cameras should overcome most of the limitations we've discussed here. The cameras will have not only higher resolution but also better optical zoom and auto focus. More importantly, improved MMAPI implementation will let the terminals exploit the full potential of smart camera phones and their high-speed processors. However, even current terminals can implement interesting and useful applications with imagination and careful design.

REFERENCES

1. *Mobile Media API*, tech. report JSR 135, Java Community Process, http://jcp.org/en/jsr/detail?id=135.
2. "Nokia UI API Programmer's Guide v1.1," Forum Nokia, Version 1.1. 2002, http://forum.nokia.com/main/0,6566,040,00.html?fsrParam=1-3-&fileID=6836.
3. L. Edwards and R. Barker, *Developing Series 60 Applications: A Guide for Symbian OS C++ Developers*, Addison-Wesley, 2004.
4. *The Project Monty Virtual Machine*, white paper, Sun Microsystems, 2002.

Jonatan Tierno is a research assistant in the Department of Telematics Engineering at University Carlos III of Madrid. Contact him at jonatan@it.uc3m.es.



Celeste Campo is an associate professor in the Department of Telematics Engineering at University Carlos III of Madrid. Contact her at celeste@it.uc3m.es.