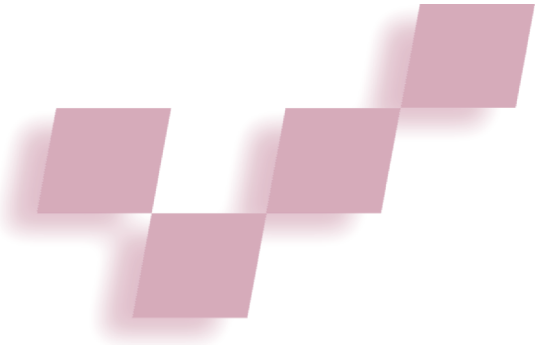
Designing Graphics Programming Interfaces for Mobile Devices



Kari Pulli, Tomi Aarnio, Kimmo Roimela, and Jani Vaarala
Nokia

The authors describe two 3D graphics interfaces for mobile devices and highlight key design decisions and nonobvious approaches taken during their standardization.

Mobile devices have evolved to a point where interactive 3D graphics is becoming feasible. The first standardized 3D programming interfaces for mobile devices—OpenGL ES for native C/C++ and Mobile 3D Graphics (M3G) for Java applications—are now available to hardware vendors and application developers. The interfaces complement rather than compete with each other and can share the same underlying rendering engine, whether implemented in hardware or software.

Three-dimensional graphics on mobile devices is still about converting descriptions of geometry, material, and illumination into pixels shown on a raster display, using the same fundamental algorithms as elsewhere. However, mobile devices' limited capabilities must be reflected in the realizations of those algorithms, as well as in the overall graphics system design.

In this article, we describe a design that attempts to take on that challenge, consisting of OpenGL ES, a low-level API, and M3G (also known as JSR-184), a high-level API for Java. We describe how the two interfaces relate to each other and existing graphics architectures on the desktop, and how they attempt to provide optimal features and performance across the whole gamut of different devices. OpenGL ES and M3G, as well as our presentation of them in this article, derive from a long tradition of graphics systems design. Particularly relevant examples of such previous work are OpenGL,[1,2] OpenInventor,[3] Iris Performer,[4] VRML, and Java 3D.

Background

The most compelling and common use of mobile 3D graphics is familiar: gaming (see Figure 1 for an example). Besides that, 3D graphics can help make the most effective use of the small display in various applications and make a product's user interface more attractive.

Until about 2002, most handheld devices were hardly capable of rendering a Gouraud-shaded cube, let alone displaying it in color (see the "Previous Work" sidebar). Now, display resolutions have reached the level of home computers in the 1980s, while the color depths and computing capacity are on par with a PC in the early 1990s. This is a far cry from today's desktop standards: In terms of fill rate, for example, we are talking about kilopixels rather than gigapixels per second.

Previous Work

3D graphics on mobile phones was first commercialized in 2001 by the Japanese operator JPhone, the company that introduced HI Corp.'s Mascot Capsule engine. Other Japanese operators soon adopted the same engine. Mascot Capsule was initially restricted to event-driven control of skeletally animated characters, using only orthographic projection and z-sorted polygons, but it was later extended with a more generic and robust feature set as well as a lower level API.

Outside of Asia, no proprietary 3D engine ever became a de facto standard. Motorola adopted the Mascot Capsule engine; Sony Ericsson has used Synergenix's Mophun and, more recently, Mascot Capsule; Siemens, Sagem, and Alcatel have used In-Fusio's ExEn. Nokia had its own 3D engine in several monochrome phone models in 2002, but that engine was only used for screen savers exported from Autodesk's 3ds Max and there was no public API.

Representing a slightly different approach, Fathammer's X-Forge engine ships with the games instead of the devices. X-Forge is used in a number of games on the Nokia N-Gage and Tapwave Zodiac platforms.

Publicly available information on these systems is generally restricted to marketing material, making it hard to judge their merits relative to each other and to our approach.

The first software implementations of OpenGL ES and M3G came from Nokia, Hybrid Graphics, HI Corporation, and Superscape. By late 2004 there were also several hardware designs targeted at these APIs.

 Published by the IEEE Computer Society

There are good reasons why the computational capacity is limited, and continues to be so. An often-cited reason is that mobile devices need to be inexpensive. This is true, but a more fundamental reason is that they are small and run on battery power. Battery capacity improves only 5 to 10 percent per year, and even if lots of power were available, compact devices couldn't use it without overheating. Thus, processing units and external memories are clocked at relatively low frequencies, while the silicon area available to memories and processing units is scarce. That precious space will not be filled with floating-point or 3D graphics functions unless there is a clear consumer demand.

One obvious difference between desktop and mobile devices is the display. Mobile displays' smaller size translates to fewer pixels, bringing down the requirements on raw fill rate and polygon throughput. If we are willing to accept the low resolution, mobile devices today offer fairly good processing power relative to the number of pixels, making software-based rendering a feasible approach. For the next several years, most low-end mobile devices supporting 3D graphics will do so by means of a software rendering engine running on an integer-only CPU.

On the other hand, to make the most of each pixel on the small screen, the quality and amount of per-pixel processing would ideally have to be higher than on a desktop device.[5] Techniques such as antialiasing and per-pixel shading become important in high-end devices (gaming devices in particular), which clearly calls for dedicated graphics hardware. A compelling reason to employ such hardware also in mid-range devices is that it will use less power than the same computations on a general-purpose CPU. Furthermore, the same piece of hardware can also accelerate bitmap operations. Several companies have seized that opportunity, and the number of different graphics hardware designs available to device manufacturers has risen from only a few in 2003 to about 10 by the end of 2004. We predict that most high-end and many mid-range devices will include graphics hardware acceleration in five years' time.

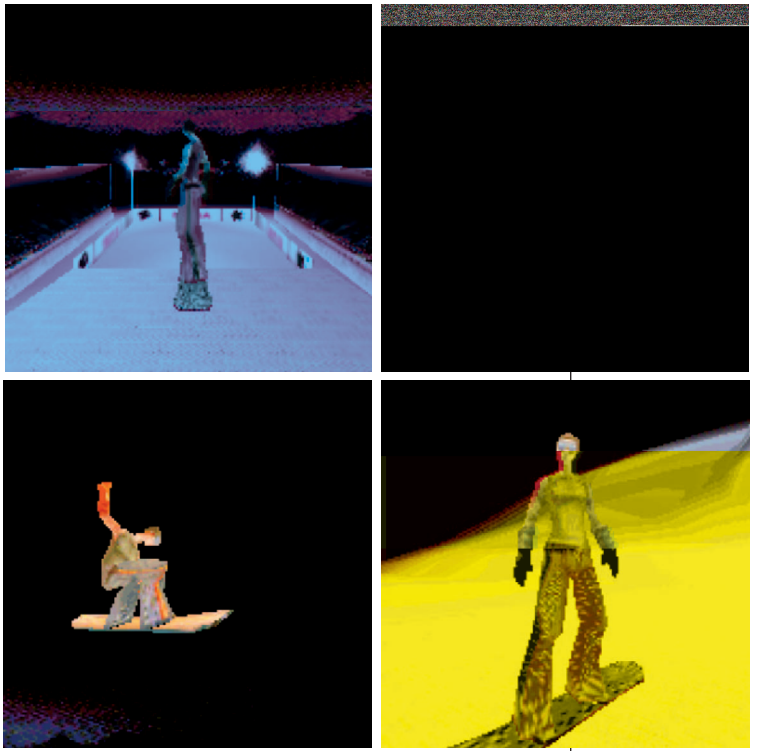
Given the rapid development of high-end graphics hardware and the growing importance of low-end software rendering, available devices will soon span a range of three orders of magnitude in terms of rendering performance. Designing an API to cater to such a variety of devices is clearly a challenge. Yet there is no real alternative: Having a different API for each performance category would be unacceptable to almost all parties involved.

Two APIs for two environments

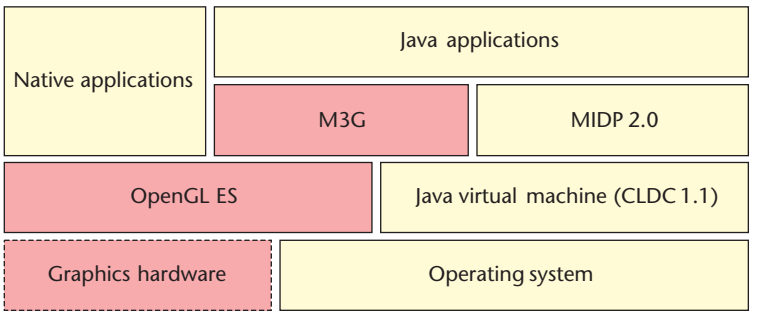
Mobile appliances, such as mobile phones, have traditionally been closed platforms in the sense that no new applications can be installed after the device is purchased. Another type of closed system is game consoles, where manufacturers carefully control who can develop and offer games for the system, and titles are tailored for custom hardware. With this situation, there is limited demand for open-standard APIs.

Mobile devices are opening up, however. Some operating systems—for example, Symbian, embedded Linux, Palm, and Windows CE—allow the installation of native applications written in C or C++, or even assembler, while otherwise closed devices now allow installation of Java midlets. The mobile Java 2 Platform, Micro Edition (J2ME) allows the same applications to run on many different devices and operating systems. Figure 2 illustrates our approach: two different APIs that together provide for both of these worlds.

An open operating system allows installation of native applications, and anyone can, in principle, implement an efficient 3D software engine and the applications using it. What device manufacturers can do, however, is to define and implement a low-level API offered as an operating system service and potentially optimized for



1 Snowboarding game running on an off-the-shelf ARM9-based mobile phone with no hardware support for 3D graphics or floating-point arithmetic. The game is written in Java, using M3G. The underlying M3G implementation sits atop OpenGL ES (Common profile).



2 Software architecture for a high-end mobile terminal. M3G enables 3D graphics for Java midlets, while OpenGL ES serves both native applications and the M3G implementation.

each specific device. This eases application developers' work, provides concrete target functionality for graphics hardware, and allows applications to access such hardware if available.

Java code is generally slower than native code, especially on mobile devices. This is mostly due to the virtual

184 Expert Group (EG) made a good start in simplifying Java 3D, it appeared that they would have to not only make a subset of it but modify and extend it so much that it would have become a different API in the end.

The JSR-184 EG then completely redid the design of M3G, retaining many good design choices of preexisting scene graph APIs. The new design offers essentially

The OpenGL fixed-function pipeline is remarkably flexible due to the orthogonality of its features.[1] Turning features on and off allows creation of interesting effects not necessarily specially designed in the API. For predictability and control, individual features should not have side

begin with, the potentially costly preprocessing step is avoided. Applying these optimizations (and similar ones for matrices and other state information) allows the vertex processing costs to be brought down almost to the level of a raw fixed-point pipeline.

. The OpenGL ES Common profile replaces all doubles with floats and introduces a

Java, on the other hand, is available on many low-end devices that have no support for installable native applications. To make M3G attractive to as broad a range of devices as possible, some of the less common features of OpenGL ES were dropped from the initial version, blending modes were replaced with predefined combinations, and fragment test functions (depth and alpha) were hardcoded to defaults that can only be enabled or disabled. The aim of this was to provide a useful set of features while bringing the combinatorial complexity of state settings down to a level that can still be managed without the complexities of runtime code generation.

Batch processing

The performance implications of fine-grained state management and rendering operations have been recognized in the established native rendering APIs. Display lists, vertex arrays, and vertex buffer objects in OpenGL, as well as state blocks, vertex buffers, and index buffers in DirectX, provide means of grouping rendering state and geometry into blocks that can be effected with a single function call. We use OpenGL terminology in the following discussion.

The foremost benefit of vertex arrays and buffers is that, especially with indexed primitives, data can be efficiently shared and the number of vertex transformations greatly reduced. This led to dropping the begin-end paradigm, where vertex data is input one vertex at a time, from OpenGL ES 1.0. Leaving just vertex arrays and indexed primitives greatly simplified the OpenGL state machine and reduced the number of API entry points required. OpenGL ES 1.1 adds buffer objects, allowing vertex data to be stored in server-side memory for faster access and more optimal data layout.

In theory, display lists can further speed up rendering by allowing implementations to optimize the encapsulated state settings and by reducing the number of API calls. In practice, there is little room for optimization if the operations are sensible to begin with. Also, the number of native function calls is a nonissue on modern CPUs. Given the added code complexity and memory use, there was not enough justification for including display lists in OpenGL ES.

Unlike in native code, minimizing the number of function calls does pay off in Java. Calling native functions from Java, in particular, is costly because of the extra levels of indirection and the extra code required to pass arguments back and forth.

M3G reflects this by batching rendering primitives and state settings into component classes that, in turn, are collected into container classes for rendering, much like in Java 3D. For example, the Appearance container contains rasterization and fragment processing components Material, CompositingMode, PolygonMode, Fog, and Texture2D. The rendering primitives are constructed from vertex and index buffers. The distribution of state settings in the components attempts to group logical sections of the OpenGL rendering pipeline together to enhance object reuse.

Optimize data

Memory resources on mobile devices are scarce, and memory accesses are also expensive in terms of performance and power consumption. To allow more compact data, OpenGL ES defines two extensions over OpenGL 1.3: byte coordinates and paletted textures. For many low-polygon models, even the 8-bit vertex precision is sufficient without any visual degradation. Paletted textures are defined as a built-in format for the compressed textures supported by OpenGL ES, with other formats to be defined by vendor-specific extensions. Paletted textures can often be 75 percent smaller than equivalent 32-bit RGBA or padded RGB textures, which in turn often take half or more of the total memory consumption.

In M3G, all data is stored inside the API objects. Beyond the application setup phase, nothing is accessed from user arrays for rendering. This design allows implementations to optimize the storage formats and locations of different kinds of data.

In M3G, all data is stored inside the API objects. Beyond the application setup phase, nothing is accessed from user arrays for rendering. This design allows implementations to optimize the storage formats and locations of different kinds of data—for example, vertex data can be stored in OpenGL ES buffer objects. It also encourages sharing of data: The same building blocks are used for both immediate mode and retained mode rendering, and objects such as the Appearance component classes, texture images, and vertex buffers can be shared by an unlimited number of both scene graph and immediate mode objects.

Various texture compression formats were also considered for both standards, but since no method that would be free of patents was identified, no texture compression format was mandated.

Optimize the interface

Desktop OpenGL has a total of 98 different entry points for specifying the current color, texture coordinate, vertex, and normal. These redundant entry points differ in their parameter types, requiring code to convert to the selected internal representation. In Java the cost of a large number of entry points is even more expensive. Library method declarations are stored in text strings and take more space, and the declarations are repeated in any application that uses the methods. Because only a few entry points are really needed in practice, eliminating redundancy does not affect the usability of either API much.

Another subtle issue with Java is garbage collection, which automatically releases memory by reclaiming

objects no longer referenced by the application. Depending on the implementation, this might result in a relatively long pause while the garbage collection algorithm runs—this is highly undesirable in an interactive application. For this reason, the user allocates all arrays for returning data from M3G, and the number of items in each input array is passed as a separate parameter (even though Java arrays include length information). This allows the application to recycle the parameter arrays, reducing the need for garbage collection.

Built-in animation engine

The majority of interactive 3D content today relies on keyframe animation and vertex deformation for game characters and objects. Implementing morphing, keyframe interpolation or skinning as part of a mobile Java application is infeasible in light of constraints on performance and download size. Also, a fair bit of mathematics is required, for example, for keyframe interpolation of orientations.[6] To make these techniques widely available to mobile Java applications, they are provided as an integral part of M3G.

. The M3G keyframe animation system enables step, linear, and spline interpolation of all vector, scalar, and quaternion properties in scene graph nodes and other objects. Animation blending is also included, and multiple animation tracks can target the same property with weighted contributions. The flexible animation engine again serves to reduce application size, as relatively complex effects can be achieved using the animation system alone.

To keep the animation data compact, no tangent vectors or other control parameters are included. For linear and step interpolation, this is obvious. For spline interpolation, we use Catmull-Rom splines for scalar and vector valued properties, and a similar scheme described elsewhere[6] (often dubbed squad) for quaternions. Both cases allow nonuniform keyframe timing. Although this requires extra keyframes for precise control of animation paths, it generally produces smooth motion with minimal data and reduced API complexity.

. Morphing and skinning are provided as dedicated subclasses of the basic static mesh class. Morphing is fashioned after the morph node in Java 3D, enabling weighted linear blending between multiple vertex buffers. This is commonly associated with facial animation, but is also usable, for example, for animation of low-polygon game characters in general, of which Figure 1 is one example. The morph target weights can also be animated via the keyframe animation system.

Skinning enables each vertex to be transformed by a weighted blend of several transformations. Somewhat unconventionally, the bones in M3G skinning are regular scene graph nodes. This automatically enables attachments on skinned meshes, such as a torch with a light source that a character is holding, without any extra application code.

. M3G leaves the application in full control of execution. Both animation and rendering are invoked explicitly from the application, with-

use. Leaving the window-binding API unspecified would have caused even more variations than on the desktop.

Because OpenGL was adopted as the basis for the core API, it was a natural choice to take GLX-like APIs as a starting point for the cross-platform windowing API, called EGL. EGL supports most of the functionality in GLX, but in a cross-platform way. It is left for the oper-

References

1. M. Segal and K. Akeley, The Design of the OpenGL Graphics Interface, tech. report, Silicon Graphics, 1994.
2. M.J. Kilgard, "Realizing OpenGL: Two Implementations of One Architecture," Proc. ACM Siggraph/Eurographics Workshop Graphics Hardware, ACM Press, 1997, pp. 45-55.
3. P.S. Strauss and R. Carey, "An Object-Oriented 3D Graphics Toolkit," Proc. 19th Ann. Conf. Computer Graphics and Interactive Techniques, ACM Press, 1992, pp. 341-349.
4. J. Rohlf and J. Helman, "Iris Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics," Proc. 21st Ann. Conf. Computer Graphics and Interactive Techniques, ACM Press, 1994, pp. 381-394.
5. T. Akenine-Möller and J. Ström, "Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones," ACM Trans. Graph., vol. 22, no. 3, 2003, pp. 801-808.
6. K. Shoemake, "Animating Rotation with Quaternion Curves," Proc. 12th Ann. Conf. Computer Graphics and Interactive Techniques, ACM Press, 1985, pp. 245-254.

Kari Pulli is a research fellow at Nokia Research Center and a visiting scientist at the Massachusetts Institute of Technologyt at nnsit 9(e)]TJ-1.8863 - Tc0.01050 1 T[(ofientis)icst P(ib]TJ9(he Mat)-7.8(er snol3D GrT)79(e)-)]Th.