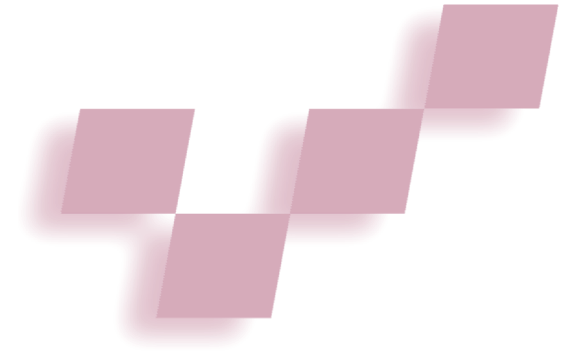
# Designing Graphics Programming Interfaces for Mobile Devices



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
0272-1716/05/$20.00 © 2005 IEEE

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.

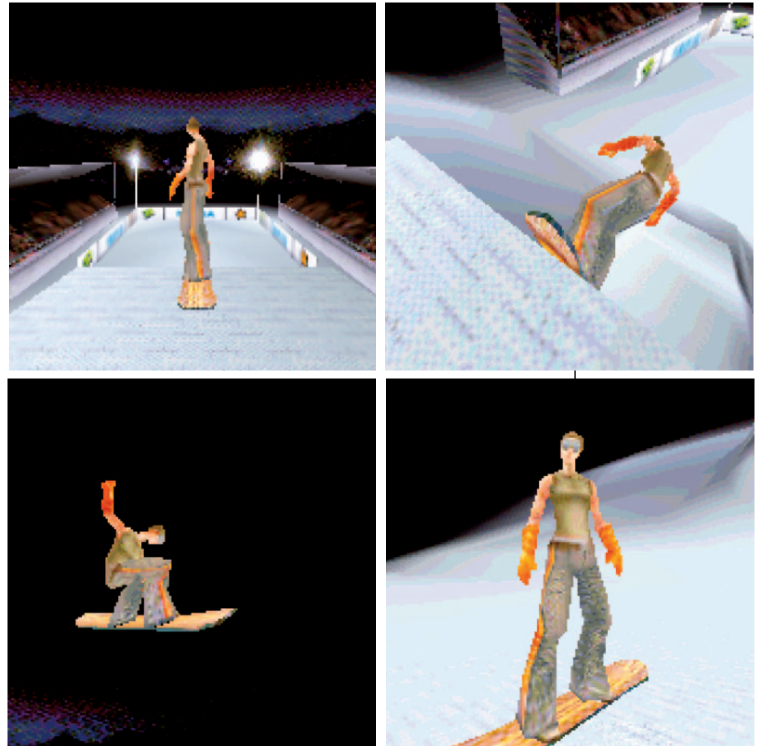
Given the rapid development of high-end graphics hardware and the growing importance of low-end software rendering, available devices will soon span a range of three orders of magnitude in terms of rendering performance. Designing an API to cater to such a variety of devices is clearly a challenge. Yet 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

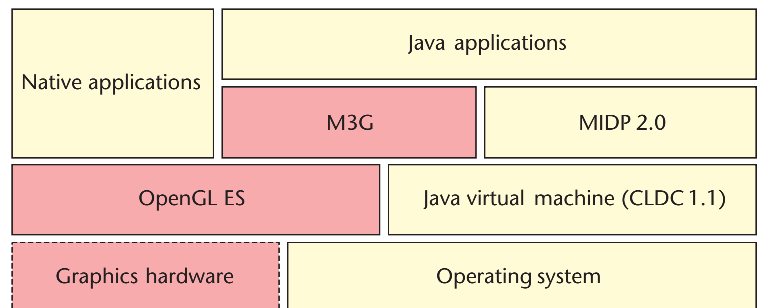
Mobile appliances, such as mobile phones, have traditionally been closed platforms in the sense that no new applications can be installed after the device is purchased. Another type of closed system is game consoles, where manufacturers carefully control who can develop and offer games for the system, 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.

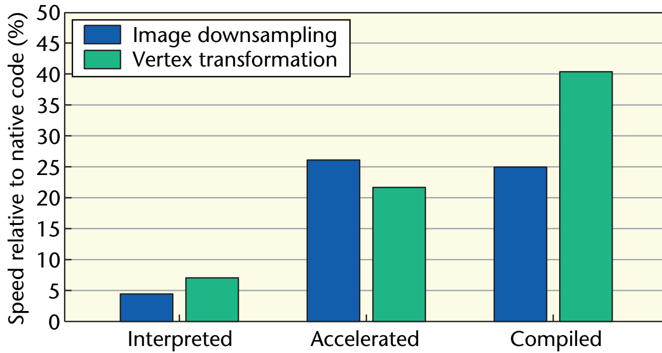
An open operating system allows installation of native applications, and anyone can, in principle, implement an efficient 3D software engine and the applications using it. What device manufacturers can do, however, is to define and implement a low-level API offered as an operating system service and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




**3 Execution speed of interpreted, hardware accelerated, and just-in-time compiled Java bytecode relative to native assembly code (the higher the bar, the faster the speed; native code is 100 percent). We ran the benchmarks on an ARM9 CPU equipped with the Jazelle bytecode accelerator.**

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

**Table 1. OpenGL 1.3 feature support in OpenGL ES 1.0.**

| Feature | Supported (IN), excluded (OUT), or optional |
|---|---|
| Vertex input and primitives | IN: arrays, points, lines, triangles<br>OUT: begin–end, polygons, quads |
| Vertex processing | IN: almost full OpenGL<br>OUT: user clip planes, texcoord generation |
| Lighting | IN: almost full OpenGL<br>OUT: back materials, local viewer, separate specular |
| Rendering modes | IN: double buffering, RGBA colors, rendering and access to back buffer<br>OUT: other variants |
| Raster processing | IN: ReadPixels<br>OUT: DrawPixels, bitmap |
| Texture mapping | IN: 2D texturing<br>OUT: 1D, 3D, borders, proxies, priorities, level of detail clamps<br>OPTIONAL: multitexturing, compression |
| Rasterization | IN: almost full OpenGL<br>OUT: polygon mode, polygon smooth, stipple |
| Fragment processing | IN: almost full OpenGL<br>OUT: separate specular<br>OPTIONAL: stencil buffer |
| Queries | IN: static queries (for example, matrix stack depth)<br>OUT: dynamic queries<br>OPTIONAL: matrices |
| Types | Lite: fixed point, integer, and byte for vertex data<br>Common: Common Lite plus floats |
| Convenience or rarely used modes | OUT: GLU utility library, evaluators, feedback mode, selection mode, display lists |

Java code is generally slower than native code, especially on mobile devices. This is mostly due to the virtual machine execution model and the built-in protection against illegal memory accesses and memory leaks. Our measurements on two representative graphics tasks show that native code can outperform Java by a factor of four even if just-in-time compilation or hardware acceleration is used (see Figure 3). This suggests that processing-intensive features are best served by adding them into the graphics API and implementing them in native code. Any functionality written in native code is effectively graphics hardware, as far as the Java application is concerned. The challenge in designing the API, then, is to find the right level of abstraction to accelerate common tasks without bloating the API and sacrificing flexibility.

Furthermore, Java midlets are commonly installed over a wireless connection, making short download times important. Providing high-level functionality such as scene graph and animation features in the graphics API, along with the ability to encode geometry, textures, and animation data into compact binary files, allows the final midlets to be smaller or include more application content.

### OpenGL ES

OpenGL provides a natural starting point for designing an open multiplatform low-level graphics API. OpenGL has stood the test of time since its introduction more than 10 years ago, has clear design principles, and was itself based on a previous API, the IRIS GL. During its lifetime, OpenGL has expanded, incorporating new functionality as new hardware was developed. DirectX is another popular, low-level 3D API, but it is only available on Windows and is therefore not a suitable basis for an open standard.

However, OpenGL in itself is too big for mobile devices. It needed to be cleaned of rarely used and redundant functionality and adapted to better address the restrictions of mobile devices. The Khronos Group set out to do these changes and create OpenGL ES, where ES stands for *embedded systems*. See Table 1 for a synopsis of version 1.0 and Table 2 for a list of version 1.1's new functionality (we discuss the Common and Lite profiles later).

Having the API resemble OpenGL closely enough brings many benefits: The standardization process can progress faster, the experiences of the strong and weaker points of OpenGL can be brought to bear more easily, and the wide availability of OpenGL literature eases the adoption of the API by the developers.

### M3G

The idea of subsetting and adapting a high-level API to fit the J2ME environment seemed attractive for the same reasons as with OpenGL ES. An obvious candidate for this was Java 3D. Java 3D is a high-level API that provides extensive support for scene graph operations and some support for animation and compressed geometry. Java 3D is based on earlier scene graph frameworks such as Open Inventor and VRML, and can be implemented on top of OpenGL. Therefore, a subset of Java 3D could probably be designed so it can be implemented on top of OpenGL ES, allowing both APIs to benefit from the same underlying rendering engine. However, after the JSR-

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 same core functionality as Java 3D, but avoids the redundant methods and overly generalized class hierarchies that hamper Java 3D. As a result, the API footprint (the number of classes and methods) is an order of magnitude smaller than that of Java 3D. See Table 3 for an overview of M3G.

## Mobile 3D design goals

The aim for this work was not to revolutionize 3D graphics APIs. Instead, the target was to produce APIs that are as useful and usable as their desktop counterparts, yet allow very compact and efficient implementation on mobile devices, with or without graphics hardware or floating-point support. The main design goals were

- performance,
- low engine complexity,
- application size,
- hardware support,
- ease of adoption,
- productivity,
- orthogonality of features,
- extensibility, and
- minimized fragmentation.

Rendering speed is of utmost importance in a graphics system whose main use case is gaming. Given the wide variation in device capabilities, it's not meaningful to set concrete fill rate or polygon count targets. Instead, a worthy goal is to minimize the penalty that an application using the API will have to pay, compared to the theoretical case of accessing the hardware directly.

The constraints on silicon area and memory consumption in mobile devices are stringent. Devoting millions of transistors or megabytes of memory to the graphics subsystem is not feasible with current technology. To reflect the need to reduce engine complexity, the code size targets for OpenGL ES 1.0 and M3G were set to 50,000 and 150,000 bytes, respectively. Although no transistor count targets were set, minimizing the size of software implementations generally translates to simplified hardware, as well.

Similarly to the graphics engine, the applications using that engine need to be small, as well. Applications are often downloaded over slow cellular networks, and many users are not willing to tolerate long download times.

Standards with proven hardware-friendly features give hardware vendors a clear design goal. It's important that the low- and high-level APIs complement rather than compete with each other, and support a similar enough hardware model so that the same hardware can accelerate both of them.

Ease of adoption is important because a programming interface is not useful if no one implements it, or if developers choose not to use it. That might happen if the API is inappropriate for the environment, encumbered with patents, not supported by development tools, or not promoted enough. Open standardization is crucial in avoiding these pitfalls.

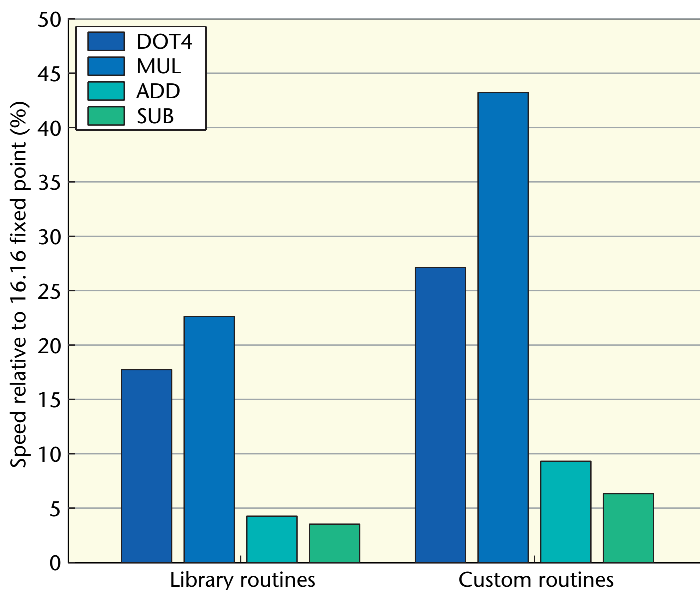
The productivity benefits of decoupling art assets from application code have long been recognized in the game development community. A robust tool chain is required to allow graphic designers to work independently of programmers. The feature set must also be in line with current desktop and console technology so that established parts of the production pipeline are usable for mobile content as well.

**Table 2. New features in OpenGL ES 1.1.**

| Feature | Purpose |
|---|---|
| Buffer objects | Enable server-side vertex data |
| DrawTexture | For blitting (copying and displaying) image data from server-side memory |
| Enhanced texturing | A minimum of two units, with texture combiners and bump mapping |
| Matrix palette | Vertex skinning for animation |
| User clip planes | Portal culling |
| Point sprites | Enable fast particle systems |
| Dynamic state queries | Enable middleware state save and restore |

**Table 3. M3G functionality.**

| Functions | Purpose |
|---|---|
| Rendering engine | Transforms, viewing, lighting, materials, texturing, and fragment operations similar to OpenGL ES 1.0 Common profile |
| Scene graph | World root node, active camera, lights, hierarchic transformations, rendering detached branches |
| Geometry | Vertex arrays specified as 8- or 16-bit integers with optional scale and bias in floating point, meshes share vertex arrays, submeshes index the arrays, triangle strips only |
| Appearance | Any combination of the following per submesh: 2D textures, material colors, fog, blending, alpha testing, depth buffering, culling, shading |
| Raster imagery | Sprites that optionally scale based on distance to the viewer; background images that can be zoomed, scrolled, and tiled |
| General animation | Keyframes (scalar, vector, orientation) for any property; step, linear, spline, slerp, squad interpolation |
| Advanced animation | Bone hierarchies and vertex skinning, morphing meshes to weighted targets, weighted blending of animation tracks, nodes can automatically align to others |
| Execution model | Synchronous; no callbacks; explicit control of animation, rendering, world time |
| File format | Compressed, can encode all API objects |




**4 Execution speed of floating-point arithmetic relative to 16.16 fixed point in primitive scalar operations and the 4-element dot product (the higher the bar, the faster the speed, where the fixed point is 100 percent). The library routines are from the ARM RVCT 2.1 compiler. The custom routines omit proper handling of infinities and not-a-number values, flush denormals to zero, and don't always round properly. We ran the benchmarks on an ARM7 simulator.**

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 effects, regardless of the API's level of abstraction.

There has been a constant evolution of graphics APIs during the past decade: APIs that have not been able to add new features have vanished. Extensibility is especially important for mobile APIs since the devices are evolving more rapidly than on the desktop.

Embracing extensions and allowing many optional parts makes it difficult to create portable content. Both APIs were developed so as to minimize the number of optional features. Some leeway in implementation is useful, but preferably so that only image quality is affected, without loss of features.

## Key design decisions

The two APIs were designed to stand on their own, but it was crucial that they be complementary and share resources. Both APIs have to be able to use the same rendering engine, whether it is implemented in software or hardware. As Figure 2 shows, the target was that you should be able to implement M3G on top of OpenGL ES. M3G, in turn, was designed so that its feature set aligns with established OpenGL concepts.

In practice, OpenGL ES Common profile provides all of the low-level transformation, lighting, and rasterization functionality for M3G, while M3G adds the scene graph and animation features on top of that. Some constructs in M3G don't have direct equivalents in OpenGL ES, notably the Sprite and Background classes. This allows software implementations to include optimized versions of them in the rasterizer, while hardware implementations can efficiently render them as textured quadrilaterals. OpenGL ES provides maximum performance and flexibility for native applications; M3G adds features necessary for efficient Java application development.

### *The decimal point: To fix or to float*

Modern 3D graphics programming is based on floating-point arithmetic. For applications, middleware, and vertex pipelines this has been the standard mode of operation; with the introduction of high-level shading languages, the trend is now toward floating-point values even in the fragment pipeline and the frame buffer.

Floating-point representation is popular because of its high dynamic range. Accuracy is not really an issue. Integers are in fact more accurate within their domain, but unfortunately that domain is limited, easily resulting in overflows and underflows. Depending on where the overflow occurs, the end result might be a small visual artifact in the best case, or a completely corrupted scene in the worst case.

The problem with current mobile devices is that they have no hardware support for floating-point arithmetic. As Figure 4 shows, this can result in an order of magnitude performance gap between integer and floating-point operations. Striking a balance between the speed of integers on one hand and the ease-of-use of floats on the other hand is one of the most important decisions in designing a mobile graphics system.

In a controlled situation with restricted input and output ranges, various shortcuts can speed up execution without compromising robustness. In particular, rasterization and per-fragment processing in a traditional rendering pipeline can be done entirely using fixed-point arithmetic, thanks to the limited size and color depth of textures and the frame buffer.

The vertex pipeline is more problematic, however. If the coordinates are given as floating-point values that reside in application memory, there is little room for optimization. The values must be treated as unrestricted unless inspected one-by-one, and worse, caching the results of such inspection is impossible, because the array contents might change between invocations. The best that can be done is to use custom floating-point routines that dispense with special cases, which can accelerate primitive operations by 50 to 100 percent over library routines (see Figure 4).

On the other hand, if vertex arrays are stored in server-side buffers to which the application has no direct access, the implementation might preprocess the arrays, for example, to select an appropriate execution path or convert the values to a more convenient format. Depending on the actual value ranges, the internal format might be, for example, fixed point with a common exponent for components of each vertex, fixed point with a common exponent for the whole array, or 24-bit floating point, which is guaranteed to fulfill the OpenGL range and accuracy requirements. Of course, if the vertex arrays are given in integer or fixed-point format to

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.

**OpenGL ES solution.** The OpenGL ES Common profile replaces all doubles with floats and introduces a 16.16 fixed-point data type, where the first 16 bits denote a signed two's complement integer, and the other 16 bits form the fractional part. Matrices and vertex arrays can be in floating-point format, but applications using fixed-point vertices will witness much better performance on current implementations. Furthermore, the Common profile requires vertex transformations to respect OpenGL's requirements for dynamic range and accuracy. This practically guarantees that no overflows will occur in the middle of the pipeline, providing ease of use in both programming and modeling. For extremely limited environments there is also the Common Lite profile that banishes floats altogether, and does allow vertex transformations to overflow.

**OpenGL ES provides maximum performance and flexibility for native applications; M3G adds features necessary for efficient Java application development.**

**M3G solution.** M3G uses standard floats as the input and output format everywhere except the vertex arrays. Vertex arrays are given as integers, coupled with scale and bias terms for mapping the integers into the full floating-point range; this can be implemented with the model-view matrix, so there is no per-vertex cost.

The solution adopted in M3G is based on the assumption that the bulk of the floating-point processing would be for vertex transformations, while everything else incurs so little computational cost that it is outweighed by the ease of use and reliability provided by floats. Using floats for matrices and restricting vertex data to be integers allows for a high dynamic range with reasonable accuracy for vertex transformations, yet executes much faster than if the vertices were floats as well. These productivity enhancing factors are extremely important in Java, because midlets are typically developed on tighter budgets and schedules than native applications, and they will have to run on a wider variety of devices.

### *Fixed-function pipeline*

The graphics hardware on desktops nowadays supports programmable shaders in both the vertex and the fragment pipeline, and there are assembly-level extensions to OpenGL and Direct3D, as well as higher level languages for programming the shaders. The standardization groups considered adopting those already in the first generation of mobile 3D APIs. This would have allowed us to get rid of fixed-pipe functionality such as the lighting model, making the API more compact and even simplifying some hardware implementations.

However, evolution on the desktop indicates that high-level C-like shading languages with full floating-point processing are becoming the standard. This presents developers with fundamentally more power and flexibility than the limited numeric range of the early assembler-like shaders. Hence, standardization groups perceived that the mobile graphics APIs should align with the desktop model, rather than attempting to reinstate low-level fixed-point programmability at this point.

Sidestepping the code size implications of a shader compiler and the existence of fixed-function hardware that also must be supported, the major problem with programmable shading was eventually performance: With floating-point shading, almost none of the optimizations possible in a fixed-function pipeline can be applied, resulting in unacceptable performance on platforms without dedicated hardware.

In the end, the standardization groups decided that the first generation of mobile 3D APIs is better served with a fixed-function graphics pipeline. A compact fixed-function API facilitates the design of compact energy-efficient hardware accelerators. Shaders are introduced in OpenGL ES 2.0, which directly caters to programmable graphics hardware.

### *Full fragment processing*

Because OpenGL does not allow you to inject data in the middle of the pipeline, emulation of fragment processing features in application code would be hard and inefficient. For this reason, almost all of the OpenGL fragment-processing features were kept, and reductions concentrated on the front end of the pipeline.

Implementing a 3D software engine that does efficient fragment processing for all possible rendering state combinations is nontrivial. A straightforward implementation is to provide a single fragment program that is used for drawing all triangles. Because current embedded CPUs suffer a large performance hit from branching, a long if-else type of fragment program is not optimal.

Another alternative is to write manually optimized assembly language versions for the fragment programs that get used most often and provide a program that can process all the other combinations as a fallback. The problem is that the engine size will grow quickly, yet many fragment pipeline combinations remain slow. A more efficient solution to the problem is employed, for example, in the Pixomatic software rasterizer, where the fragment pipeline is generated on the fly. A similar method called *stitching* was used at the microcode level in InfiniteReality.[2]

With simple optimizations on the generated code, the performance of all rendering state combinations is close to a hand-optimized implementation. Also, as program constructs needed for fragment programs are rather simple, the code generator and optimizer can be made small enough for embedded use.

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—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

**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.**

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.

**Flexible keyframing.** 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.

**Character animation.** 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.

**Application control.** M3G leaves the application in full control of execution. Both animation and rendering are invoked explicitly from the application, without imposing any specific execution or event handling model. Rendering and animation are completely independent of each other and can run at different frequencies, if required.

### *Standard file format*

A significant amount of code in a game is usually devoted to loading game levels and other art assets. M3G eases that burden by providing a standard file format and a built-in loader. The file format allows any M3G objects to be stored in a compressed binary stream that can be loaded into an application with a few lines of code. The format is flexible enough to contain anything from complete scene graphs to individual models or components. The built-in connectivity of J2ME also makes over-the-air downloading as easy as loading from the Java Archive (JAR) that contains the application.

**M3G provides a standard file format that allows any M3G objects to be stored in a compressed binary stream that can be loaded into an application with a few lines of code.**

The compression used in the M3G format is based on delta encoding (which stores only the differences of consecutive values rather than the values themselves) followed by zlib compression (see http://www.gzip.org/zlib/). Zlib is already used by J2ME for JAR and PNG images, and, in practice, delta encoding followed by zlib proved more effective than more sophisticated bit-packing followed by zlib.

Another benefit of the file format is that it encourages the decoupling of content from the application logic. This enables artists to experiment and iterate without programmer intervention, as art assets can be changed without touching the code. To make this more robust, M3G has a built-in mechanism for finding named objects from a scene graph; each object has a 32-bit ID number that can be set in a content tool and used to access the object in the application, even if the object is relocated in the scene graph hierarchy.

### *Cross-platform window system bindings*

In desktop OpenGL, window bindings and management of other resources are not specified in the core API, but there is a different API for each platform (for example, GLX for the X Window System and WGL for Microsoft Windows). Although these APIs have similar functionality, their function names are different, and each API has some unique features. In the handheld device market, many different operating systems are in

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 operating system or device vendor to specify how abstract rendering surface types are mapped to concrete types of a given platform. Even though the actual data types are different for each operating system, API concepts and function calls are the same, making application porting easier.

### Mixed 2D and 3D rendering

Mobile Java offers a relatively rich set of 2D graphics rendering functionality for downloadable mobile Java games. Because 3D games might still use 2D graphics, for example, for user interface elements or backdrop images, the M3G API allows applications to take advantage of the existing functionality.

The main issue in mixed 2D and 3D rendering is the synchronization between two different renderers. One of those could be accelerated, while the other is completely in software, and given the multitude of supported rendering target types, they might even require different frame buffer formats in some cases.

M3G addresses this by making these issues explicit. All 3D rendering must begin and end with dedicated bindTarget and releaseTarget method calls, and synchronization between 2D and 3D only occurs at those points—the result of drawing 2D while a target is bound for 3D rendering is undefined. A rendering target can be bound and released as many times per frame as necessary, but it's clear to developers that each time comes with a potential drop in performance.

The API itself does not define the types of rendering targets supported, but accepts a generic Java object that is checked for compatibility inside the implementation. This makes it trivial to accept new rendering target types in future revisions and enables the API to work with Java profiles other than Mobile Information Device Profile, which is by far the most common.

## Discussion and conclusion

We believe that the OpenGL ES and M3G open standards will make 3D graphics available on mass-market mobile devices, as well as fuel advances in mobile graphics hardware.

The OpenGL ES and M3G specifications were released in July and November of 2003, respectively. As of this writing, at least 10 vendors have implemented these APIs (either software or hardware). The first mobile devices having built-in support were launched in 2004.

The APIs were designed so that software implementations on integer-only processors are as efficient as possible, while still allowing high-end devices with graphics hardware to fully leverage the capabilities of that hardware. The positive market reaction suggests that the compromises made were the right ones.

As for M3G, the key goal in its design was to overcome the performance penalty caused by Java virtual machines. This was achieved by raising the abstraction level of the API above that of OpenGL, allowing the implementation to apply more aggressive optimization on object and scene data structures, and applications to minimize the amount of slow Java code required for common animation and rendering tasks. This will become even more important with the introduction of graphics hardware.

The graphics architecture presented here takes full advantage of the capabilities of mobile devices today and in the next few years, and paves the way for more advanced devices that lie further ahead. Links to the standards we have covered or mentioned can be found in the "Further Information" sidebar. ■

### Further Information

For more information on the standards discussed in this article, visit the following Web sites:

- OpenGL ES, EGL
  http://www.khronos.org/opengles/spec.html
- OpenGL, GLX, OpenGL SL
  http://www.opengl.org/documentation/
- M3G
  http://www.forum.nokia.com/java/jsr184/
- DirectX and HLSL
  http://www.microsoft.com/windows/directx/
- VRML
  http://www.web3d.org/x3d/specifications/vrml/
- CG
  http://developer.nvidia.com/page/cg_main.html
- Java 3D
  http://java.sun.com/products/java-media/3D/
- Pixomatic
  http://www.radgametools.com

### Acknowledgments

Although we were closely and actively involved in specifying these APIs, the design was the joint work of the members of the OpenGL ES Working Group and the JSR-184 Expert Group. We thank everybody who contributed to the definition of the APIs. Especially active in the work were David Blythe, Mark Callow, Graham Connor, Sean Ellis, Claude Knaus, Jon Leech, Tom McReynolds, Ville Miettinen, Aaftab Munshi, Tom Olsen, Mark Patel, Ed Plowman, Jacob Ström, and Doug Twilleager. Neil Trevett and Jyri Huopaniemi provided excellent chairmanship of the standards. We also thank implementation teams at Nokia and elsewhere; getting feedback from real implementations during standardization was crucial to a successful end result.

## 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 Technology. He has been an active contributor to several mobile graphics standards and steers most graphics research activities at Nokia. Pulli has a BS in computer science from the University of Minnesota; a Lic.Tech. and an MS in computer engineering and an EMBA from the University of Oulu, Finland; and a PhD in computer science from the University of Washington. Contact him at kari.pulli@nokia.com.*

***Tomi Aarnio*** *is a senior research engineer at Nokia Research Center. His research interests include real-time rendering in general, compression of 3D assets, and Java virtual machine technology. Aarnio has an MS in computer science from the University of Turku, Finland. He is the editor of the M3G (JSR-184) standard. Contact him at tomi.aarnio@nokia.com.*

***Kimmo Roimela*** *is a senior research engineer at Nokia Research Center. His research interests include game technology and 3D graphics in general, and their application to constrained platforms in particular. Roimela has an MS in computer science from Tampere University of Technology, Finland. He was actively involved both in the design and implementation of the M3G API. Contact him at kimmo.roimela@nokia.com.*

***Jani Vaarala*** *is a graphics architect at Nokia. Vaarala has an MS in computer science from the University of Oulu, Finland. He was actively involved in the OpenGL ES standardization and implementation. Contact him at jani.vaarala@nokia.com.*



# Call for General Submissions

*IEEE Computer Graphics and Applications* magazine invites original articles on the theory and practice of computer graphics. Topics for suitable articles might range from specific algorithms to full system implementations in areas such as modeling, rendering, animation, information and scientific visualization, HCI/user interfaces, novel applications, hardware architectures, haptics, and visual and augmented reality systems. We also seek tutorials and survey articles.

Articles should up to 10 magazine pages in length with no more than 10 figures or images, where a page is approximately 800 words and a quarter page image counts as 200 words. Please limit the number of references to the 12 most relevant. Also consider providing background materials in sidebars for nonexpert readers.

Submit your paper using our online manuscript submission service at http://cs-ieee.manuscriptcentral.com/. For more information and instructions on presentation and formatting, please visit our author resources page at http://www.computer.org/cga/author.htm.

Please include a title, abstract, and the lead author's contact information.

IEEE Computer Graphics and Applications™