COMPARING M3G AND JSR-239 FOR 3D GAMES PROGRAMMING

Andrew Davison and Sophie Radenahmud

Dept. of Computer Engineering
Prince of Songkla University
Hat Yai, Songkhla 90112, Thailand

E-mail: ad@fivedots.coe.psu.ac.th

KEYWORDS

M3G, JSR-239, OpenGL-ES, Java, 3D, games, comparison

ABSTRACT

We compare two graphics APIs for programming 3D games in Java on mobile devices: M3G (Mobile 3D Graphics API for Java, JSR-184) and JSR-239 (a Java binding for OpenGL ES 1.x). We have developed a series of casual games (a puzzle game, a simple FPS, a strategy game, and others) using the versions of M3G and JSR-239 available in Sun's Wireless Toolkit 2.5.1, and use them to compare the APIs in three areas: suitability for casual games programming, ease of performance tweaking, and API and Java integration.

INTRODUCTION

Mobile gaming is in the midst of a minor revolution, driven by advances in graphics processing (FPUs, GPUs), new interfaces (high definition video, touch, spatial sensors), and improved networking (GPS, faster wireless). These improvements are encouraging new forms of gaming to appear, relying on persistent network connectivity, mobility, parameters derived from the physical environment, and the episodic nature of communication. 3D graphics are closely linked to these themes as a means of representing and enhancing the user's interface to the 3D world.

But there are problems: device fragmentation is accelerating, with an ever widening range of CPU speeds, memory sizes, screen resolutions, color depths, power consumption, non-standard user interfaces, audio and video types, and variations in hardware support for floating point numbers and advanced graphic processing (such as shaders).

The demands placed on a 3D graphics API are substantial: one major issue is abstraction versus control – how much of the growing hardware and software complexity should be exposed to the programmer? Another question is how the API should deal with device variety: should it attempt to fall back on (slow) software emulation when necessary hardware is absent, or should it simply refuse to work?

Java ME is currently the most popular programming language for mobiles (it has been estimated that 1200 million phones were running Java in 2006 (Evans et al. 2006)), so how should a 3D API be integrated with Java, and with its myriad extensions, especially those related to graphics?

We consider three broad API issues in this paper: suitability for casual games programming, ease of performance tweaking, and how the API and Java are combined. We do this by referring to our experience in developing a range of casual games using M3G (Mobile 3D Graphics API for Java, JSR-184) and JSR-239 (a Java binding for OpenGL ES 1.x) in Sun's Wireless Toolkit 2.5.1. We also discuss what the future holds for M3G and JSR-239 (i.e. in M3G 2.0 and OpenGL ES 2.0).

CASUAL GAMES

Casual gaming is ideally suited to mobile devices: simple gameplay, action in short bursts, no long-term time commitment or special skills required, and comparatively cheap game production. Casual gaming is also appealing to people outside the usual gamer demographic, a potentially massive audience.

Our examples include a puzzle game (CubeFinder), a simple FPS involving penguins, a strategy game (Tic-Tac-Toe), written in both M3G 1.1 and JSR-239. We have also implemented a number of smaller MIDlets to test different aspects of the APIs: a model viewer, an animated scene, a particles

demo, and examples using skinning and morphing. Figure 1 shows screenshots from some of the games and demos.

Figure 1. Casual Games and Demos

These M3G and JSR-239 examples can be found at the first author's websites: http://fivedots.coe.psu.ac.th/~ad/jg/ and http://fivedots.coe.psu.ac.th/~ad/jg2/.

These applications were developed by a programmer new to M3G and JSR-239, and our comments here are based on his logbook notes. Our intention was to gain the perspective of a programmer new to these APIs, albeit one with several semesters experience of Java and desktop OpenGL programming.

AN OVERVIEW OF JSR-239

JSR-239 is a Java binding around OpenGL ES (OpenGL for Embedded Systems) which is itself a subset of OpenGL aimed at smaller devices such as mobile phones, PDAs, and games consoles (see http://jcp.org/en/jsr/detail?id=239 and http://www.khronos.org/opengles/). It is small (around 50 KB), and yet its capabilities are very similar to OpenGL's.

The most obvious loss of functionality is probably the OpenGL glBegin()/glEnd() technique for grouping instructions for shape creation. In OpenGL ES, the programmer defines arrays for a shape's vertices, normals, colors, and texture coordinates (Astle and Durnil 2004).

Another significant loss are the GLU and GLUT utility libraries. GLU includes convenience functions for such tasks as positioning the camera, setting up the viewing volume, generating basic shapes, and texture mipmapping. GLUT is mainly utilized in OpenGL applications for its I/O support; in JSR-239 that is handled by Java ME's GameCanvas.

OpenGL ES differs from OpenGL in its support for fixed-point numbers in addition to floats, to better match the limited hardware of smaller devices. Its 16.16 fixed-point data type utilizes the first 16 bits for a signed two's compliment integer, and the other 16 bits for a fractional part. A shape defined using fixed-point vertices should render much more quickly than one employing floats.

OpenGL ES only has primitives for creating shapes out of points, lines, or triangles; polygons and quadrilaterals (quads) primitives are missing.

OpenGL ES is a 'moving' specification, with three incarnations at the moment. OpenGL ES 1.0 is based upon OpenGL 1.3, OpenGL ES 1.1 is defined relative to OpenGL 1.5, and OpenGL ES 2.0 is derived from the OpenGL 2.0 specification.

OpenGL ES 1.1 includes support for multi-texturing, mipmap generation, and greater control over point rendering (useful for particle systems). OpenGL ES 2.0 is a more radical departure, employing a programmable rendering model based around shaders, with only floating point operations. The motivation behind this design is the belief that mobile devices will very shortly have the rendering power of today's desktop and laptop machines (Munshi et al. 2008).

The GLBenchmark site (http://www.glbenchmark.com/result.jsp) includes a long list of OpenGL ES devices (including Symbian devices, the PlayStation 3, Nintendo's GameBoy DS, and the iPhone), and their results against its benchmarking software. As of August 2008, only a few high-end Sony Ericsson phones support JSR-239 (Hellman 2008).

AN OVERVIEW OF M3G

M3G (the Mobile 3D Graphics API) was developed as JSR 184 (http://www.jcp.org/en/jsr/detail?id=184), and is currently at version 1.1. Version 2.0 is in development as JSR 297, with the aim of adding programmable shaders and other advanced features (http://jcp.org/en/jsr/detail?id=297).

M3G provides two ways for developers to draw 3D graphics: immediate mode and retained mode. In immediate mode, commands are issued directly into the graphics pipeline in a similar way to OpenGL ES. However, it's most common for programmers to utilize retained mode which employs a scene graph to link the geometric objects in the 3D world into a tree structure, and specify the camera, lights, and background (Höfele 2007).

At the lowest level, M3G deals with concepts similar to those in OpenGL ES, but the scene graph combines and hides these features inside higher-level graph nodes. For example, vertex and index buffers are combined into Mesh objects; textures, materials, and other rendering parameters form Appearance objects, and Group nodes hierarchically combine transformations.

M3G offers keyframe animation that can be attached to almost any property of any object. It also supports vertex deformation through morphing and skinning. There is a compact, binary M3G file format that can store anything from complete 3D animations and scene graphs down to individual objects or their components.

M3G can be implemented on top of OpenGL ES (although this is not a requirement), and the resulting relationship between M3G, OpenGL ES, and Java ME is represented in Figure 2.

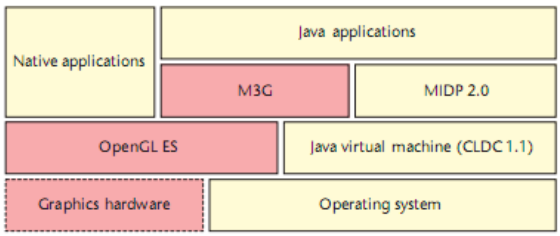

Figure 2. The Java ME 3D Programming Layers.

The JBenchmark site (http://www.jbenchmark.com) contains a long list of M3G compatible mobile devices, and performance data.

SUITABILITY FOR CASUAL GAMES PROGRAMMING

M3G's scene graph makes programming much easier for novices (and even for experienced programmers) because it emphasizes scene design rather than rendering, by hiding the underlying graphics pipeline. A scene graph naturally supports complex graphical elements such as 3D geometries, the camera, and lighting.

At the implementation level, the scene graph can be employed to group shapes with common properties, carry out frustum culling, scene management, level of detail selection, execution culling, and batching of graphics operations – all optimizations which must be coded directly by the programmer in OpenGL ES. However, it is unclear whether different phone manufacturer's M3G implementations actually support these optimizations (Pulli et al. 2005).

A very common coding requirement in current 3D games is to mix-and-match 3D and 2D, utilizing 2D images for overlays, backgrounds, and game characters. The switching between modes that this entails can slow down rendering by as much as three times (Pulli et al. 2007a), and so it's necessary to structure code so that mode switching is minimized. In M3G, the Appearance node possesses a layers mechanism for ordering rendering. For instance, it's easy to specify that overlays are drawn first, followed by objects further back in the scene. M3G's 3D rendering is based on a bind-render-release sequence, which makes it clear when 3D rendering begins and ends, and so makes it much harder to inadvertently mix 2D and 3D processing.

The M3G bind-render-release mechanism will also be useful for integrating 3D rendering with other types of 2D processing, such as GUIs, vector graphics, and streaming media (e.g. LWUIT, JavaFX mobile, OpenVG, OpenMAX) (Petroshinko et al. 2007).

M3G also provides OpenGL ES-like immediate mode rendering, suited for special effects or when the application needs more control over the rendering process. The same data objects are used for both retained and immediate mode rendering, so the two can be interleaved.

OpenGL ES 's low-level nature means that a programmer must write much more 'boiler-plate' code, and reinvent common library functionality before getting things to work. This includes the implementation of a mobile

camera, animation, skinning, morphing – all features offered directly in M3G.

One of M3G's design principles is the ease of content creation: the M3G file format is simple to decode, offers compression, matches the API directly, and can be employed to load objects or even entire, animated, scenes. The format is supported by all the main modeling tools, including 3d Studio Max, Maya, and Blender. Also, each object in an M3G file can be given a unique ID number for easy access by the MIDlet. Additional user-specific data can be associated with each object (Aarnio et al. 2007). The OpenGL ES specification does not support any model format.

The initialization of a scene in M3G is a matter of graph building, while a state machine (the EGL) must be configured in OpenGL. This often requires a knowledge of the underlying device hardware in order to get the settings correct.

The next iterations of M3G and OpenGL ES will significantly effect programming: M3G 2.0 will continue to support the fixed function graphics pipeline but add programmable shaders as optional extras. OpenGL ES 2.0 will break with the past and only offer shaders, requiring existing applications that use the fixed graphics pipeline to be modified to employ shaders to perform transformations, lighting, texture blending, alpha testing, bump mapping, coloring, and fog. OpenGL ES 2.0 will not offer a software fallback if the platform's hardware is insufficient (Ginsburg 2006). M3G, by contrast, makes a point of not mandating any hardware features, but at the expense of having 20% more classes than a pure shader version of the API (Pulli et al. 2007b).

PERFORMANCE TWEAKING

Since M3G can be viewed as a high-level abstraction over OpenGL ES (see Figure 2), many of the well-known performance tweaks, tips, and tricks recommended for OpenGL ES can also be applied to M3G.

Complex models should be simplified due to the small screen size and lower resolution of mobile devices. Scaling involving floating point calculations should be avoided: instead the model should be correctly sized before being imported into the game (Leal 2008).

Floating point operations should not be utilized on devices without FPU support; OpenGL ES's fixed point notation is an effective substitute for many tasks, and it's a shame that this type is missing from M3G. However, Java ME fixed point libraries are available (e.g. FPLib, http://www.geocities.com/andre_leiradella/#fplib), and FPComp translates fixed point library calls into inline code, producing drastic speedups (de Leiradella 2004).

Special effects, such as particle systems and background elements, should be curtailed.

Unless lighting is very simple (i.e. a single directional light), it may be better to replace it with light maps or bump mapping. At the very least, expensive lighting effects, such as specular illumination and distance attenuation, should be turned off. Simplified lighting may produce performance boosts of 50% (Wright 2006).

Mipmaps always help performance, and are created automatically in OpenGL ES 1.1 and M3G. Multi-texturing is better than multipass rendering, and texture resolution should be reduced on small devices.

Dynamic geometry is very expensive, but M3G offers cheaper alternatives via skinning and morphing.

More advanced optimizations tend to require a good knowledge of the underlying hardware, and produce varying results across different platforms. For example, it is almost certainly better to use fixed point numbers to define vertices, but the performance gain over floats should be measured. On some platforms, fixed-point values are converted to floats before processing, and so may actually be slower than using floats directly (Wright. 2006).

Another performance trick is to replace large meshes by multiple smaller meshes, which may allow them to be culled when out of sight, but it may also increase the rendering time costs.

It may be better to combine textures (i.e. replace four 128x128 textures by a single 256x256 image) to take advantage of texture compression and to avoid switching between textures at render time. However, the frame rate must be examined to determine the real benefit.

API AND JAVA INTEGRATION

One of M3G's key design principles is to avoid the use of Java for any critical graphical operations – all graphics processing is passed to native code, including morphing, skinning, and keyframe animation. In addition, all

vertices and indices data is stored outside of Java. This is in response to speed measurements of Java virtual machines on mobiles against assembly code. Native code is usually 10-20x faster than the KVM, the most common VM on mobile phones (Pulli et al. 2007b). Hardware accelerators, such as Jazelle from ARM and the HotSpot VM from Sun are better performers, but native code is still 3-4 times faster.

These problems mean that JSR-239 code must utilize some rather advanced OpenGL ES features to avoid Java's slowness. For example, data should be stored in VBOs (Vertex Buffer Objects) so that the data is passed over to video memory, thereby bypassing Java and reducing bandwidth requirements. However, this strategy requires careful testing in larger games since it's quite possible to overload GPU memory with too much data. Another approach is to render into textures using PBuffers (Pixel Buffers) to ensure that rendering is done natively. PBuffers also are useful for special effects such as motion blur, light blooms, and fluid visuals.

Both M3G and JSR-239 use a simple rendering loop in Java, something like:

```
initialize the graphics engines;
initialize the 3D scene;
while (isRunning) {
  process any user input;
  update the application state;
  draw the scene (3D and 2D);
  perhaps sleep a while to
    maintain the frame rate;
}
shutdown the 3D graphics engine;
```

M3G offers a few variations on this approach, using either a MIDP 1.0 Canvas or a MIDP 2.0 GameCanvas as a rendering surface. It's also possible to trigger redraws using a Timer.

The JSR-239 code for the initialization steps is considerably longer than the M3G version due to the need to configure the graphics state machine. Also, some care must be taken to carry out all graphics state operations inside a single thread (a subtlety that also catches out JOGL programmers using OpenGL).

CONCLUSIONS

Our experience with using M3G and JSR-239 for 3D games programming has highlighted numerous differences between them, which can be grouped under three headings: suitability for casual games programming, ease of performance tweaking, and API and Java integration.

M3G is much better suited to casual gaming than JSR-239 because of its retained mode (scene graph) mechanism, which hides a great deal of low-level graphics detail, while performing optimizations such as frustum culling and batch processing. If necessary, M3G's immediate mode can be used to 'peer behind the curtain'.

This two-tier approach will continue in M3G 2.0, which will offer both a fixed function pipeline and programmable shaders. This contrasts with JSR-239's design principles (actually OpenGL ES's principles) which aim for full programmable access to the graphics pipeline, with no fallback to software emulation. This position makes sense in the long term, but for the next few years there will be many phones utilizing only a fixed graphics pipeline.

A great deal of simple performance tweaking can be achieved in M3G and OpenGL ES by reducing model complexity, texture resolution, lighting effects, and utilizing multi-texturing. More complex optimizations are possible in OpenGL ES, but they require careful profiling of their effectiveness, and a good understanding of the underlying hardware.

M3G and OpenGL ES offer a similar interface to Java based on an update-render-wait loop. However, the coding details for M3G are simpler since the manipulation is of a scene graph rather than a state machine. Also, M3G's bind-render-release sequence for 3D processing and layered Appearance nodes makes it much easier to integrate 3D with 2D and other graphics APIs.

REFERENCES

Aarnio, T., K.Roimela, and K.Pulli. 2007. "M3G: Bringing 3D Graphics to Mobile Java", *Power Management*, Vol. 5, No. 7, November/December.

Astle, D. and D.Durnil. 2004 *OpenGL ES Game Development*, Course Technology, PTR.

de Leiradella, A. 2004. "Optimizing Fixed Point (FP) Math with J2ME", http://www.devx.com/Java/Article/21850, Last accessed September 3rd 2008.

Evans, J., N.Ramani, and A.Bhanushali. 2006. "Developing Java Platform, Micro Edition Graphical Applications to Take Advantage of Hardware Acceleration", *JavaOne Conference*, TS-3024,

http://gceclub.sun.com.cn/java_one_online/2006/TS-3024/TS-3024.pdf

Hellman, E. 2008. “New Gaming Experiences with OpenGL ES and the Mobile Sensor API”, *Sun Development Network*, April, http://developers.sun.com/mobility/apis/articles/opengles_mobilesensor/

Höfele, C. 2007. *Mobile 3D Graphics: Learning 3D Graphics with the Java Micro Edition*, Course Technology PTR.

Ginsburg, D. 2006. “OpenGL ES 2.0: Shaders Go Mobile”, *Game Developers Conference*, San Diego, USA, http://ati.amd.com/developer/gdc/2006/GDCMobile2006-Ginsburg-OpenGLES2.0.pdf

Leal, M. 2008. “Tips and Tricks for 3D Interfaces on Mobile Devices”, *JavaOne Conference*, TS-6258, http://developers.sun.com/learning/javaoneonline/2008/pdf/TS-6258.pdf

Munshi, A., D.Ginsburg, D.Shreiner. 2008. *OpenGL ES 2.0 Programming Guide*, Addison-Wesley.

Petroshenko, P., A.Bhanushali, J.Evans, N.Ramani. 2007. “Whiz-Bang Graphics and Media Performance for Java Platform, Micro Edition”, *JavaOne Conference*, TS-5585, http://developers.sun.com/learning/javaoneonline/2007/pdf/TS-5585.pdf

Pulli, K., T.Aarnio, K.Roimela, and J.Vaarala. 2005. “Designing Graphics Programming Interfaces for Mobile Devices”, *IEEE Computer Graphics and Applications*, Vol. 25, No 8.

Pulli, K., T.Aarnio, V.Miettinen, K.Roimela, and J.Vaarala. 2007a. *Mobile 3D Graphics with OpenGL ES and M3G*, Morgan Kaufmann.

Pulli, K., J.Vaarala, V.Miettinen, R.Simpson, T.Aarnio, and M.Callow. 2007b. “The Mobile 3D Ecosystem”, One day course at *SIGGRAPH 2007*, San Diego, USA, August, http://people.csail.mit.edu/kapu/siggraph_2007/, Last accessed September 3rd 2008.

Wright, Jr., R.S. 2006. “OpenGL & Mobile Devices”, *Dr. Dobb's Journal*, May, http://www.ddj.com/mobile/187203532/, Last accessed September 3rd 2008.

BIBLIOGRAPHY

ANDREW DAVISON received his Ph.D. from Imperial College in London in 1989. He was a lecturer at the University of Melbourne for six years before moving to Prince of Songkla University in Thailand in 1996. He has also taught in Bangkok, Khon Kaen, and Hanoi.

His research interests include scripting languages, logic programming, visualization, and teaching methodologies. This latter topic led to an interest in teaching games programming in 1999.

His O'Reilly book, *Killer Game Programming in Java*, was published in 2005, and his Apress text, *Pro Java 6 3D Game Development*, in 2007.

SOPHIE RADENAHMUD recently completed his studies in Computer Engineering at Prince of Songkla University.