CHAPTER 1

INTRODUCTION

Mobile phones are the new vehicle for bringing interactive graphics technologies to consumers. Graphics that in the 1980s was only seen in industrial flight simulators and at the turn of the millennium in desktop PCs and game consoles is now in the hands of billions of people. This book is about the technology underpinnings of *mobile three-dimensional graphics*, the newest and most rapidly advancing area of computer graphics.

Computer graphics has been around since the 1960s. Its application areas range from user interfaces to video gaming, scientific visualization, special effects in movies, and even full-length animated films. In the field of computer graphics, it is the subset of three-dimensional (3D) graphics that produces the most life-like visuals, the "wow" effects, and the eye-candy. Since the late 1990s, almost all computer games, and more recently even operating systems such as OS X and Windows Vista, have come to rely heavily on real-time 3D graphics. This has created an enormous drive for graphics hardware development. Dedicated graphics hardware is ubiquitous on desktop and laptop computers, and is rapidly becoming common on high-end mobile phones. Low-cost software-based implementations bring 3D graphics to mass-market consumer phones as well. Computer graphics is nowadays an integral part of the phone user experience: graphics is the face of the device.

Mobile phones, also known as cellular or cell phones, have recently become universal communication and computation devices. In countries such as the UK there are more mobile phone subscriptions than there are people. At the same time, the capabilities of the devices are improving. According to Moore's law [Moo65], the transistor density on integrated circuits roughly doubles every one or two years; today's high-end mobile phone has more computational power than a late 1990s home PC. The display resolutions of mobiles will soon reach and surpass that of conventional broadcast television, with much better color fidelity. Together, these advances have resulted in a truly *mobile* computer. As a side effect, real-time, interactive 3D graphics has become feasible and increasingly desirable for the masses.

1.1 ABOUT THIS BOOK

This book is about writing real-time 3D graphics applications for mobile devices. We assume the reader has some background in mathematics, programming, and computer graphics, but not necessarily in mobile devices.

The 3D graphics capabilities of mobile devices are exposed through two standardized application programming interfaces (APIs): OpenGL ES, typically accessed through C or C++, and M3G, for mobile Java. We introduce the latter standard in terms of the former. As OpenGL ES is utilized as the fundamental building block in many real-world M3G implementations, expressing this relationship explicitly is highly useful for describing the inner workings of M3G.

The two APIs are equally suited to programming embedded devices other than mobile phones, from car navigation systems to display screens of microwave ovens. However, most of such platforms are *closed*—parties other than the device manufacturer cannot develop and install new applications on them. By contrast, most mobile phones are *open*: third parties such as professional software developers, students, and individual enthusiasts can program, install, and distribute their own applications. Having a programmable mobile phone at hand to try out the techniques described in this book is actually a great idea. However, the details of mobile application development vary considerably across platforms, so we defer those details to each platform's developer documentation.

This book consists of three parts and several appendices. Part I gives an introduction to the 3D graphics concepts that are needed to understand OpenGL ES and M3G, which are then covered in Parts II and III, respectively. The use of each API is demonstrated with hands-on code examples. The appendices provide additional information and optimization tips for both C/C++ and Java developers as well as a glossary of acronyms and terms used in this book. There is also a companion web site, `www.graphicsformasses.com`, hosting code examples, errata, and links to other online resources.

A more comprehensive treatment of 3D graphics, such as Real-Time Rendering by Tomas Akenine-Möller and Eric Haines [AMH02], is recommended for readers new to computer graphics. The "OpenGL Red Book" [SWN05] is a traditional OpenGL beginner's guide, while a book by McReynolds and Blythe [MB05] collects more advanced OpenGL tips in one place. Those unfamiliar with programming in mobile Java may find Beginning J2ME: From Novice to Professional by Sing Li and Jonathan Knudsen [LK05] useful.

1.1.1 TYPOGRAPHIC CONVENTIONS

Alongside the basic text, there are specific tips for achieving good performance and avoiding common pitfalls. These hints are called *performance tips* and *pitfalls*, respectively. An example of each follows:

> **Performance tip:** Enabling the optimization flag in the compiler makes your application run faster.
>
> **Pitfall:** Premature optimization is the root of all evil.

Code snippets and class, token, and function names are shown in `typewriter` typeface like this:

```
glPointSize( 32 );
glEnable( GL_POINT_SPRITE_OES );
glTexEnvi( GL_POINT_SPRITE_OES, GL_COORD_REPLACE_OES, GL_TRUE );
glDrawArrays( GL_POINTS, 0, 1 );
```

When API functions are introduced, they are marked like this:

`void function(int` *parameter*).

Any later references to the `function` or *parameter* in the text are also similarly emphasized.

1.2 GRAPHICS ON HANDHELD DEVICES

The very first mobile phones were heavy bricks with separate handsets; a few examples can be seen in Figure 1.1. They were designed to be lugged around rather than carried in

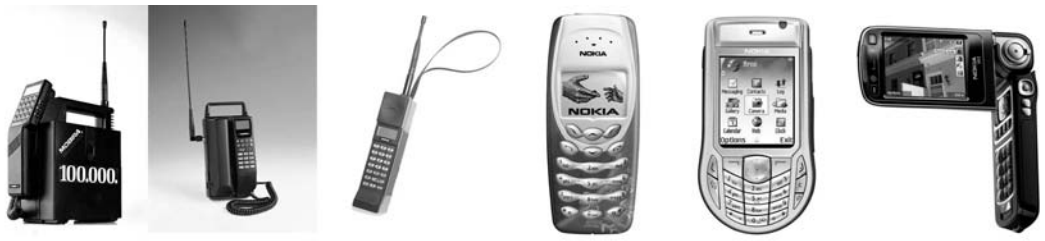


Figure 1.1: The evolution of mobile phones from the early car phones on the left to the multimedia computer on the right spans roughly two decades. From the left: Mobira Talkman, Nokia R72, Mobira Cityman, Nokia 3410 (the first GSM phone with a 3D graphics engine), Nokia 6630 (the first phone to support both OpenGL ES and M3G), and Nokia N93 (the first phone with hardware acceleration for both APIs). Images Copyright © 2007 Nokia Corporation.

a pocket, and they operated using analog radio networks. Toward the late 1980s and early 1990s, mobile phones started to become truly portable rather than just movable. By then the phones were pocket-sized, but still only used for talking.

Eventually, features such as address books, alarm clocks, and text messaging started to appear. The early alphanumeric displays evolved into dot matrices, and simple games, such as the Snake available in many Nokia phones, arrived. Calendars and e-mail applications quickly followed. Since the late 1990s, the mobile phone feature palette has exploded with FM radios, color displays, cameras, music players, web browsers, and GPS receivers. The displays continue to improve with more colors and higher resolutions, memory is installed by the gigabyte for storing increasing amounts of data, and ever more processing power is available to run a plethora of applications.

1.2.1 DEVICE CATEGORIES

Mobile phones today can be grouped roughly into three categories (see Figure 1.2): *basic phones*, the more advanced *feature phones*, and the high-end *smart phones*. There is significant variance within each category, but the classification helps imagine what kind of graphics applications can be expected in each. The evolution of mobile phones is rapid—today's smart phones are tomorrow's feature phones. Features we now expect only in the most expensive high-end devices will be found in the mass market in just a few years' time.

The basic phone category is currently not very interesting from the point of view of graphics programming: basic phones have closed environments, usually with proprietary operating systems, and new applications can be developed only in close association with the maker of the device. Basic phones are very limited in terms of their processing power and both the physical screen size and the display resolution. This class of phones does not have graphics hardware, and while software-based 3D solutions can be implemented, the limited CPU performance allows only the simplest of 3D applications.

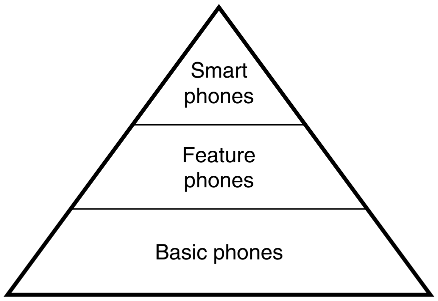


Figure 1.2: Three phone categories. Smart phones are more powerful than feature phones or basic phones, but there are more basic phones than either feature phones or smart phones.

The second category, on the other hand, *is* very interesting for graphics applications. Feature phones represent the bulk of the market in developed countries, and most of them incorporate mobile Java. Hundreds of different Java-enabled phone models are manufactured, and every year hundreds of millions of handsets are sold. Mobile Java makes it possible to develop applications for that entire volume of devices through a fairly uniform programming platform. It offers sufficient programming interfaces for most multimedia needs, 3D graphics included; the Mobile 3D Graphics API for Java ME (M3G) is one of the main topics in this book. The Java phones also span the largest range in terms of performance and feature differences—while the theory is "write once, run anywhere," in practice a lot of time is spent managing tens or even hundreds of different application configurations for different devices, prompting some to re-express the theory as "write once, debug everywhere."

The Qualcomm BREW platform[1] can be seen as a subset of mid-range devices that allow installation of native applications, written in C or C++. The security concerns of native applications are addressed through mandatory certification of developers and applications. BREW provides 3D graphics through OpenGL ES. Many BREW devices also support Java and M3G.

The top category in our classification is the high-end smart phone. The logical conclusion to the current smart phone evolution seems to be that these devices evolve into true mobile computers. Already today, the key features of the category include large, sharp, and vivid color displays, powerful processors, plenty of memory, and full-blown multimedia capabilities, not to mention the inherent network connectivity. Some of the latest devices also incorporate dedicated 3D graphics hardware. The operating systems (OSes), such as Symbian, Linux, and Windows Mobile, support the installation of third-party native applications. Java is also featured on practically all smart phones, and both OpenGL ES and M3G are typically available for 3D content.

1.2.2 DISPLAY TECHNOLOGY

The evolution of mobile phones coincides with the evolution of digital photography. Digital cameras started the demand for small, cost-efficient, low-power, high-quality displays. Mobile phones were able to leverage that demand, and soon small-display technology was being driven by mobile phones—and, eventually, by mobile phones *incorporating* digital cameras. Suddenly the world's largest mobile phone manufacturer is also the world's largest camera manufacturer.

Apart from the extreme low end, all mobile phones today have color displays. In the mid-range, resolutions are around one or two hundred pixels per side, with 16 or 18 bits of color depth, yielding 65K or 262K unique colors. High-end devices pack screens from QVGA (320 × 240 pixels) upward with good contrast, rapid refresh rates, and

1 `brew.qualcomm.com/brew/en/`

24 bits becoming the norm in color depth. Although there is room for improvement in brightness, color gamut, and field of view, among other things, it is safe to assume that display quality will not be the main obstacle for interactive 3D graphics on any recent feature phone or smart phone.

The main limitation of mobile displays is clearly their small physical size. A 50mm screen will never provide a truly immersive experience, even though the short viewing distance compensates for the size to some extent. For high-end console type of gaming, the most promising new development is perhaps the TV-out interface, already included in some high-end devices. A phone connected to a high-definition display has the potential to deliver the same entertainment experience as a dedicated games console. Near-eye displays, also known as data goggles, may one day allow as wide a viewing angle as the human eye can handle, while embedded video projectors and foldable displays may become viable alternatives to TV-out. Finally, autostereoscopic displays that provide different images to both eyes may yield a more immersive 3D experience than is possible using only a single image.

As with most aspects of mobile phones, there is a lot of variation in display properties. Application developers will have to live with a variety of display technologies, sizes, orientations, and resolutions—much more so than in the desktop environment.

1.2.3 PROCESSING POWER

Mobile phones run on battery power. While the processing power of integrated circuits may continue to increase in line with Moore's law [Moo65], roughly 40–60% per year, this is certainly not true of battery capacity. Battery technology progresses at a much more modest rate, with the energy capacity of batteries increasing perhaps 10% per year at best. In ten years' time, processing power may well increase twenty times *more* than battery capacity.

Needless to say, mobile devices need to conserve battery power as much as possible in order to provide sufficient operating times. Another reason to keep the power consumption low is heat dissipation: mobile devices are small, so there is very little surface area available for transferring the heat generated in the circuits out of the device, and very few users appreciate their devices heating noticeably. There is a potential ray of hope, though, in the form of *Gene's law*. It states that the power usage, and therefore heat dissipation, of integrated circuits drops in half every 18 months. This effect has made it possible to build ever smaller and faster circuits.

As shown in Figure 1.3, mobile phones typically have one or two processors. Each processor incorporates an embedded CPU, a digital signal processor (DSP), and perhaps some dedicated hardware for audio, imaging, graphics, and other tasks. The *baseband processor* takes care of the fundamental real-time operations of the device, such as processing the speech and radio signals. In basic phones and feature phones, the baseband

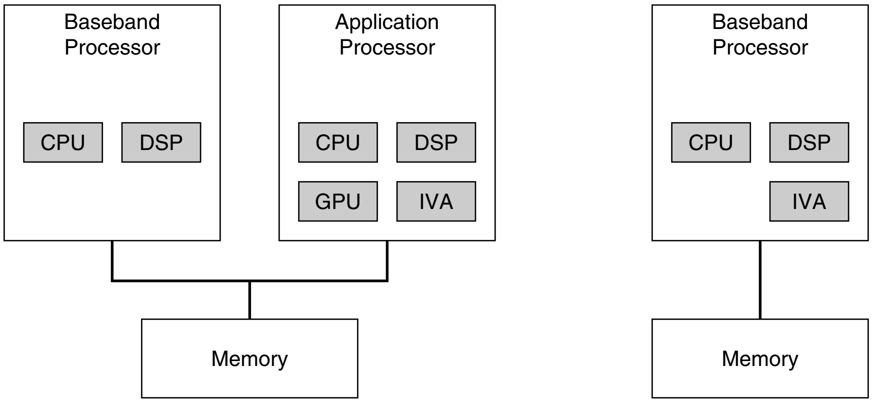


Figure 1.3: System architecture of a typical high-end smart phone (*left*) and a feature phone (*right*) in late 2007. Note that feature phones often include an Imaging and Video Accelerator (IVA), whereas a Graphics Processing Unit (GPU) is still relatively uncommon even in the smart phone segment.

processor also runs the operating system, applications, and the user interface—but of course at a lower priority. Smart phones usually have a separate *application processor* for these secondary purposes. To anyone coming from outside the mobile phone industry it may seem odd to call all this complex functionality "secondary." Indeed, the way forward is to make the application processor the core of the system with the modem becoming a peripheral.

The presence or absence of an application processor does not make much difference to the developer, though: exactly one CPU is available for programming in either case, and dedicated hardware accelerators may be present whether or not there is a separate application processor. The phone vendors also tend to be secretive about their hardware designs, so merely finding out what hardware is included in a particular device may be next to impossible. As a rule, the presence or absence of any hardware beyond the CPU that is running the application code can only be inferred through variation in performance. For example, a dual-chip device is likely to perform better for web browsing, multiplayer gaming, and other tasks that involve network access and heavy processing at the same time. In the rest of this book, we will not differentiate between baseband and application processors, but will simply refer to them collectively as "the processor" or "the CPU."

A mainstream mobile phone can be expected to have a 32-bit reduced instruction set (RISC) CPU, such as an ARM9. Some very high-end devices may also have a hardware floating-point unit (FPU). Clock speeds are reaching into half a gigahertz in the high end, whereas mid-range devices may still be clocked at barely 100MHz. There are also large variations in memory bus bandwidths, cache memories, and the presence or absence of hardware accelerators, creating a wide array of different performance profiles.

1.2.4 GRAPHICS HARDWARE

At the time of writing, the first generation of mobile phones with 3D graphics accelerators (GPUs) is available on the market. Currently, most of the devices incorporating graphics processors are high-end smart phones, but some feature phones with graphics hardware have also been released. It is reasonable to expect that graphics acceleration will become more common in that segment as well. One reason for this is that using a dedicated graphics processor is more power-efficient than doing the same effects on a general-purpose CPU: the CPU may require a clock speed up to 20 times higher than a dedicated chip to achieve the same rendering performance. For example, a typical hardware-accelerated mobile graphics unit can rasterize one or two bilinear texture fetches in one cycle, whereas a software implementation takes easily more than 20 cycles.

Figure 1.4 shows some of the first-generation mobile graphics hardware in its development stage. When designing mobile graphics hardware, the power consumption or *power efficiency* is the main driving factor. A well-designed chip does not use a lot of power internally, but power is also consumed when accessing external memory—such as the frame buffer—outside of the graphics core. For this reason, chip designs that cache graphics resources on the GPU, or store the frame buffer on the same chip and thus minimize traffic to and from external memory, are more interesting for mobile devices than for desktop graphics cards.

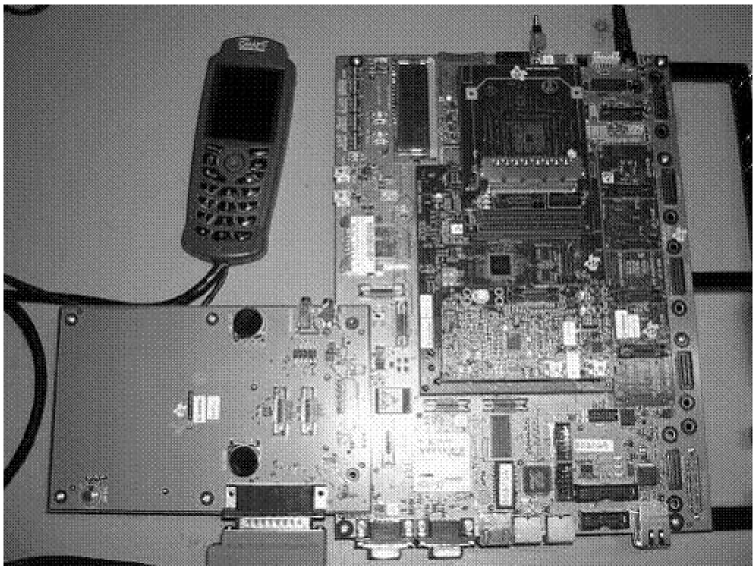


Figure 1.4: Early mobile graphics hardware prototype. Image copyright © Texas Instruments.

The graphics processor is only a small part of a multi-purpose consumer device which is sold as a complete package. Not all consumers take full advantage of the features made possible by the graphics hardware (e.g., high-end gaming, 3D navigation or fancy user interfaces), so they are not willing to pay a premium for it. In order to keep the cost of the device appealing to a variety of customers, the graphics core must be cheap to manufacture, i.e., it must have a small silicon area.

Graphics hardware for mobile devices cannot take the same approach as their desktop counterparts, sacrificing silicon area and power consumption for high performance. The design constraints are much tighter: the clock speeds and memory bandwidths are lower, and different levels of acceleration are required by different types of devices. For instance, many mobile GPUs only implement the rasterization stage of the rendering pipeline in hardware, leaving the transformation and lighting operations to be executed in software.

Rather than looking at raw performance, a much better metric is *performance per milliwatt*. High-end mobile GPUs in phones currently available in the market consume some hundreds of milliwatts of power at full speed, and can reach triangle throughputs of several million triangles per second, and pixel fill rates of hundreds of megapixels per second. Next-generation mobile GPUs are expected to have relative performance an order of magnitude higher.

1.2.5 EXECUTION ENVIRONMENTS

In the desktop arena, there are only three major families of operating systems: Windows, Linux, and Mac OS. Even though they have various differences in their design, and can seem very different from each other on the surface, the basic low-level idioms for writing programs are relatively similar. In the mobile space, there are dozens of different operating systems, and they each have their own quirks. As an example, some OSes do not support writable static data, i.e., static variables inside functions, global variables, or nonconstant global arrays. Other operating systems may lack a traditional file system. This means that things often taken for granted cannot be used in a portable fashion.

Open development environments

Traditionally all the embedded operating systems were closed, meaning that only the platform providers could write and install applications on them. The basic phones are appliances dedicated to a single purpose: making phone calls.

There are several reasons to keep platforms closed. If you allow third parties to install applications on your device after the purchase, the requirements for system stability are much higher. There are also significant costs related to software development, e.g., documentation, supporting libraries, and developer relations. Additionally, you have less freedom to change your implementations once other parties rely on your legacy features. Security is also a critical aspect. If applications cannot be installed, neither can malware, e.g., viruses that could erase or forward your private information such as the address book and calendar entries, or call on your behalf to a $9.95-per-minute phone number.

However, modern smart phones are not any longer dedicated appliances, they are true multimedia computers. Providing all applications is a big and expensive engineering task for a single manufacturer. When a platform is opened, a much larger number of engineers, both professionals and hobbyists, can develop key applications that can both create additional revenue and make the device on the whole a more attractive offering. Opening up the platform also opens possibilities for innovating completely new types of applications. On the other hand, there may be financial reasons for the exact opposite behavior: if one party can control which applications and functionalities are available, and is able to charge for these, it may be tempted to keep an otherwise open platform closed.

Nevertheless, the majority of mobile phones sold today have an open development environment. In this book, we employ the term *open platform* rather loosely to cover all devices where it is possible to program and install your own applications. Our definition also includes devices that require additional certifications from the phone manufacturer or the operator. Examples of open platforms include Java, BREW/WIPI, Linux, Palm OS, Symbian OS, and Windows Mobile.

A *native* application is one that has been compiled into the machine code of the target processor. We use the designation *open native platform* for devices that allow installing and executing native applications. For example, S60 devices are considered native whereas Java-only phones are not. Some 10–15% of all phones sold worldwide in 2006 fall into this category, roughly half of them being S60 and the other half BREW/WIPI phones.

Native applications

In basic phones and feature phones, the only way to integrate native binary applications is to place them into the firmware when the phone is manufactured. Smart phones, by contrast, allow installing and executing native binary applications. A key advantage for such applications is that there are few or no layers of abstraction between the running code and the hardware. They also can have access to all device capabilities and the functionality provided by system libraries. Therefore these applications can get all the performance out of the hardware.

This comes at the cost of portability. Each platform has its own quirks that the programmers have to become familiar with. There are several initiatives underway that aim to standardize a common native programming environment across the various operating systems, e.g., the OpenKODE standard[2] from the Khronos Group.

With regards to the 3D graphics capability, most mobile operating system vendors have selected OpenGL ES as their native 3D programming API. There still exist a few

2 `www.khronos.org/openkode`

proprietary solutions, such as Direct3D Mobile on Windows Mobile, and the Mascot Capsule API in the Japanese market. Regardless, it seems highly unlikely that any new native 3D rendering APIs would emerge in the future—the graphics API wars waged in the desktop arena in the mid-1990s are not to be re-fought in the embedded world. This furthers the portability of the core graphics part of an application. Even if OpenGL ES is not integrated with the operating system out of the box, software-based OpenGL ES implementations are available which can be either directly linked to applications or installed afterward as a system-level library.

Mobile Java

Nearly all mobile phones sold in developed countries today are equipped with Java Micro Edition,[3] making it by far the most widely deployed application platform in the world. Java ME has earned its position because of its intrinsic security, fairly open and vendor-neutral status, and its familiarity to millions of developers. It also provides better productivity for programmers compared to C/C++, especially considering the many different flavors of C/C++ that are used on mobile devices. Finally, the fact that Java can abstract over substantially different hardware and software configurations is crucial in the mobile device market where no single vendor or operating system has a dominating position. Most manufacturers are hedging their bets between their proprietary software platforms and a number of commercial and open-source options, but Java developers can be blissfully unaware of which operating system each particular device is using. Practically all mobile Java platforms provide the same 3D graphics solution: the M3G API, described in this book.

The Java platform is a perfect match for an otherwise closed system. It gives security, stability, and portability almost for free, thanks to its virtual machine design, while documentation and support costs are effectively spread among all companies that are participating in Java standardization, i.e., the Java Community Process, or JCP, and shipping Java-enabled products.

Even for a platform that does allow native applications, it makes a lot of sense to make Java available as a complementary option. Java gives access to a vast pool of applications, developers, tools, and code that would otherwise not be available for that platform. Also, developers can then choose between the ease of development afforded by Java, and the more powerful native platform features available through C/C++.

Of course, the secure and robust virtual machine architecture of Java has its price: reduced application performance and limited access to platform capabilities. Isolating applications from the underlying software and hardware blocks access to native system libraries and rules out any low-level optimizations. It is not just a myth that Java code is slower than C/C++, particularly not on mobile devices. The Java performance issues are covered more thoroughly in Appendix B.

3 `java.sun.com/javame`

1.3 MOBILE GRAPHICS STANDARDS

The mobile graphics revolution started small. The first phones with an embedded 3D engine were shipped by J-Phone, a Japanese carrier, in 2001. The graphics engine was an early version of the Mascot Capsule engine from HI Corporation. Its main purpose at the time was to display simple animated characters. Therefore many common 3D graphics features such as perspective projection, smooth shading, and blending were omitted altogether.

The first mobile phone to support 3D graphics outside of Japan was the Nokia 3410, first shipped in Europe in 2002 (see Figure 1.1). Unlike the Japanese phones, it still had a monochrome screen—with a mere 96 by 65 pixels of resolution—but it did incorporate all the essential 3D rendering features; internally, the graphics engine in the 3410 was very close to OpenGL ES 1.0, despite preceding it by a few years. A lightweight animation engine was also built on top of it, with an authoring tool chain for Autodesk 3ds Max. The phone shipped with animated 3D text strings, downloadable screensaver animations, and a built-in Java game that used simple 3D graphics. The application that allowed the users to input a text string, such as their own name or their sweetheart's name, and select one of the predefined animations to spin the 3D extruded letters around proved quite popular. On the other hand, downloading of artist-created screensaver animations was less popular.

Other early 3D graphics engines included Swerve from Superscape, ExEn (Execution Engine) from InFusio, X-Forge from Fathammer, and mophun from Synergenix. Their common denominator was that they were not merely hardware abstraction layers. Instead, they were middleware and game engine solutions incorporating high-level features such as animation and binary file formats, and in many cases also input handling and sound. All the solutions were based on software rendering, so there was no need to standardize hardware functionality, and features outside of the traditional OpenGL rendering model could easily be incorporated. However, in the absence of a unified platform, gaining enough market share to sustain a business proved difficult for most contenders.

1.3.1 FIGHTING THE FRAGMENTATION

A multitude of different approaches to the same technical problem slows down the development of a software application market. For example, a large variety of proprietary content formats and tools increases the cost of content creation and distribution. To make creating interesting content sensible for content developers, the market needs to be sufficiently robust and large. This is not so much an issue with pre-installed content, such as built-in games on handsets, but it is crucial for third-party developers.

There are strong market forces that encourage fragmentation. For example, the mobile phone manufacturers want their phones to differentiate from their competition. Operators want to distinguish themselves from one another by offering differing services. And the dozens of companies that create the components that form a mobile phone, i.e., the hardware and software vendors, all want to compete by providing distinct features. In other words, there is a constant drive for new features. When you want the engineering problems related to a new feature solved, you will not normally wait for a standard to develop. As a result, any new functionality will usually be introduced as a number of proprietary solutions: similar, but developed from different angles, and more or less incompatible with each other.

After the first wave, a natural next step in evolution is a *de facto* standard—the fittest solution will rise above the others and begin to dominate the marketplace. Alternatively, lacking a single leader, the industry players may choose to unite and develop a joint standard. The required committee work may take a while longer, but, with sufficient support from the major players, has the potential to become a win-win scenario for everyone involved.

For the third-party application developer, the size—or market potential—of a platform is important, but equally important is the ease of developing for the platform. Portability of code is a major part of that. It can be achieved at the binary level, with the same application executable running on all devices; or at the source code level, where the same code can be compiled, perhaps with small changes, to all devices. Standard APIs also bring other benefits, such as better documentation and easier transfer of programming skills. Finally, they act as a concrete target for hardware manufacturers as to which features should be supported in their hardware.

In 2002, the Khronos Group, a standardization consortium for specifying and promoting mobile multimedia APIs, created a steering committee for defining a subset of OpenGL suitable for embedded devices. The following companies were represented in the first meeting: 3d4W, 3Dlabs, ARM, ATI, Imagination Technologies, Motorola, Nokia, Seaweed, SGI, and Texas Instruments. Concurrently with this, a Nokia-led effort to standardize a high-level 3D graphics API for Java ME was launched under the auspices of the Java Community Process (JCP). It was assigned the Java Specification Request number 184 (hence the moniker "JSR 184") but the standard has become better known as M3G. The Expert Group of JSR 184 was a mixture of key mobile industry players including Nokia, Motorola, Vodafone, and ARM, as well as smaller companies specializing in 3D graphics and games such as Hybrid Graphics, HI Corporation, Superscape, and Sumea. The two standards progressed side-by-side, influencing each other as there were several people actively contributing to both. In the fall of 2003 they were both ratified within a few months of each other, and OpenGL ES 1.0 and M3G 1.0 were born. The first implementations in real handheld devices began shipping about a year later.

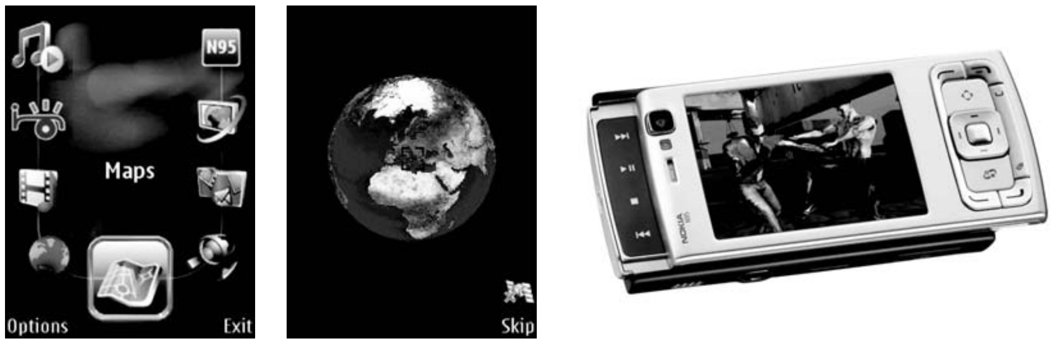


Figure 1.5: Uses of OpenGL ES in the Nokia N95 multimedia computer. On the left the multimedia menu and the mapping application of Nokia N95; on the right, a mobile game. Images Copyright © Nokia. (See the color plate.)

Today, you can get an overview about the market status by looking at the result databases of the different mobile graphics benchmarks: JBenchmark[4] (Figure 1.12), GLBenchmark[5] (Figure 1.6), and the various Futuremark benchmarks[6] (Figure 1.9). Devices supporting M3G are available from all major handset vendors, and OpenGL ES 1.1 hardware is being supplied to them by several companies, e.g., AMD, ARM, NVIDIA, and Imagination Technologies (PowerVR). Practical implementations vary from software renderers on ARM7 processors to high-end GPUs. The initial focus of mobile 3D graphics has also broadened from games and screen savers; it is now finding its way to user interfaces (see Figures 1.5, 1.7, and 1.8), and is available for the visualization needs of all applications.

The emergence of open standards shows that healthy competition should occur over implementation—quality, performance, cost, and power consumption—but not functionality that causes fragmentation.

1.3.2 DESIGN PRINCIPLES

The planning for the mobile 3D graphics standards was based on the background outlined earlier in this chapter: the capabilities of mobile devices, the available software platforms, and the need to create an interesting, unified market for both content developers and hardware vendors. It was clear from the start that a unified solution that caters for both Java and native applications was needed. A number of design principles, outlined in the following, were needed to guide the work. For a more in-depth exposition, see the article by Pulli et al. [PARV05].

4 www.jbenchmark.com

5 www.glbenchmark.com

6 www.futuremark.com

Performance is crucial on devices with limited computation resources. To allow all of the processing power to be extracted, the APIs were designed with performance in mind. In practice, this means minimizing the overhead that an application would have to pay for using a standard API instead of a proprietary solution.

Figure 1.6: Screen shot from the GLBenchmark benchmarking suite for OpenGL ES. Image copyright © Kishonti Informatics LP. (See the color plate.)

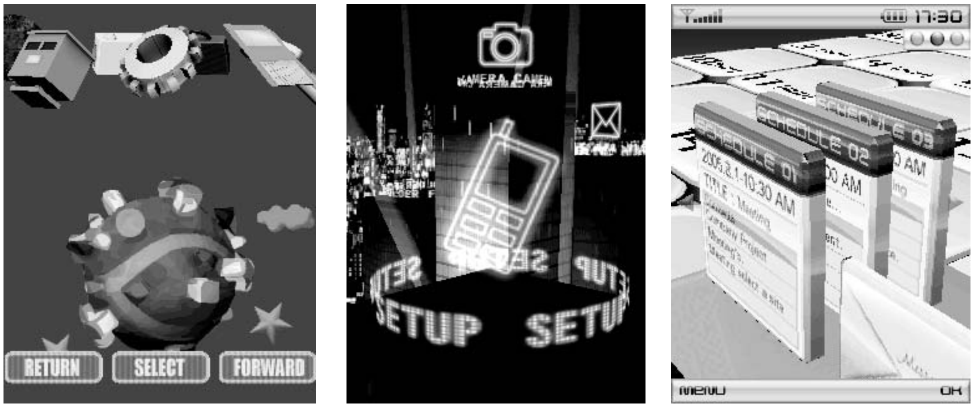
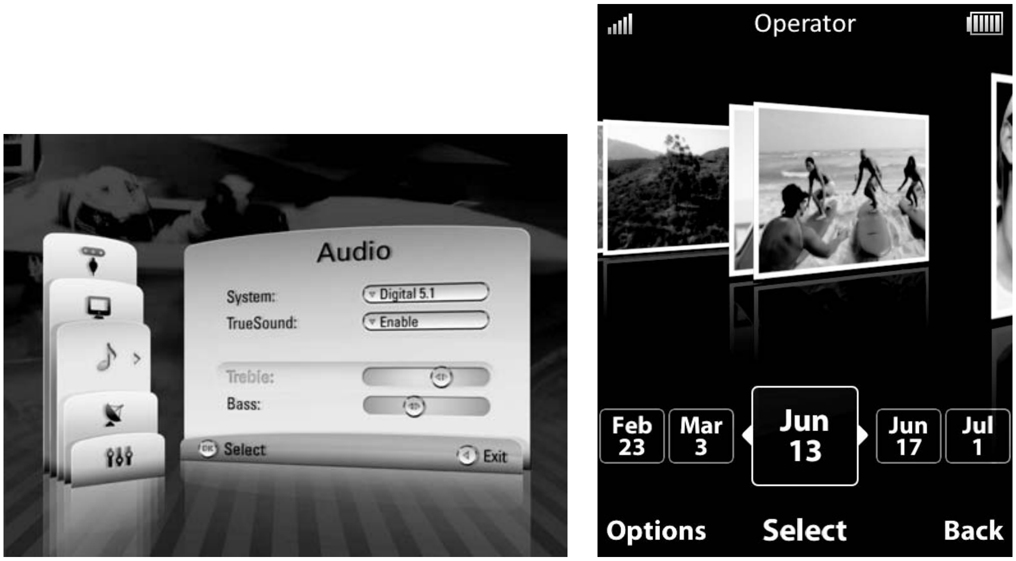


Figure 1.7: More 3D user interface examples. Images copyright © Acrodea. (See the color plate.)



System: Digital 5.1
TrueSound: Enable
Treble:
Bass:

Figure 1.8: 3D user interface examples. Images copyright © TAT. (See the color plate.)

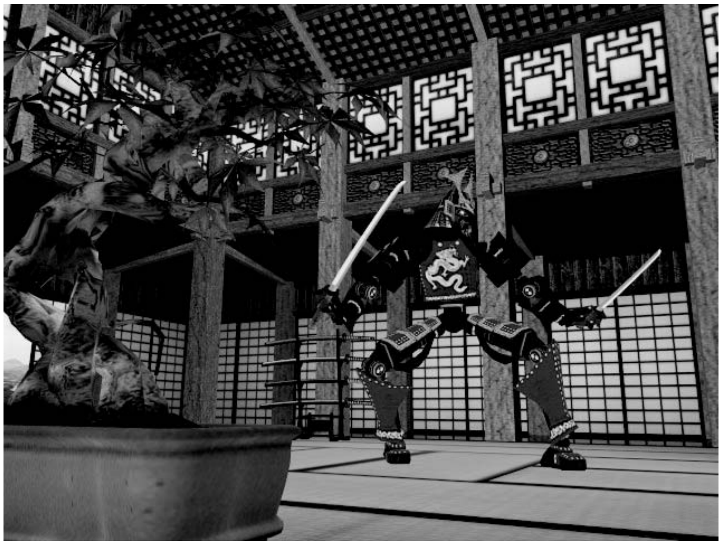


Figure 1.9: A VGA resolution screen shot from 3DMark Mobile 06, an OpenGL ES benchmark program. Image copyright © Futuremark. (See the color plate.)

Low complexity as a requirement stems from the stringent silicon area and ROM footprint budgets of mobile phones. To satisfy this goal, the engines underlying the OpenGL ES and M3G APIs were required to be implementable, in software, in under 50kB and 150kB, respectively. The key tools for reaching these targets were removal of redundant and seldom-used features.

A rich feature set should not be compromised even when aiming for compact APIs. As a guideline, features that would be very difficult to replicate in application code—the latter parts of the graphics pipeline, such as blending and texture mapping, fall into this category—should be adopted as fully as feasible, whereas front-end features such as spline evaluation or texture coordinate generation can be left for the applications to implement.

Small applications are much more important on mobile devices than on the desktop. Applications are often delivered over relatively slow over-the-air connections, with the users paying by the kilobyte, and stored in small on-device memories. This means that the 3D content has to be delivered efficiently, preferably in a compressed binary format. Support of compact geometry formats (such as using bytes or shorts for coordinates, instead of floats) helps in reducing the RAM consumption. Finally, it makes sense for the API to incorporate functionality that is common to many applications, thus saving the code space that would otherwise be required to duplicate those features in each application.

Hardware-friendly features and a clear path for hardware evolution were among the most important design goals. Adopting the familiar OpenGL rendering model as the base technology enabled the design of dedicated mobile graphics hardware for mass markets.

Productivity is especially important for mobile developers, as the development times of mobile games are typically short compared to desktop. M3G is designed especially to have a good match to existing content creation tools and to support concurrent development of application code and art assets.

Orthogonal feature set means that individual rendering features are not tied to each other. Feature orthogonality makes the behavior of the graphics engine easier to predict, as complex interdependencies and side-effects are minimized. This was already one of the key design criteria for desktop OpenGL.

Extensibility is important for any API that is to be around for several years. The mobile graphics industry is proceeding rapidly, and there has to be a clearly defined path for evolution as new features need to be incorporated.

Minimal fragmentation lets content developers work on familiar ground. Therefore, both OpenGL ES and M3G attempt to strictly mandate features, keeping the number of optional features as small as possible.

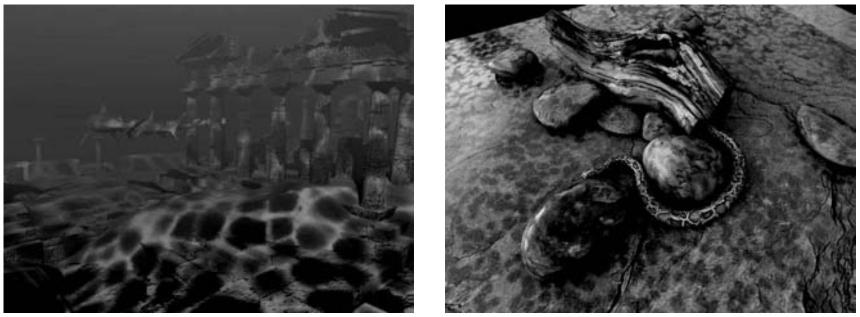


Figure 1.10: Demonstrating some of the advanced shading capabilities made possible by OpenGL ES 2.0. Images copyright © AMD. (See the color plate.)

1.3.3 OPENGL ES

OpenGL ES is a compact version of the well-known OpenGL graphics standard. It is a low-level rendering API adapted for embedded systems. The first version, OpenGL ES 1.0, aimed to provide an extremely compact API without sacrificing features: it had to be implementable fully in software in under 50kB of code while being well-suited for hardware acceleration. The graphics effects familiar from desktop had to be available on mobile devices as well.

Later, OpenGL ES 1.1 included more features amenable to hardware acceleration, in line with the feature set of first-generation mobile 3D graphics chipsets. The latest version, OpenGL ES 2.0, provides a completely revamped API, and support for a high-level shading language (GLSL ES): it replaces several stages of the traditional fixed-function graphics pipeline with programmable vertex and fragment shaders, and is therefore not backward-compatible with the 1.x series. The 1.x and 2.x generations of OpenGL ES continue to coexist, together providing 3D graphics capabilities to the entire range of embedded devices from wristwatches to smart phones, modern games consoles, and beyond. All OpenGL ES 2.x devices are expected to ship with ES 1.1 drivers. Details of the 2.x standard are beyond the scope of this book. GLSL ES is closely related to the OpenGL Shading Language, well described by Rost [Ros04].

A companion API called EGL, described in Chapter 11, handles the integration of OpenGL ES into the native windowing system of the operating system, as well as managing rendering targets and contexts. Finally, there is a separately specified safety-critical profile called OpenGL SC, but its markets are mostly outside of consumer devices—for example, in avionics instrumentation. OpenGL ES bindings are also available for other languages, such as Java and Python.

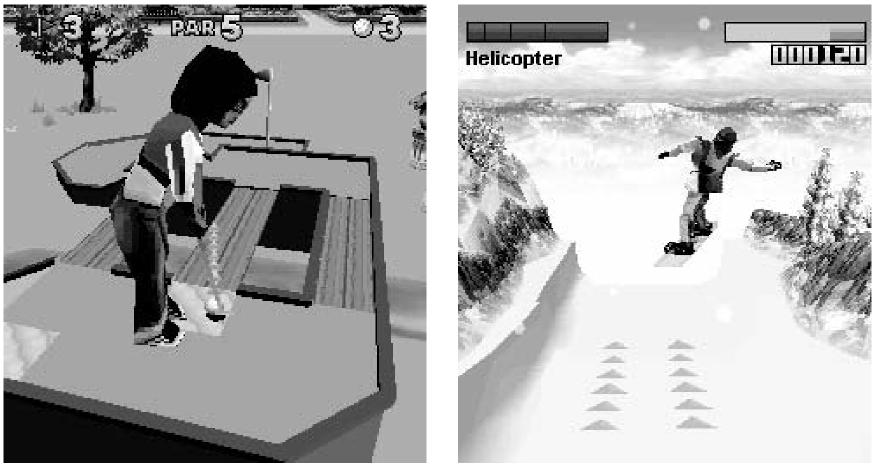


Figure 1.11: Java games using M3G. Images copyright © Digital Chocolate. (See the color plate.)

1.3.4 M3G

As the first Java-enabled phones hit the market in 2000 or so, it became evident that the performance and memory overhead of Java was prohibitive for real-time 3D. Software rasterizers written in pure Java would run orders of magnitude slower compared to those implemented in native code, while the power of any graphics hardware would be wasted on not being able to feed it with triangles fast enough.

Since the overhead of mobile Java was not going to magically vanish, there was a need for a new standard API that would shift as much processing as possible into native code. Since the data used by the native code cannot reside in the Java heap, a *retained mode* API was deemed more suitable than a direct mapping of OpenGL ES to mobile Java.

M3G is a completely new high-level API that borrows ideas from previous APIs such as Java 3D and OpenInventor. It consists of nodes that encapsulate 3D graphics elements. The nodes can be connected to form a scene graph representing the graphics objects and their relationships. M3G is designed so that it can be efficiently implemented on top of an OpenGL ES renderer.

Standardized high-level APIs have never been as popular on desktop as low-level ones. The main reason is that a high-level API is always a compromise. The threshold of writing a dedicated engine, such as a game engine, on top of a hardware-accelerated low-level API has been relatively low. However, if developers want to create such an engine using mobile Java, it has to be implemented completely in Java, incurring a significant performance penalty compared to native applications. A standardized high-level API, on the

Figure 1.12: Screen shot from the JBenchmark performance benchmarking suite for M3G. Image copyright © Kishonti Informatics LP. (See the color plate.)

other hand, can be provided by the device manufacturers, and it can be implemented and optimized in C/C++ or even assembly language. The native core then only has a thin Java layer to make the functionality available to Java applications.

Additional features of M3G include extensive support for animation and binary content files. Any property of any object can be keyframe-animated, and there are special types of meshes that support *skinning* (e.g., for character animation), and *morphing* (e.g., for facial animation). There is also an associated standardized binary file format that has one-to-one mapping with the API. This greatly facilitates separation of artistic content from programmable application logic.

Version 1.1 of M3G was released in mid-2005, with the aim of tightening up the specification for better interoperability. As M3G 1.1 does not add any substantial functionality over the original version, device vendors have been able to upgrade to it pretty quickly. M3G 1.1 is in fact required by the Mobile Service Architecture standard (JSR 248).

As of this writing, M3G 2.0 is being developed under JSR 297. The new version will make programmable shaders available on high-end devices, while also expanding the feature set and improving performance on the mass-market devices that do not have programmable graphics hardware, or any graphics hardware at all.

1.3.5 RELATED STANDARDS

There are several mobile graphics and multimedia standards closely related to OpenGL ES and M3G. This book concentrates only on graphics APIs, but for sound and multimedia in general, you can refer to standards such as JSR 135 for Java applications, or the native standards OpenSL ES, OpenMAX, and OpenKODE from the Khronos Group.

OpenGL ES for Java (JSR 239)

JSR 239[7] is a Java Specification Request that aims to expose OpenGL ES and EGL to mobile Java as directly as possible. Its promise is to provide the full OpenGL ES functionality for maximum flexibility and performance. The different OpenGL ES versions are presented as a hierarchy of Java interfaces. The base `GL` interface is extended with new functions and tokens in `GL10` and `GL11`, for OpenGL ES versions 1.0 and 1.1, respectively. Several OpenGL ES extensions are also exposed in the API, so features beyond the core functionality can be accessed.

Being a Java API, JSR 239 extends the error handling from native OpenGL ES with additional exceptions to catch out-of-bounds array accesses and other potential risks to system security and stability. For example, each draw call is required to check for indices referring outside the currently enabled vertex arrays.

There are no devices available as of this writing that would include JSR 239. Sony Ericsson have announced support for it in their latest Java Platform release (JP-8), and the first conforming phone, the Z750i, is likely to be shipping by the time this book goes to press. There is also a reference implementation available in the Java Wireless Toolkit from Sun Microsystems. Finally, in Japan, the DoCoMo Java (DoJa) platform version 5.0 includes proprietary OpenGL ES bindings.

2D vector graphics

The variety of screen resolutions on mobile devices creates a problem for 2D content. If graphics are rendered and distributed as bitmaps, chances are that the resolution of the content is different from the screen resolution of the output device. Resampling the images to different resolutions often degrades the quality—text especially becomes blurry and difficult to read. Bitmap graphics also requires significant amounts of memory to store and a high bandwidth to transmit over a network, and this problem only gets worse as the display resolutions increase. Scalable 2D vector graphics can address both of these

7 `www.jcp.org/en/jsr/detail?id=239`

problems. If the content is represented as shapes such as curves and polygons instead of pixels, it can often be encoded more compactly. This way content can also be rendered to different display resolutions without any loss of quality, and can be displayed as the content author originally intended.

2D vector graphics has somewhat different requirements from 3D graphics. It is used for high-quality presentation graphics, and features such as smooth curves, precise rules for line caps, line joins, and line dashes are much more important than they are for 3D content. Indeed, these features are often only defined in 2D, and they may not have any meaning in 3D. It is also much easier to implement high-quality anti-aliasing for 2D shapes.

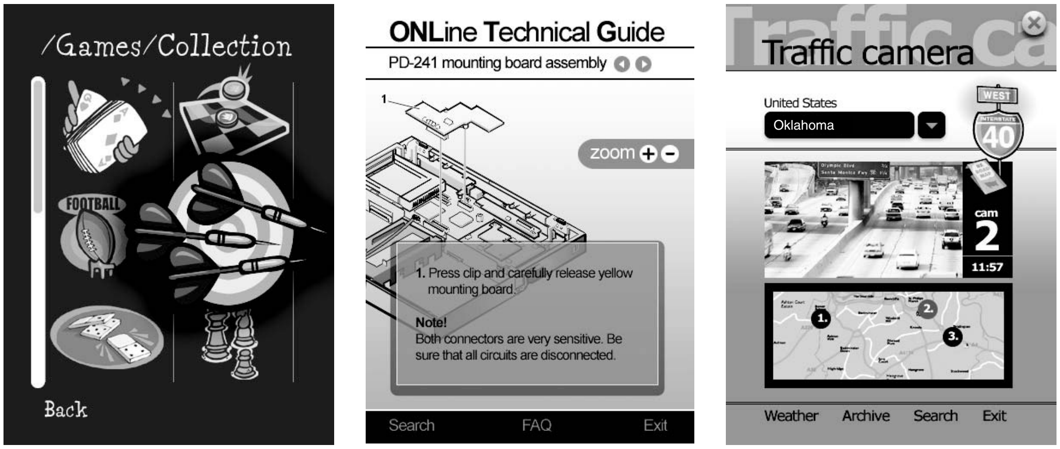
Scalable Vector Graphics (SVG) is a standard defined by the World Wide Web Consortium (W3C).[8] It is an XML-based format for describing 2D vector graphics content. SVG also includes a declarative animation model that can be used, for example, for cartoons and transition effects. In addition, the content can be represented as a Document Object Model (DOM), which facilitates dynamic manipulation of the content through native application code or scripting languages such as JavaScript. The DOM API also allows applications to register a set of event handlers such as `mouseover` and `click` that can be assigned to any SVG graphical object. As a result, SVG can be used to build dynamic web sites that behave somewhat like desktop applications.

W3C has also defined mobile subsets of the standard, SVG Tiny and SVG Basic.[9] The latter is targeted for Personal Digital Assistants (PDAs), while the smaller SVG Tiny is aimed for mobile phones. However, it seems that SVG Basic has not been widely adopted by the industry, while SVG Tiny is becoming commonplace and is being further developed.

The Khronos Group has defined the OpenVG API for efficient rendering of 2D vector graphics. OpenVG has similar low-level structure as OpenGL ES, and its main use cases include 2D user interfaces and implementations of 2D vector graphics engines such as SVG Tiny and Adobe's Flash. Whereas most 2D vector graphics engines traditionally execute on the CPU, OpenVG has been designed for off-loading the rasterization to dedicated graphics hardware (see Figure 1.13). This was necessary in the mobile space because most devices have limited CPU resources. The OpenVG rendering primitives were chosen so that all rendering features of SVG Tiny can be easily implemented using the API. The basic drawing primitive is a path which can contain both straight line segments as well as smoothly curving Bézier line segments. The paths can describe arbitrary polygons, which can be filled with solid colors, color gradients, bitmap images, or even patterns made of other 2D objects. Recent versions of EGL allow rendering with both OpenGL ES and OpenVG to the same image, and even allow sharing data such as texture maps across the different Khronos APIs.

8 `www.w3.org/Graphics/SVG/`

9 `www.w3.org/TR/SVGMobile/`



1. Press clip and carefully release yellow mounting board.

Note!
Both connectors are very sensitive. Be sure that all circuits are disconnected.

Figure 1.13: The use of vector graphics makes it possible to create scalable, antialiased user interfaces. Hardware-accelerated OpenVG demonstrations. Images copyright © AMD.

Various 2D graphics interfaces exist for Java ME. Mobile Information Device Profile (MIDP), the most common Java profile on mobile phones, offers basic 2D graphics functionality with primitives such as lines, circles, and polygons, as well as bitmap graphics. It is quite well suited for the needs of simple 2D games and applications.

JSR 226, the scalable 2D vector graphics API for Java,[10] was created for more challenging 2D vector graphics applications. It is compatible with SVG Tiny 1.1, and can render individual images and graphics elements under the control of a Java application, or simply used as an "SVG Tiny player." It also supports the XML/SVG Micro DOM (μDOM) for manipulating properties of the SVG content via accessor methods and event handlers. JSR 226 was completed in 2005, and can be found in several phone models from manufacturers such as Nokia and Sony Ericsson.

JSR 287[11] is a backward-compatible successor to JSR 226. The enhancements of this API include the new graphics and multimedia features from SVG Tiny 1.2, e.g., opacity, gradients, text wrapping, audio, and video. The new version also allows creating animations on the fly. The Micro DOM support is extended from the previous version. The API also includes the necessary framework for processing streamed SVG scenes, and there is an immediate-mode rendering API that is compatible with OpenVG and designed for high performance. The standard is expected to be completed by the end of 2007. Based on historical evidence, the first devices can then be expected in late 2008.

10 `www.jcp.org/en/jsr/detail?id=226`

11 `www.jcp.org/en/jsr/detail?id=287`

COLLADA

COLLADA, short for COLLAborative Design Activity,[12] started as an open-source project led by Sony, but is nowadays being developed and promoted by the Khronos Group. COLLADA is an interchange format for 3D content; it is the glue which binds together digital content creation (DCC) tools and various intermediate processing tools to form a production pipeline. In other words, COLLADA is a tool for content development, not for content delivery—the final applications are better served with more compact formats designed for their particular tasks.

COLLADA can represent pretty much everything in a 3D scene that the content authoring tools can, including geometry, material and shading properties, physics, and animation, just to name a few. It also has a mobile profile that corresponds to OpenGL ES 1.x and M3G 1.x, enabling an easy mapping to the M3G binary file format. One of the latest additions is COLLADA FX, which allows interchange of complex, multi-pass shader effects. COLLADA FX allows encapsulation of multiple descriptions of an effect, such as different levels of detail, or different shading for daytime and nighttime versions.

Exporters for COLLADA are currently available for all major 3D content creation tools, such as Lightwave, Blender, Maya, Softimage, and 3ds Max. A stand-alone viewer is also available from Feeling Software. Adobe uses COLLADA as an import format for editing 3D textures, and it has been adopted as a data format for Google Earth and Unreal Engine. For an in-depth coverage of COLLADA, see the book by Arnaud and Barnes [AB06].

12 `www.khronos.org/collada`