

Special Interest Paper

November 2004

Mobile 3D Graphics and Java Applications Development for Sony Ericsson Phones



K700



S700



K500



Z500



V800



F500

Contents

Contents.....	2
Preface.....	4
1. What is Java Mobile 3D?	5
1.1. Java Mobile 3D is NOT Java 3D.....	5
1.1.1. Characteristics of Java 3D	5
1.1.2. Characteristics of Java Mobile 3D.....	5
1.2. Mascot Capsule Micro3D as a Bridge.....	6
1.2.1. Characteristics of Mascot Capsule Micro3D version 3.....	6
1.3. What's in a Name?	6
2. Sony Ericsson's Vision for Mobile 3D Applications	7
3. Adding a Dimension	9
3.1. Increasing Processing Power.....	9
3.2. More Capable Hardware	9
3.3. More Capable Software.....	9
3.4. Addition of Wireless Connectivity	10
3.5. The Pros and Cons of using Java for Mobile 3D Applications	10
3.6. User Expectations.....	11
3.7. Overview of Mascot Capsule Micro3D version 3	11
3.7.1. Micro3D version 3 classes	12
3.7.2. Graphics pipeline.....	12
3.7.3. Object Import	13
3.8. Overview of JSR 184.....	13
3.8.1. JSR 184 classes.....	14
3.8.2. JSR 184 rendering modes and graphics pipeline	16
3.8.3. Object Import	16
3.9. Impact on game engines and the platform.....	17
3.10. Where it's all going	18
4. Building Java Mobile 3D MIDlets.....	19
4.1. Java Development Tools	19
4.2. Sony Ericsson J2ME SDK.....	20
4.3. 3D Capable Mobile Phones.....	20
4.3.1. Basic hardware and software specifications	20
4.4. Running an Example 3D MIDlet	21
4.4.1. Make the project.....	22
4.4.2. Build and run the MIDlet	22
4.4.3. Debugging a MIDlet	23

4.5. File Translation/Export.....	25
4.5.1. Micro3D version 3 file translation chain	25
4.5.2. JSR 184 file translation chain	26
4.6. Tips and Tricks.....	27
4.6.1. 3D object descriptions for immediate mode	27
4.6.2. Texture map differences	28
4.6.3. Keep immediate mode and retained mode operations straight	28
4.6.4. Don't mix LCDUI and 3D graphics	28
4.6.5. Design within limits	28
5. Summary.....	30
6. Resources	31
6.1. Abbreviations	31
6.2. Further Information and Links	31

Preface

The purpose of this paper is to introduce the reader to Java-based mobile 3D technology as it appears Sony Ericsson's K500, K700, S700, and Z500 series, and the Vodafone F500i and V800 mobile phones. Specifically, the paper covers the following topics:

- Description of the capabilities of a Java mobile 3D graphics implementation
- An overview of the JSR 184 API (also known as Mobile 3D Graphics, or M3G), and HI Corporation's Mascot Capsule Micro3D API
- Discussion of how 3D content can be created on PCs for inclusion into Mobile 3D Java applications
- An overview of compiling a 3D Java application

1. What is Java Mobile 3D?

Java Mobile 3D Graphics (also known as JSR 184) is an API tailored for the generation and presentation of 3D content on mobile platforms. This section discusses this API and an alternate 3D implementation, HI Corporation's Mascot Capsule Micro3D version 3 (also known as Micro3D). Sony Ericsson has chosen to implement both API sets, allowing Mascot Capsule Micro3D serve as a migration path for 3D applications until Java Mobile 3D-enabled devices become widely available. Here, we'll also briefly try to address any possible point of confusion related to the two implementations of 3D Java APIs in Sony Ericsson phones.

1.1. Java Mobile 3D is NOT Java 3D

Before we delve into what Java Mobile 3D is, it is equally important that we understand what Java Mobile 3D is *not*. In particular, Java Mobile 3D should not be mistaken for Java 3D. Since the two Java extensions share nearly the same name and serve the same purpose, it's worth comparing and contrasting the two to avoid confusion.

Java 3D is a Java extension API under development by Sun Microsystems to implement 3D graphics rendering for the Java run-time environment on desktops and workstations. To be explicit, these are the APIs and classes that belong to the package `com.sun.j3d`.

Java Mobile 3D is being developed under the guidance of the Java Community Process (JCP), a consortium of companies whose purpose is to develop extensions for the Java platform. Members of the JCP (of which Sony Ericsson is one member) have devised a Java extension API that supports 3D graphics rendering on mobile devices. The JCP has documented this API in the Java Specification Request (JSR) 184, which is often termed "JSR 184". The classes that implement the JSR 184 specification belong to the package `javax.microedition.m3g`.

To reduce confusion, let's examine the characteristics of each 3D API more closely.

1.1.1. Characteristics of Java 3D

Java 3D provides a set of object-oriented interfaces that allow you to assemble 3D models from geometric primitives, render them, and manipulate them. Java 3D is designed to extend the capabilities of the Java 2 Standard Edition (J2SE) platform for desktop PCs and workstations. Java 3D supports a number of advanced technologies, such as stereo displays for virtual reality (VR) presentations and exotic input devices such as data gloves and VR headsets.

Because PCs and workstations have ample hardware resources such as memory, processing power, and storage, Java 3D's classes are tailored for high-speed rendering rather than conserving memory or limiting the use of the processor. The classes are also optimized for using graphics acceleration hardware and a floating-point unit (FPU) when performing computations. Java 3D is clearly designed for its target platform, the capabilities of which are far different from those of mobile devices.

1.1.2. Characteristics of Java Mobile 3D

Like its desktop cousin, Java Mobile 3D (a.k.a. JSR 184) specifies object-oriented interfaces that enable developers to assemble 3D models, place these objects in a world, and render a view of this world. It is capable of displaying 3D scenes and animations in real-time. JSR 184 was designed to support 3D graphics on portable devices constrained in terms of amount of memory, memory bandwidth, and processing power. In addition, such devices often lack a FPU and graphics acceleration hardware. Specifically, JSR 184 extends the capabilities of the Java 2 Micro Edition (J2ME), a version of the Java platform tailored for portable devices.

The JSR 184 rendering engine can be implemented entirely in software, yet its architecture allows the engine to scale up and take advantage of any acceleration hardware or FPU, if present. The implementation has a small memory footprint, requiring only some hundred kilobytes of ROM, and the API methods are structured so that their use doesn't spawn duplicate objects. This conserves RAM and reduces garbage collection.

While an enterprising developer can write code that builds 3D models directly, JSR 184 can also import 3D model data from an external file. Along with geometric data, the file can contain visual attributes and texture maps for the

models, and even animation information. The 3D content is created by 3D authoring programs on a PC, and a special plug-in exports it into the proper file format. This allows 3D content to be created on a more powerful platform, then imported and displayed on the mobile device. The format of this 3D content file is an open standard and is described in great detail in the JSR 184 specification.

1.2. Mascot Capsule Micro3D as a Bridge

The JSR 184 specification's final revisions were approved in November of 2003. There's always an inevitable delay as implementations of the specification are brought into alignment with the final release. For example, while the Mobile Information Device Profile (MIDP) 2.0 specification was ratified in late 2002, MIDP 2.0-enabled phones are just now starting to ship in volume eighteen months later. JSR 184 will be no exception in this area: many vendors have spent most of this year putting the finishing touches on their JSR 184 implementations. Just now JSR 184-enabled phones are starting to appear in quantity: Sony Ericsson expects to have millions of K500, K700, S700, Z500, mobile phones on the market in 2005, and also that the Vodafone-exclusive F500i and V800 phones will sell well.

The dilemma a developer currently faces is how to write and deploy a 3D-capable Java application for product visibility during this interval as such devices begin to appear in numbers. Aggravating the problem is a current lack of 3D content in the JSR 184 file format. Sony Ericsson has teamed up with HI Corporation, a well-known maker of 3D rendering engines for embedded devices, to ease developers through this transition period. Therefore, the K500, K700, S700, and Z500 series, as well as the F500i and V800 mobile phones come equipped with *two* 3D APIs. These are: JSR 184 as implemented by HI Corporation, (through its Mascot Capsule Micro3D Engine version 4) for Sony Ericsson phones, as well as that company's own embedded 3D implementation, known as Mascot Capsule Micro3D Engine version 3. We'll discuss the differences between these two versions in detail in the next section.

1.2.1. Characteristics of Mascot Capsule Micro3D version 3

The Mascot Capsule Micro3D Engine version 3 implementation consists of a proprietary API and a rendering engine. It has been widely adopted by a number of handset manufacturers, network operators, and embedded vendors, making it something of a de facto standard, especially in high-tech consumer electronics countries like Japan and Korea. Micro3D is designed to work on embedded devices and has a small resource footprint, requiring only 100KB of RAM and about 250 KB of RAM.

Like Java Mobile 3D, Micro3D can import 3D geometry, appearance, and animation data from a file. HI Corporation has written plug-ins that allow you to save 3D model and action information from high-end 3D authoring programs such as Newtek's Lightwave, Discreet's Studio 3D Max, Avid System's SoftImage, and Alias System's Maya and export it for use in a Micro3D Java application. Note that the data files exporting the 3D information for import into the Micro3D environment are in a proprietary format.

Writing a Java MIDlet using the Mascot Capsule Micro3D Engine version 3 offers important benefits. First, you can develop and deploy a 3D Java MIDlet immediately for product visibility. Second, given Micro3D's reach in the embedded market, it's possible for you to migrate some unique 3D content and applications from other platforms to 3D-capable Sony Ericsson phones. Third, to migrate your application to JSR 184, you can leverage the expertise gained during the development of the Micro3D application to expedite the process. Since both Mascot Capsule Micro3D and JSR 184 use a similar a content import/export scheme, this reduces the amount of effort required to migrate application code.

1.3. What's in a Name?

The near-identical nomenclature for the J2SE and J2ME 3D implementations—Java 3D and Java Mobile 3D, respectively—is a source of confusion for anyone studying these APIs. Nor does the lengthy name of “Mascot Capsule Micro3D Engine version 3” for the proprietary 3D implementation lend itself to easy reading. To eliminate any possible ambiguity among the various 3D implementations, this paper adopts the following terms:

- **Java 3D** – Designates the Java 3D implementation for the J2SE platform.
- **JSR 184** – Designates the Java Mobile 3D implementation for the J2ME platform as implemented by Mascot Capsule Micro3D version 4 in Sony Ericsson phones.
- **Micro3D version 3** – Designates HI Corporation's proprietary Mascot Capsule Micro3D Engine 3D version 3 implementation for the Java platform in Sony Ericsson phones.

2. Sony Ericsson's Vision for Mobile 3D Applications

The mobile phone has evolved considerably over the years. Starting in 1973 as a 30-ounce (850-gram) “brick” that placed only phone calls over an analog network, today’s mobile phone has shrunk to a 3- or 4-ounce (85- or 113-gram) mite with a color screen that sends voice or text messages through an all-digital network. Interestingly, as the mobile phone’s physical dimensions shrank, its capabilities expanded. Besides placing calls, a mobile phone operates as a pager, an instant messaging and e-mail system, and a Web browser. Furthermore, you can download new programs to it Over The Air (OTA). Finally, the addition of audio hardware that supports multiple audio formats, a color screen that displays up to hundreds of thousands of colors, and enhancements to J2ME all make Sony Ericsson mobile phones a capable gaming platform.

Sony Ericsson has played a key role in shaping the design of mobile phones so that they have become such as versatile devices. The company was an early adopter of software standards such as J2ME MIDP 2.0, Mobile Media API (MMAPI, often known by its JCP specification, JSR 135), Wireless Messaging API (WMA, or JSR 120), and other standards-based extensions to the Java platform. Sony Ericsson’s support of software standards accomplishes the following:

- Provides features that network operators can leverage to offer new services or to enhance existing ones.
- Provides standards-based interfaces that developers can use to write best-of-breed applications that make use of these new features.
- Provides a well-documented infrastructure of new technologies that future applications and services can build upon.

It is at that latter point where Sony Ericsson looks forward by offering standards-based technologies that will enable the next generation of mobile applications. We believe that JSR 184 is a seminal technology that opens up new opportunities for developers, for the following reasons:

- **It will be a key driver for mobile phones to deliver dynamic content and entertainment to the user.** JSR 184 makes possible sophisticated 3D animations that can enhance the presentation of material. For example, with a little coding effort, JSR 184 permits novel ways to display and interact with the 3D content (perhaps a walkthrough of a very simplified building plan, or a game).
- **3D scenes scale more readily than 2D images, for better application portability.** Any developer who has ported a J2ME application across several mobile phone families with different screen sizes knows that the UI elements and 2D image graphics have to be tweaked so that the application displays properly on each phone. With JSR 184, a scene’s 3D models are generated via algorithms rather than by painting a fixed array of pixels onto the screen. In rendering an image, the model geometry is scaled computationally to accommodate a larger or smaller screen without distorting the appearance of the 3D objects in the scene. Stated another way, a single set of geometric models can be displayed on a range of screen sizes and still look good, and without additional code or resources.
- **Horizontal gaming styles will appear.** Most mobile phones use a vertically-oriented screen, while most portable game consoles prefer a horizontally-oriented screen. (You may hear the term “portrait” to describe the vertical orientation, and “landscape” to describe the horizontal orientation.) While most mobile phone applications use the vertical orientation to better display lots of text, game developers will continue to design a 3D game’s UI and scenery along a horizontal orientation. Changes in both software and hardware will both support and accelerate this trend. In terms of software, the horizontal orientation is possible because JSR 184 allows 3D graphics to be presented in the horizontal layout with a minimum of coding effort. In terms of hardware, the mobile phone’s controls must be arranged to facilitate its use in any orientation. The best example of this is our own 3D-capable phone, the Sony Ericsson S700 series. It presents a 240 W by 320 H pixel screen that, if oriented horizontally, presents a large display area that rivals that of some portable game consoles. In addition, the arrangement of its controls (such as the navigation keys) readily lends themselves to using the phone in either orientation.
- **3D capabilities will become ubiquitous.** Currently, JSR 184 is only an optional extension to the J2ME platform. That is, vendors can either choose to implement it in their devices or not. However, for 3D applications to become widespread, so must the support for JSR 184. To this end, Sony Ericsson plans to

eventually implement JSR 184 in all of its mobile phones. Currently, Sony Ericsson has embarked on this plan by providing JSR 184 in a number of high-end mobile phones. However, as more developers—and more importantly, users—embrace the technology, our company intends to push JSR 184 into lower segments of the mobile phone market. Over time, a developer can safely expect JSR 184 to be available on every Sony Ericsson mobile phone, from the high end to the low end. This results in a large potential audience that the developer can to sell applications to.

The K500, K700, S700, and Z500 series, and well as the F500I and V800 mobile phones, demonstrate Sony Ericsson's commitment to new technologies and standards that offer new avenues of opportunity for developers and network operators. Each of these 3D-capable phones has a large color screen that can display 16-bit pixels (65,536 colors) or 18-bit pixels (262,144 colors). They have large amounts of memory (the Java heap size is now 1.5 MB), and plenty of video RAM. Importantly, because 3D applications tend to be larger than other Java applications, these phones place no limit of the application's JAR file size. These mobile phones and their successors provide a solid mobile platform on which developers can devise innovative applications that display and manage 3D content.

3. Adding a Dimension

While it might appear that 3D-capable mobile devices have suddenly burst onto the market, the reality is that several key technologies had to both evolve and converge to a point where such a platform was possible. These technologies are: increasing processing power, more capable hardware, more capable software, and the addition of wireless connectivity. We shall explore how each of these advancements in technology contributes to the platform.

3.1. Increasing Processing Power

The display of 3D graphics requires a platform with considerable memory and computing prowess. That might not seem obvious when you're studying a mobile phone that fits comfortably in your hand. However, if you examine the specifications for Sony Ericsson's K500, K700, S700, and Z500 series, and the F500i and V800 mobile phones on the Sony Ericsson Developer World Web site, you'll see that each has a considerable amount of memory, particularly for what is essentially an embedded system. The amount of memory for these phones range from 6 MB of RAM in the Z500 series, to an impressive 41 MB for the K700 series. They use 32-bit processors clocked at frequencies of 100 MHz or faster.

These specifications closely parallel those of desktop computers sold in 1994, where a common memory size for a system was 8 to 16 MB, and the clock frequency of their 32-bit processors ranged from 60 to 100 MHz. While these numbers might not seem impressive when compared to today's desktop computers running at 3 GHz, that's still a lot of processing power that you're holding in your hand—and the built-in computer operates on just a battery. Over the past decade, the relentless drive in shrinking the dimensions of critical hardware components and reducing their power consumption has given us mobile phones that contain powerful computers. It is this formidable—yet portable—computing power that makes the presentation of 3D graphics possible.

3.2. More Capable Hardware

Besides a faster processor and more memory, other hardware improvements contributed to the genesis of the mobile 3D platform. For example, most Sony Ericsson phones now offer higher resolution screens (128 W by 160 H pixels or larger) that can display 16-bit or 18-bit colors. They also have high frame rates and low response times, which makes flicker-free animations possible.

Another improvement was the diminishing physical size of memory cards, especially those with non-volatile memory. Their dimensions have shrunk to the point where mobile phone manufacturer can offer a memory slot in mobile phones. This capability allows new programs to be loaded onto the mobile phone in minutes, and thus enables the mass distribution of software and content.

Last but not least, Improvements in battery technology, along with better power management functions in processors and chipsets, permit a mobile phone to display and manage large amounts of 3D content without exhausting the battery.

3.3. More Capable Software

Powerful hardware by itself does not make a 3D platform. Prior to 2001, serious technical issues confronted any developer aspiring to write applications for mobile phones. These issues were: non-existent software compatibility, security concerns, and the lack of an expansion mechanism.

To solve these fundamental problems, in 2001 many phone manufacturers adopted Sun Microsystem's Java platform as the execution environment of choice. The Java platform was chosen because it addressed these issues squarely.

In terms of software compatibility, Java presents an execution platform that's hardware independent. The platform consists of a set of APIs and a virtual machine (VM). Java programs are made up of bytecodes that execute on the VM and invoke the API for services. The Java platform never permits direct access to the hardware. A Java

application can therefore execute on any device that has a Java VM and API libraries on it. Practically, there are some compatibility issues, but overall the Java implementation comes closer to achieving broad application portability than other runtime schemes.

In addition to offering a device-independent execution platform, Java also has security built into it from the ground up. A program must pass a battery of code validity tests prior to being loaded, and the VM performs numerous run-time checks that prevent the execution environment from crashing. Java also has a trusted code mechanism that only allows signed code to access critical APIs or resources on the phone.

Finally, Java enables easy software expansion. From the beginning, it was designed so that additional programs and software components could be downloaded and integrated into the execution environment. On a desktop computer, the download occurs through a network link, while for a mobile phone the transfer takes place over its wireless connection. A recent OTA specification codifies how a mobile phone can search for, select, and download an application from a remote network. This specification also describes how these applications can be deleted from the phone later. For large applications, Java programs can also be loaded from a card inserted into the mobile phone's memory slot.

Java's unified software platform thus encourages third-party developers to write applications that can be distributed across a wide range of phones made by a phone manufacturer and provide value by promoting network use.

3.4. Addition of Wireless Connectivity

J2ME offered support for wireless connections from its inception. Add this capability on top of the Java's ability to download and incorporate new classes, and the basic mechanism for downloading programs OTA is in place.

Originally, data transfers over the operator's network were horribly slow, much slower than even dial-up lines. However, over time network operators upgraded their infrastructure to support 2.5G and 3G configurations and protocols so that faster data transfers are possible. It is not uncommon for users to download applications just under 1MB in size. Today with J2ME's OTA capabilities firmly established and with the higher-bandwidth networks available, it is believed that 10 million J2ME downloads occur each month globally.

In conclusion, the evolution and convergence of these technologies made a mobile 3D platform possible. The wireless infrastructure had to be built so that transfers of data and program code were possible to the mobile platform. The Java environment had to evolve (as J2ME) to operate within the confines of a mobile device, support wireless downloads, and offer a standards-based 3D API. And the phones had to become powerful enough to run the compute-intensive tasks required in the display of 3D graphics.

3.5. The Pros and Cons of using Java for Mobile 3D Applications

The J2ME runtime environment, as described previously, is a key component of a portable 3D platform. However, no technology is perfect, and even if it were, compromises are often made when trying to shoehorn it onto a portable device. It's worth taking time to do a reality check on Java to understand its limitations in this area. This is especially important if the goal is to profitably write and deploy 3D applications on mobile phones.

There are several positive capabilities of the Java platform. It provides a standard interface that makes it possible to write an application that can execute with little modification on a wide range of mobile J2ME devices. This single consistent execution environment expands the potential market, even though phone manufacturers often use different processors and chipsets to build families of mobile handsets. Since there are tens of millions of J2ME-enabled mobile phone in use, this represents a huge potential market to tap into. Developers can do the math: Large number of devices = Large user audience = Large revenues.

In addition, inexpensive Java development tools are available. You can, for example, download a free version of Sun Microsystems's Sun ONE Studio 4 Java tool suite. It features an Integrated Development Environment (IDE) with tools that let you write, emulate, and debug J2ME applications. There are also native code development tools (professional IDEs) that are available for purchase.

Of course, there are still some restrictions with Java for application development. The major issue is that because portions of the application are interpreted by a VM, its performance suffers. The Java performance can be improved if the phone manufacturer writes the preloaded API class libraries in native code. However, the context switches between the Java environment and native code can add overhead that saps performance. This situation can be especially problematic for games where such context switches can occur often.

A secondary issue is that subtle implementation issues mar compatibility. Parts of this problem can be traced to the lack of conformance among the various Java VMs. Another contributing factor is the variation in the implementation of optional APIs. For example, each manufacturer's audio playback APIs for their JSR 135 implementation might (and do) support different audio data formats.

Finally the ratification of standards is too slow for the fast-paced mobile phone market. It can easily take up to eighteen months or longer to introduce and ratify a specification that extends the Java APIs. This is about the same time interval as a product cycle for a mobile phone.

In summary, Java provides valid reasons for writing 3D applications, but be aware of the pitfalls. Java does incur a performance penalty. While Java's runtime architecture abstracts the mobile device's hardware and thus promotes application portability across a wide range of devices, you still have to tweak an application for each mobile phone in terms of its screen size, key bindings, and supported audio formats.

On the plus side, Java's hardware independence offers a larger potential market, which means more revenues for the developer. It also offers a modest amount of security (both in application execution and support for encryption mechanisms that can implement secure transactions). While some interface modifications might be required to migrate a Java application to different mobile phones, such changes are minor compared to the cost of completely rewriting and testing the application's core code. Finally, the inexpensive developer tools lower the initial cost of writing Java applications.

3.6. User Expectations

While today's generation of mobile phones stand poised to provide 3D applications and content, it's important to temper the user's expectations in this area. First and foremost, a mobile phone can't compete with a game console in performance—nor should it. A mobile phone has to fit a specific form factor that encourages users to carry it everywhere so that they can place a call and make effective use of data and multimedia services. This places severe constraints on the phone's screen size and computational power.

As a consequence, the scenes in 3D games are going to be less intricate and much smaller than a portable game console's. The complexity of the game play will also be reduced for the same reason. However, the effort of adding a 3D interface can be worth the effort: the user experience will be much better than that of a 2D version of the game.

A mobile phone's wireless capability can compensate for these limitations in certain ways. Because the operator's infrastructure is ready-made to handle both downloads and billing transactions, it's easy to download new game levels for an existing 3D game, or episodic content, or even new 3D applications. Furthermore, the wireless capability can provide new opportunities in which to push application design. Via the operator's network, you can develop 3D applications that offer users, via 3D avatars, the ability to collaborate or challenge one another. It's too soon to tell in what direction the wireless capability will push 3D applications to evolve. However, this opportunity is the result of the powerful mobile phones available on the market today, along with the Java environment and its 3D API extensions that make the large-scale distribution of 3D applications possible.

Now that we understand the capabilities of the mobile 3D platform, let's take a look at the 3D APIs that developers for Sony Ericsson mobile phones will use to write the applications.

3.7. Overview of Mascot Capsule Micro3D version 3

HI Corporation's Mascot Capsule Micro3D version 3 implementation consists of a 3D rendering engine and a Java extension that exposes its proprietary API. To add 3D graphics to a J2ME application (termed a MIDlet), you simply make calls to the Micro3D version 3 methods. No special linking or post processing is required when you build the application's JAR file.

The Micro3D version 3 engine implements all 3D operations with 32-bit integer arithmetic to achieve maximum execution speed on processors that lack an FPU or graphics coprocessor. As a consequence, all coordinate information and arguments that you pass to Micro3D version 3 methods use integer values. This can be slightly jarring for experienced 3D programmers, since most 3D APIs typically deal with floating-point values.

3.7.1. Micro3D version 3 classes

The Mascot Capsule Micro3D version 3 API consists of ten classes that manage the display and control of 3D content. The methods provided by these classes are feature rich and handle the most commonly used 3D graphics operations.

Table 1. Mascot Capsule Micro3D version 3 classes.

Class	Description
ActionTable	Stores the action data that controls the movement of an associated 3D model. Action table data can be read from a JAR resource.
AffineTrans	Handles the matrix math used to scale or move a 3D object.
Effect3D	Contains the rendering effect data that is associated with a particular 3D object. Such effects include transparency, shading type, and lighting.
Figure	Stores a 3D object's geometric information. Also stores position information, which is known as a "pose". This data is read from a JAR resource.
FigureLayout	Container for all of a 3D object's rendering information, such as its position, size, and orientation.
Graphics3D	Implements all rendering functions. Must be bound to a LCDUI Graphics object where the pixels are drawn.
Light	Contains lighting information, such as its direction and intensity.
Texture	Stores the texture data for objects or the environment. The data can be read from a .bmp file stored as a JAR resource.
Util3D	Contains utility methods for use in 3D algorithms, notably <code>sin()</code> , <code>cos()</code> , and <code>sqrt()</code> .
Vector3D	Contains methods that construct 3D vectors or extracts information (such as the vector's x, y, and z components) from them. Also implements some vector math routines.

Some of these classes, such as the `Figure` class, store geometric object information. Other classes, such as `Effect3D` and `FigureLayout`, attach rendering attributes to instances of the geometric objects. Other classes describe the lighting and textures that a scene uses. The `Graphics3D` class handles all rendering operations, while a utility class provides methods that can help with the design of graphics algorithms. Table 1 provides a brief summary of these classes.

For more information on the Micro3D version 3 APIs, consult the Javadoc documentation that accompanies the Sony Ericsson J2ME SDK.

3.7.2. Graphics pipeline

Micro3D version 3, like many 3D graphics APIs, operates in an *immediate mode* where graphics commands are issued into a graphics pipeline and the rendering engine executes them immediately. For Micro3D version 3, these commands implement the following functions:

- 3D object descriptions—These commands describe the coordinates and colors of triangles, polygons, and quad surfaces that comprise the 3D object. They also specify the texture maps and the color blending processes that are to be applied to each object. The object descriptions also support point sprites.
- 3D environment configuration—These commands adjust the position and characteristics of a scene's lighting, its viewing geometry (parallel or perspective), the type of shading to be applied, and whether to use semitransparent blending.

- Manage the rendering process—These commands describe the type of color blending used in the rendering operation and specify the clipping region's coordinates. They also instruct the engine to execute a transformation (such as a rotation or motion) on an object. A `flush` command indicates when the scene is to be rendered and drawn to the device's screen.
- Graphics primitives—These commands draw points, lines, triangles, and quads.

Commands can be stored in lists. This allows a 3D object to be stored as a list of commands that specify the object's shape, its color, and its shading and blending characteristics. To render the object—say, a robot—your application would issue the command list that describes and draws the robot into the Micro3D version 3 graphics pipeline.

You can't organize `Figure`s hierarchically to assemble a graphics "world". This is a characteristic of retained mode operation, which Micro3D version 3 doesn't support. Note that there are commands for either drawing or rendering a `Figure`. The `render` command processes all of a `Figure`'s transformations and effect computations, but the actual generation of pixel data is deferred until a `flush` command executes. The `draw` command performs all of a `Figure`'s rendering computations and the pixels are generated immediately. This allows you to mimic a 3D retained mode by first arranging and updating a scene's `Figure` objects with `render` commands, and then perform a final `flush` command to draw the entire scene. In many cases, Micro3D version 3's rendering scheme will be suitable for most 3D operations.

3.7.3. Object Import

Because it is difficult to describe any 3D object of moderate complexity using graphics commands, Micro3D version 3 allows you to import 3D object data that was generated by 3D authoring programs. A plug-in module exports 3D model data and its attributes into a file with an extension of `.bac`. After being processed by a translation program, this data can be stored as a resource in the MIDlet's JAR file. The resource is then loaded into an instance of Micro3D version 3 `Figure` class, which now contains the 3D object information. Various attributes, such as an object's shading type, is also imported and associated with the corresponding instance of this class. Other associations specify operations such as rotation, scaling, or motion for the `Figure`. Command lists can also be attached to each instance of `Figure`.

The `Figure` class also helps implement animation effects for a model. The model's current arrangement of 3D objects represents a *pose*. An action description class, `ActionTable`, is associated with the `Figure` and stores the motions that change the model's pose. The action tables animate the model by changing its pose over time under program control. Animation effects generated in 3D authoring programs can be exported as `ActionTables` into a file with an extension of `.tra`. After being processed by a translation program, the files are also stored as JAR file resources, and their contents imported into an instance of `ActionTable`.

Micro3D version 3's ability to import object and action data allows a developer to use desktop 3D authoring programs to generate 3D models and animate them for use on a mobile platform.

3.8. Overview of JSR 184

JSR 184 is an optional package, in that its use is not mandated by any existing J2ME profile, such as MIDP 2.0. This requires that your Java application check for the package's existence before invoking any of the JSR 184 APIs. To discover this package, call `System.getProperty` with a key of `microedition.m3g`. If the API is present, the call returns the JSR 184 API's version number. Otherwise, it returns null.

All of the Sony Ericsson K500, K700, S700, and Z500 series, as well as the Vodaphone-exclusive F500i and V800 mobile phone families currently offer the JSR 184 package.

The JSR 184 implementation consists of API classes and a rendering engine. Because JSR 184 must function on platforms without the benefit of an FPU, the 3D engine and all internal operations use integer arithmetic to achieve optimal execution performance. Its architecture was designed so that the JSR 184 classes can take advantage of any FPU or hardware accelerators if present.

JSR 184's APIs follow typical 3D conventions and require that you specify floating-point values for coordinates, transforms, and other 3D information. The floating-point numbers must conform to the Java language's `float` data type. As a consequence of this requirement, any mobile platform using JSR 184 must be based on CLDC version 1.1 or later, where this data type is defined.

3.8.1. JSR 184 classes

Compared to Micro3D version 3, JSR 184 contains lots of classes: thirty in all, as Table 2 shows. The large number of classes increases the implementation's firmware footprint. However, the wide variety of classes enables more specialized 3D operations—not just the common ones—to be managed with just a few API calls. For example, a `Background` class allows you to apply a backdrop image to a 3D scene, and the JSR 184 automatically handles its repainting as other 3D objects move in front of it. JSR 184's many classes make it capable of handling many different 3D operations, which in turn should reduce the code size of 3D MIDlets. In addition, the API has been designed to reduce RAM use by storing most objects by reference, not by making new instances of them.

Table 2. JSR 184 classes.

Class	Description
<code>AnimationController</code>	Manages the location and speed of a collection of objects that comprise an animation sequence.
<code>AnimationTrack</code>	Contains the information that controls a single animation property on one target object. Animation sequences consist of a set of <code>AnimationTracks</code> handled by an <code>AnimationController</code> .
<code>Appearance</code>	Stores the rendering attributes of a set of component objects. The attributes describe each object's material characteristics, its polygons, how it is to be blended into the scene, and any fog effects. It also specifies texture map characteristics and the images involved.
<code>Background</code>	Used to specify a color or image that clears or fills the given viewport (a drawing area).
<code>Camera</code>	The node in a scene graph that establishes the scene's point of view. Used to render the scene graph from 3D to 2D. It also establishes what elements in the scene graph are visible (clipping).
<code>CompositingMode</code>	An <code>Appearance</code> component that contains the attributes used in pixel compositing operations.
<code>Fog</code>	An <code>Appearance</code> component that contains the attributes used to apply a fog effect to pixels.
<code>Graphics3D</code>	A graphics context that's applied when rendering an image. This object binds to a rendering target, which is <code>Canvas</code> , mutable <code>Image</code> , or a <code>CustomItem</code> . The rendering target receives the rendering operation's output. This class handles all of the drawing for this API.
<code>Group</code>	A scene graph node that stores a collection of unordered nodes as its children.
<code>Image2D</code>	Stores a two-dimensional image that can be used as a texture, a background, or a sprite image.
<code>IndexBuffer</code>	Describes how to connect vertices so that they assemble a geometric object. As it is an abstract class, you use <code>TriangleStripArrays</code> to construct objects.
<code>KeyFrameSequence</code>	Stores animation data. The data is stored as a sequence of time-stamped, vector-valued keyframes. Each keyframe stores the value of an animation characteristic at a given moment in time.

Light	A scene graph node that represents a light source. Lights have color, intensity, and a type (ambient, directional, omnidirectional, and spot).
Loader	Downloads node components, scene graph nodes, and entire scene graphs for use in graphics operations. The data can contain <code>Cameras</code> and <code>LightS</code> , <code>Appearance</code> and <code>Material</code> attribute classes, and animation classes such as <code>AnimationTrack</code> and <code>KeyFrameSequence</code> . Used to import ready-made 3D content stored as a resource in the application's JAR file.
Material	An <code>Appearance</code> component that stores material attributes used in lighting calculations.
Mesh	A scene graph node that represents a 3D object. The object is described with polygonal surfaces. A <code>Mesh</code> consists of triangle strips defined in an <code>IndexBuffer</code> instance, along with its visual properties, which are stored in instances of <code>Appearance</code> .
MorphingMesh	A scene graph node that represents a vertex morphing polygon mesh. <code>MorphingMesh</code> is similar to <code>Mesh</code> , but its vertices are calculated using weighted values in <code>VertexBuffers</code> . This enables a <code>MorphingMesh</code> object to change its shape.
Node	An abstract base class for all scene graph nodes. <code>Camera</code> , <code>Group</code> , <code>Light</code> , <code>Mesh</code> , and <code>Sprite3D</code> are all sub-classed from it.
Object3D	An abstract base class for all objects that can be part of a 3D world.
PolygonMode	An <code>Appearance</code> component that contains polygon-level attributes. These attributes include back/front face culling, lighting computations, perspective corrections, shading, and winding.
RayIntersection	An object that contains rays added to it by <code>Group</code> 's <code>pick()</code> method. <code>RayIntersection</code> stores references to the meshes or sprites that intersect each ray, and information about the intersection point.
SkinnedMesh	A scene graph node that represents a skeletally animated polygon mesh. Allows groups of vertices to change independently of one another while smoothly deforming the polygon mesh. Provides an efficient means of animating characters.
Sprite3D	A scene graph node that represents a 2D image with a 3D position.
Texture2D	An <code>Appearance</code> component that stores a 2D texture image and the attributes that direct how the image is applied to submeshes.
Transform	A 4 by 4 matrix that contains values used to perform a transformation operation.
Transformable	An abstract base class for <code>Node</code> and <code>Texture2D</code> . It defines common methods used to handle node and texture transformations.
TriangleStripArray	Defines an array of triangle strips that are used to construct a geometric object.
VertexArray	An array of integer vectors that represent vertex positions, normals, colors, or texture coordinates.

VertexBuffer	Contains references to the <code>VertexArrays</code> that define the characteristics for a set of vertices.
World	A special <code>Group</code> node that is the top-level container for scene graphs.

JSR 184 supports both retained and immediate modes of operation. The *retained mode* uses scene graph that links all of the geometric objects in a 3D world via a structured tree of nodes. Each node on the graph represents a geometric object and carries information about its appearance, position in space, and how it behaves in relation to other nodes. You use objects sub-classed from `Node` (such as `Lights`, `Sprites`, and `Meshes`) to assemble the 3D world. The `Group` class lets you gather and organize unordered node objects together, and the `World` class defines a special `Group` node that acts as the top-level container for all of the nodes in the 3D world. To display a view of the 3D world using the retained mode, you execute a `Graphics3D` `render` method on the `World` node.

Two classes, `TriangleStripArray` and `VertexArray`, serve as building blocks from which you can assemble 3D objects and models. A sub-class of `Node`, `Mesh`, is used to build more complex 3D geometric objects. Other `Node` subclasses control the scene graph's lighting and point of view (`Light` and `Camera`, respectively). There are classes that define a 3D object's visual properties (`Material`, `Fog`, `CompositingMode`, and others) and they can be used in either the immediate mode or retained mode.

Interestingly, neither JSR 184 nor Micro3D version 3 supplies a method for handling collision detection between sprites or 3D models.

A detailed description of these classes is outside of the scope of this paper. For more information, consult the JSR 184 specification documentation at <http://www.jcp.org/en/jsr/detail?id=184>.

3.8.2. JSR 184 rendering modes and graphics pipeline

As mentioned previously, JSR 184 supports retained and immediate modes of scene rendering. The advantage to JSR 184's retained mode is that it allows less 3D-savvy developers to readily build complex 3D models. For example, you can use a scene graph to assemble a dune buggy that has rotating wheels and movable doors. The nodes that represent the wheels can not only specify how they appear, but also that they can rotate about their axes and are constrained to be parallel to the orientation of the dune buggy's body. You could then animate this vehicle under program control, either having the vehicle change direction according to steering commands issued by a user interface, or in response to the environment (like bouncing over a hill, or rebounding from a wall). To execute a retained mode operation, you invoke `Graphics3D`'s `render` method on either a `World` node or a `Group` node.

The retained mode simplifies the design of a 3D world by hiding lots of low-level gritty technical details. However, experienced 3D developers will want full control of the 3D scene's rendering. For a mobile platform with limited processing power, access to the graphics pipeline is critical to reduce processing overhead, shorten an image's rendering time, and improve the MIDlet's responsiveness to user events. The low-level approach also exposes geometric data such that 3D graphics hardware accelerators on high-end platforms can boost the speed of the rendering process. To this end, JSR 184 also supports an immediate mode, where you invoke the `render` method on sub-mesh objects (actually objects that consist of arrays of vertex information and attributes). Unlike Micro3D version 3, JSR 184 doesn't implement graphics commands that can be stored into lists and poured into the graphics pipeline.

Because the retained mode capabilities are built on top of the immediate mode operations, the retained mode also benefits from any graphics acceleration hardware, capabilities that are likely to be seen in future mobile phones. This design approach also allows the simultaneous use of both retained and immediate mode operations with JSR 184. The ability to use both modes allows a programmer to put the limited resources of the mobile phone to their best possible use: you can effectively balance rendering speed versus resource consumption by using the appropriate mode for different 3D graphics tasks.

3.8.3. Object Import

Like Micro3D version 3, JSR 184 allows you to import 3D content that contains geometric object data and animation information. Object data can also carry attribute information, such as material, fog, texture, compositing mode, and others. This 3D content can be generated on PCs and workstations using 3D authoring programs, and using a plug-in translator to export the content into an `.m3g` file. The JSR 184 specification contains a detailed description of this file's format. The file can be stored as a JAR resource in the MIDlet, where it is read by the

`Loader` class. The `Loader` generates instances of 3D objects as it reads the resource. Entire scene graphs with animation, lighting, and camera views can be imported and displayed.

Because the format of the `.m3g` file is an open standard, it enables 3D content to be generated on high-end platforms, then readily imported or shared among a wide variety of mobile platforms that implement JSR 184. The ability of both Micro3D version 3 and JSR 184 to import 3D content allows developers to use the same PC 3D authoring programs and design skills to generate content for mobile phone games.

Now let's consider how these APIs can affect the game engine and the 3D platform itself.

3.9. Impact on game engines and the platform

Games are typically designed as a user interface wrapped around a game engine. The game engine contains the core code that implements the bulk of a game's logic. It might be as simple as an event loop that spins endlessly, executing game actions in response to user events or play rules. More often the game engine is a complex package of code that handles collision detection, manages the behaviour of objects in response to events such as collisions and other actions, and updates the location and status of objects. For example, if a torpedo hits an asteroid, there's an expanding explosion where pieces of rocky debris are created, each with its own trajectory.

For 3D games, the game engine might also implement 3D rendering operations, particularly if the platform offers little in the way of 3D support. If the engine does its own 3D rendering, it can be quite large in size.

The availability of 3D APIs on the mobile platform affects the design of a game engine in several ways:

- They define a standard interface which developers can write to. The APIs serve as a hardware abstraction layer, sparing a developer from learning a morass of hardware register details when they want to execute a 3D graphics operation. Writing to APIs rather than registers also promotes code portability, especially on the J2ME platform where the execution environment is nearly identical among a manufacturer's family of mobile phones.
- They define a standard set of functions for phone manufacturers. While there's going to be a disparity between the hardware features of a manufacturer's low-end mobile phone versus its high-end phones, the 3D APIs encourage phone manufacturers to provide the same 3D functions across their entire product lines. This eases program development and code portability because there's a common set of functions always available, regardless of the device's hardware capabilities. In a sense, the 3D APIs define a level playing field of features and functions for hardware vendors to strive for. Because both Micro3D version 3 and JSR 184 assume no hardware acceleration or FPU support, it is relatively easy for low-end phones to support these APIs as well.
- They reduce the complexity and code footprint of the game engine. Given a common set of 3D functions, a lot of the rendering tasks that the engine formerly performed can now be handed off to the 3D API. This reduces the size of the game engine code—always a plus—and improves reliability because the developer doesn't have to roll his own 3D functions for the engine and test them. This also improves performance since the API methods are implemented as native code.

The addition of 3D APIs to a mobile phone therefore brings important benefits to the device. In summary, these are: improved 3D MIDlet portability, smaller MIDlets, and better performance. These benefits in turn can help differentiate the 3D-enabled mobile phone in such a highly competitive market.

3.10. Where it's all going

This paper has already discussed how several key trends, notably that the adoption of J2ME, the appearance of more powerful processors, capable hardware, and wireless connectivity have made the 3D-capable mobile platform possible. What happens next?

Currently, 3D mobile phones are no match for the performance of portable game consoles. In fact, today's 3D phones are just capable of managing some 3D graphics in real time. Still, this is an impressive performance for a battery-powered device whose primary purpose is being a phone, and not a dedicated game console.

Looking ahead, we can expect the 3D graphics performance of mobile phones to improve. Part of this will be thanks to faster processor speeds, but a large amount of the growth in 3D graphics performance will be determined by the appearance of low-power graphics acceleration hardware and FPUs within the device. There is already evidence in the industry that this trend is happening: QUALCOMM has announced that it will incorporate ATI Technologies Inc.'s 3D graphics core into its wireless chipset. Not to be left out of the fray, Nvidia Corp. has added a 3D core to its GoForce wireless media processor line.

Another trend you can expect to see is faster wireless data rates as the demand for 3D applications grow. Since 3D applications tend to be a lot larger than typical MIDlets, any boost in wireless data throughput will be welcome. Whether the better bandwidth comes about by improvements to the operator's network or by adding WiFi (802.11b) capabilities to the mobile phone remains to be seen. Another possible means of 3D application distribution will be through memory cards.

4. Building Java Mobile 3D MIDlets

Because both Micro3D version 3 and JSR 184 are extensions to the Java environment, using them to write a 3D MIDlet is straightforward. You add the API calls to your Java source code, then build and debug the MIDlet with your choice of development tools.

4.1. Java Development Tools

A few Java Integrated Development Environments (IDEs) are available for writing MIDlets. Two of the most popular are: Sun Microsystems' Sun ONE Studio 4 update 1 Mobile Edition, and Borland's JBuilder X Mobile Edition. Both toolsets have a GUI where you can edit, compile, and debug Java MIDlets. A simulator on the Windows PC emulates the J2ME runtime environment and executes the MIDlet's code for testing. The emulated MIDlet can be run under debugger control. In this mode you use the IDE to step through the source code and display variable values. To interact with the program or view its graphic output, the emulator uses a phone "skin" to present the target device's user interface. The skin appears as a bit-mapped image of the mobile phone, and you can click on its keypad to enter numbers or key presses for soft keys and game keys. The 3D output appears on the skin's representation of the mobile phone screen. As Figure 1 shows, the emulator and skin work in concert with the IDE's debugger so that you can set breakpoints in the MIDlet's code that trigger when you, say, click a virtual softkey on the skin's keypad.

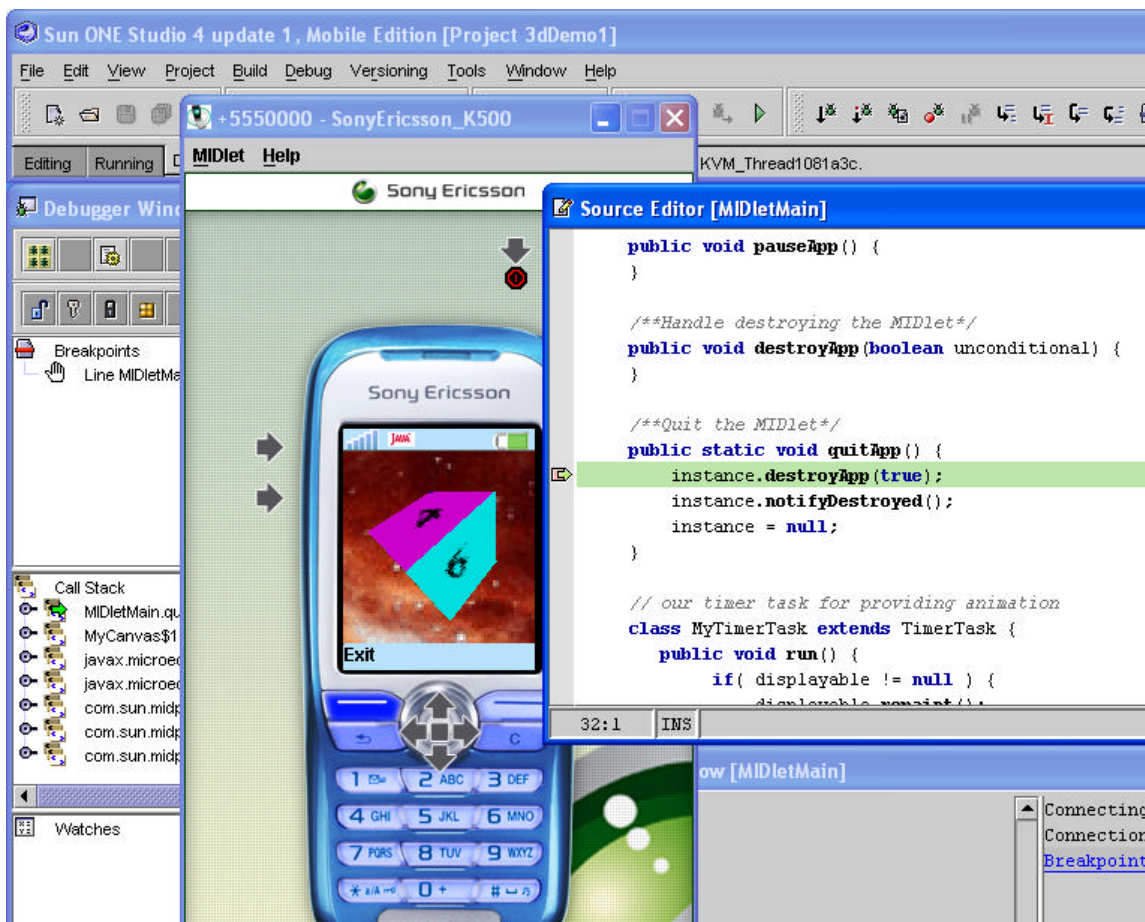


Figure 1 – Debugging a JSR 184 3D MIDlet with the Sun One Studio 4 IDE, Sony Ericsson SDK emulator, and K500 skin.

The free version of Sun One Studio 4 update 1 Mobile Edition can be downloaded from the Sun Web site at: http://www.sun.com/software/sundev/previous/studio_me/index.html.

A trial version of JBuilder X Mobile Edition can be obtained from the Borland Web site at: <http://www.borland.com/mobile/jbuilder/>.

These tools require the Sony Ericsson J2ME SDK, which contains the Micro3D version 3 and the JSR 184 API classes, some API documentation, and the Sony Ericsson J2ME phone emulators. The SDK is discussed next.

4.2. Sony Ericsson J2ME SDK

The Sony Ericsson J2ME SDK (hereafter known as the “SDK”) contains:

- Mascot Capsule Micro3D Version 3 API and API documentation.
- Java Mobile 3D Graphics API (JSR 184). For the API documentation, download the Javadoc from the JCP Web site at: <http://www.jcp.org/en/jsr/detail?id=184>.
- Micro3D version 3 emulation.
- JSR 184 emulation.
- Reference documents for MIDP 1.0 and MIDP 2.0 implementations in Sony Ericsson mobile phones.
- Developers’ Guidelines documents to configure the Sun and Borland IDEs to function with the SDK.

The SDK also contains:

- Device emulation, device explorer, and device connection proxy tools.
- On Device Debugging (ODD) support for MIDlets. ODD allows developers to perform real-time source-level debugging of MIDlets directly on those Sony Ericsson handsets that support this capability. You can use ODD to debug code that uses MIDP 2.0, Micro3D version 3, or JSR 184 APIs on all of the Sony Ericsson phones mentioned in this paper.

Sony Ericsson is currently the only handset manufacturer to provide an SDK that supports full 3D emulation as well as On-Device Debugging of 3D applications on mobile phones. The SDK can be downloaded from the Sony Ericsson Developer World web site at http://developer.sonyericsson.com/site/global/docstools/java/p_java.jsp.

At the time of this writing, the current version of the SDK is 2.1.4 beta.

In order to use the SDK, you must have the SDK for J2SE installed on your Windows PC. Version 1.4.2 or later is recommended.

Now let’s turn our attention to the mobile phones that support the 3D APIs.

4.3. 3D Capable Mobile Phones

Sony Ericsson’s 3D-capable mobile phones offer a rich set of J2ME features on top of their support for Micro3D version 3 and JSR 184 APIs. The general capabilities of these devices are summarized in the next section.

4.3.1. Basic hardware and software specifications

As hinted at in the discussion of what makes a 3D-capable platform, the hardware and software specifications for a Sony Ericsson 3D-capable phone are impressive. Table 4 provides the specifications for one such Sony Ericsson mobile phone, the K700 series.

Table 4 – Overview of the Sony Ericsson K700 hardware and software features.

J2ME APIs	MIDP 2.0, CLDC 1.1, MMAPI, WMA, JSR 184, Micro3D version 3
Audio formats	iMelody 1.2, MIDI (40 voices), AMR, MP3, MP4, 3GP
Display size	176 W by 220 H
Color depth	16-bit, 65536 colors
RAM	41 MB



Other 3D-capable Sony Ericsson phones currently are:

- **Sony Ericsson Z500 series**
- **Sony Ericsson K500 series**
- **Sony Ericsson S700 series**
- **Sony Ericsson F500i**
- **Sony Ericsson V800**

All of these phones have features similar to the K700 series. For more information on a particular phone, read either its White Paper, or the J2ME Developers' Guidelines, both which are all available for download from the Sony Ericsson Developer World web site.

Now that we understand the 3D APIs and the mobile phones that they'll execute on, it's time to make a 3D MIDlet.

4.4. Running an Example 3D MIDlet

In this section we'll show how to make two simple 3D MIDlets, one that uses the JSR 184 API and the other that uses the Micro3D version 3 API. The MIDlets demonstrate how to use both of these APIs to produce the same 3D animation. Because the example code uses immediate mode operations, no 3D content files are required.

There are two archives, 3D_Demo1.zip and 3D_Demo2.zip. When these archives are expanded, they generate the directories, 3D_Demo1 and 3D_Demo2. The contents of these directories are:

- **3D_Demo1** – This directory contains two source files, `MIDletMain.java` and `MyCanvas.java`. The example code uses the JSR 184 API. The subdirectory `res` stores the image files `cubeface.png` and `backdrop.png` that serve as a texture map and a background image, respectively.
- **3D_Demo2** – This directory contains two source files, `SampleApp.java` and `SampleCanvas.java`. The example code uses the Micro3D version 3 API. The subdirectory `res` stores the image files `cubeface.bmp` and `backdrop.png` that serve as a texture map and a background image. Note that Micro3D version 3 uses 8-bit BMP format for its texture map image.

The code in the `MIDletMain.java` and `SampleApp.java` files is identical; the file names were changed to prevent confusion. These files perform the standard MIDlet initialization and start a timer thread that implements the scene's animation. The initialization code next creates an instance of a custom `Canvas`. Its `paint` method makes 3D API calls that generate a cube and apply the image in `cubeface.png` or `cubeface.bmp` as a texture map onto its faces. The method also performs a matrix transformation that changes the cube's orientation slightly. The timer thread periodically calls the `Canvas`'s `repaint()` method, which continues to change the cube's orientation and thus generates the rotation effect. These simple demos show that if the MIDlet is carefully designed, it is possible to substitute the code that makes JSR 184 API calls with the code that makes Micro3D version 3 calls without modifying the MIDlet's core logic.

This paper uses the Sun ONE Studio 4 Mobile Edition IDE to illustrate how to build the example MIDlets. We'll assume that the Sony Ericsson J2ME SDK 2.1.4 and any supporting software have been installed, and that the IDE has been configured to use the SDK. For information on how to configure the Sun ONE Studio 4 Mobile Edition IDE, consult the Developers' Guidelines document, *Developing MIDlets with the Sony Ericsson J2ME SDK*, that comes with the SDK.

4.4.1. Make the project

The first thing we want to do is make a project for the example code. A *project* acts as a virtual container that holds the MIDlet's source files, class files, plus any JAR files generated from the build operations. The IDE performs automatic book-keeping for us, flagging a source file to be recompiled when we modify it with the IDE's editor.

To make a project for the JSR 184 example code, follow the sequence of steps described next.

- 1) Launch the IDE. Now choose **Project | Project Manager**.
 - 2) In the **Project Manager** window that appears, click on the **New...** button.
 - 3) A **Create New Project** dialog appears. Type in `3D_Demo1` and click **OK**.
 - 4) Now the IDE displays a **Project Configuration (1 of 1)** window. Since you're going to be building a J2ME MIDlet, click on the **Mobile Information Device Profile (CLDC/MIDP)** radio button. Now click **Finish**.
 - 5) An **Explorer [Filesystems]** and **Properties** windows should be visible, and the **Explorer [Filesystems]** window displays a **Filesystems** icon.
- Click on the **Filesystems** icon to select it, and then right-click it again. A drop-down menu appears. Select **Mount > Local Directory**.
- 6) The IDE presents a **New Wizard – Local Directory** window. Use its controls to navigate to the `3D_Demo1` directory. Click **Finish**.
 - 7) The `3D_Demo1` directory appears as an icon the **Explorer [Filesystems]** window.
- Click on `3D_Demo1`'s directory tree control to expose the source code files (Figure 2).



Figure 2 – The Sun ONE Studio 4 Mobile Edition IDE displaying the JSR 184 example code files.

4.4.2. Build and run the MIDlet

Now that we've got a project made and the IDE recognizes the source files within it, let's compile and execute the MIDlet.

- 1) Click on the `MIDletMain` file icon to select it.

2) Right-click on the MIDletMain file icon and choose **Build** from the drop-down menu.

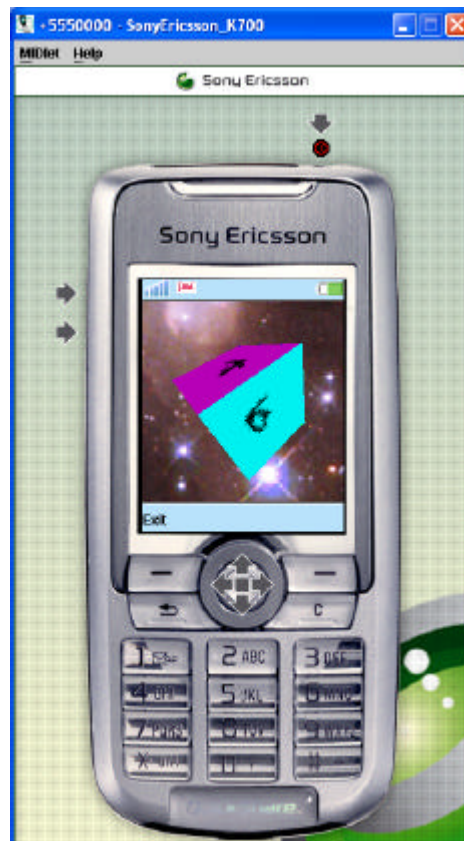


Figure 3 – The JSR 184 example program executing on the emulator and displayed on a Sony Ericsson K700 skin.

The IDE compiles the files. An **Output Window [Compiler]** appears, and displays the status of the operation. If all goes well, both files are compiled and this window displays a `Finished MIDletMain` message.

3) Right-click on the MIDletMain file icon and pick **Execute** from the drop-down menu.

An **Execution [Execution View]** and **Output Window [MIDletMain – I/O]** windows appear. The **Execution** window displays the current running process, `MIDletMain`. The **Output** window presents status information and reports any exceptions that occur as the MIDlet runs. Finally, a phone skin appears, with the 3D graphics appearing on the phone's screen. Figure 3 shows the MIDlet's 3D graphics appearing on the skin for a Sony Ericsson K700 mobile phone.

4) To exit the MIDlet, click on the left softkey that's defined as **Exit**. The skin disappears and a summary of the JVM's operation appears in the **Output** window. As this short trial run demonstrates, building and running 3D-capable MIDlets is easy. The second example program, which uses the Micro3D version 3 API, can be built and executed with the exact same sequence of steps, other than selecting the `3D_Demo2` directory for the project instead. Because of the way the Sony Ericsson organized the SDK, a developer doesn't have to modify the IDE's configuration to switch between the two 3D APIs. The developer simply uses the `import` statement to specify the proper 3D package, inserts the API calls into the Java code, then builds and executes the MIDlet.

4.4.3. Debugging a MIDlet

As you write that killer 3D application, things will inevitably go wrong, mostly due to programming errors in your code. The SDK, with the IDE's source level debugger, lets you debug faulty code at the source code level. Let's explore the capabilities of source-level debugging as provided both by the IDE and the SDK.

First, use the steps described previously to have the IDE make a second project named **3D_Demo2** and use the `3D_Demo2` directory. This directory contains the version of the 3D MIDlet that uses the Micro3D version 3 API. Build the MIDlet.

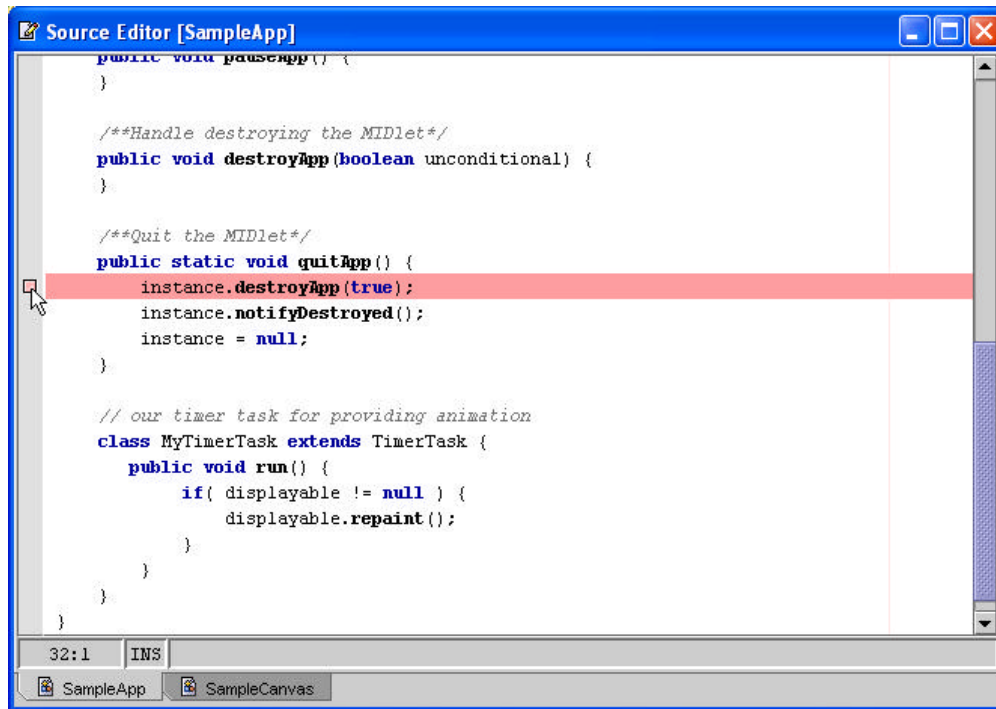


Figure 4 – Setting a breakpoint in SampleApp's quitApp method.

Now, double-click on the `SampleApp` icon in the **Explorer [Filesystems]** window. The IDE's built-in editor opens the `SampleApp.java` file and displays its source in a window. Scroll through the source until you find the `quitApp` method. Click on the left window bar at this method's first line of code. A red block, which represents a breakpoint, appears in the left bar (Figure 4).

Click the `SampleApp` file icon in the **Explorer [Filesystems]** window to select it. Choose **Debug | Start** to launch the MIDlet under debugger control. A **Debugger Window** appears, as does a phone skin. The MIDlet starts rotating a cube, as before. Let the cube spin for a while, then press the **Exit** softkey. The animation should stop, and the Editor window appears, with the source line of the designated breakpoint highlighted in green (Figure 5). You can examine the MIDlet's call stack and other information through controls on the **Debugger Window** panes. For now, just pick **Debug | Continue** to allow the MIDlet to terminate.

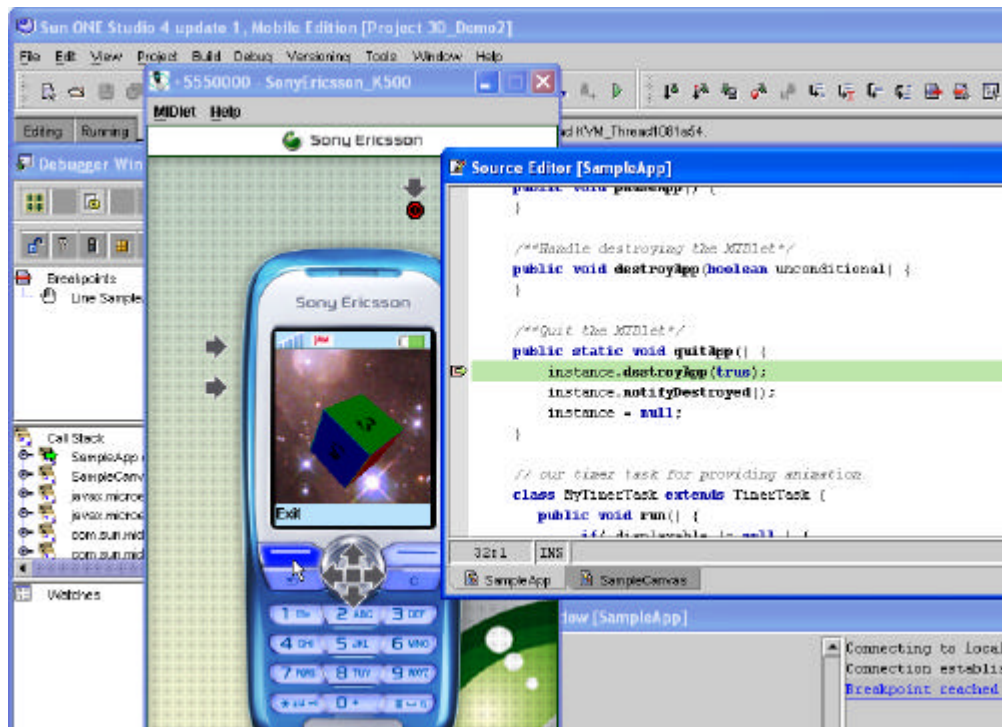


Figure 5 – Using the IDE’s source level debugger on a Mirco3D MIDlet.

As these simple descriptions illustrate, a developer can easily build and debug 3D MIDlets using GUI-based tools and the Sony Ericsson SDK.

4.5. File Translation/Export

Both JSR 184 and Micro3D version 3 APIs can import 3D content that contains model geometry and appearance data, along with information to animate the models. This allows the work of generating the 3D content to be off-loaded onto a PC or workstation. In addition, a professional graphics designer with no programming skills can use 3D authoring software to create the 3D models and generate the animations for a programming project.

As is, the files produced by these authoring tools aren’t compatible with the object loaders present in either JSR 184 or Micro3D version 3. However, HI Corporation offers a number of plug-ins that can be used from within these authoring tools to export the model and animation data to the appropriate files. For JSR 184, entire scene graphs can be captured and exported. These files can subsequently be added to the MIDlet as a resource, and the loader can import the content on demand.

HI Corporation offers plug-ins for both their proprietary Micro3D version 3 API and for the JSR 184 API. For JSR 184, there is also a plug-in that allows direct output from 3ds max 7.

4.5.1. Micro3D version 3 file translation chain

HI Corporation offers Micro3D version 3 file export plug-ins for the following 3D authoring applications: Newtek’s Lightwave, Discreet’s Studio 3D Max, Avid System’s SoftImage, and Alias System’s Maya. These plug-ins export the 3D model geometry data and the animation action data into intermediate files. For 3D geometry data, the intermediate files have an extension of `.bac`. For the animation information, the intermediate files have an extension of `.tra`.

Once the model and animation/action data are exported into their intermediate formats, you use several HI Corporation utility tools for post processing and final translation. The `PAC.exe` utility lets you view and adjust the attributes of certain types of polygonal data in the intermediate model data file. A converter utility application, `Micro3D_Converter.exe`, translates the intermediate 3D data and action files into file formats that Micro3D version 3 can import. The file extensions for the translated 3D geometry data and action information are `.mbac` and `.mtra`, respectively. These files are incorporated into the file JAR files as resources. A `PVMicro.exe` application lets you view the 3D models in their native Micro3D version 3 format for final check out.

For texture maps, any drawing or painting program that can save pixel images as BMP-formatted files will do.

Table 3 summarizes the files involved in the export/translation process.

Table 3: File types and extensions used in the export/translation process of 3D model and animation data for use by Micro3D version 3.

File extension	Description
.bac	Intermediate file that contains the 3D model data exported from the 3D authoring program.
.tra	Intermediate file that contains animation (action) data exported from the 3D authoring program.
.mbac	Final file that contains the 3D model data translated for use in Micro3D version 3.
.mtra	Final file that contains the 3D action data translated for use in Micro3D version 3.
.bmp	Stores 8-bit texture image data.

Figure 6 schematically displays the data generation/export and translation process for 3D content for use by Micro3D version 3.

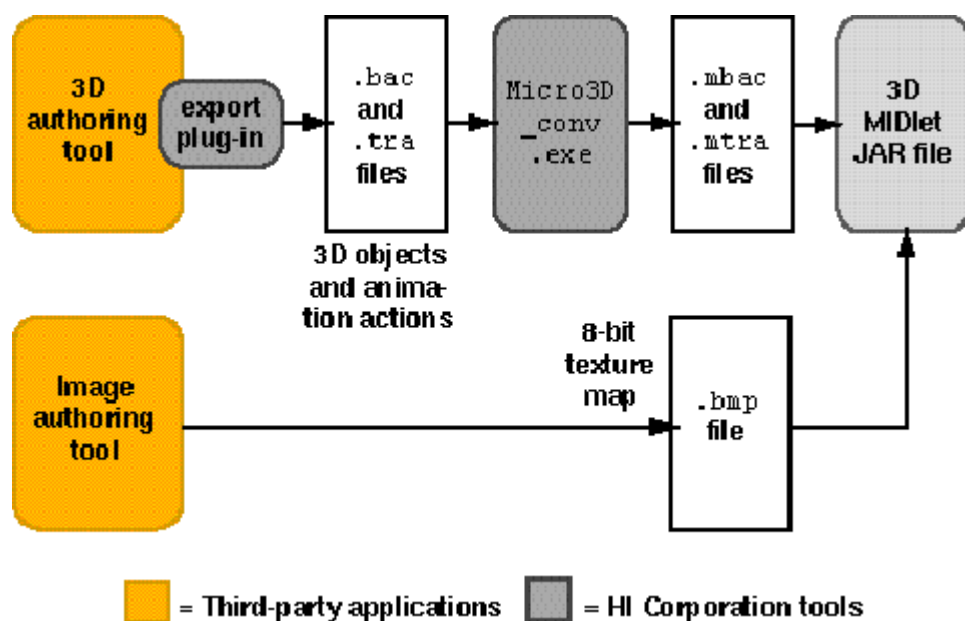


Figure 6 – How 3D model data and animation actions are exported and translated into Micro3D’s internal file format. Not shown is the use of HI Corporation’s polygon attribute editor and preview tools.

All of the plug-ins, tools, and the documentation necessary to manage the export/translation process are available from Mascot Capsule Web site at http://www.mascotcapsule.com/toolkit/sony_ericsson/.

4.5.2. JSR 184 file translation chain

The same 3D authoring applications discussed for generating Micro3D version 3 content can be used to create data for use in a JSR 184 MIDlet. HI Corporation provides a number of plug-ins that can be used from within these authoring applications to export the model and animation data into an intermediate file.

Texture maps can be made in any quality drawing or painting program and saved as a `.png` file. Be sure that the dimensions of the image are a non-negative power of two. For example, a texture map image might be 32 pixels wide by 64 pixels tall.

Table 4 summarizes the file types involved in the export/translation procedure.

Table 4: File types and extensions used in the export/translation process of 3D model and animation data for use by JSR 184.

File extension	Description
<code>.h3t</code>	Intermediate file that contains the 3D model data and animation actions exported from the 3D authoring program.
<code>.m3g</code>	Final file that contains the 3D model data and animation information translated for use in JSR 184.
<code>.png</code>	Stores texture image data.

After the model and action data is exported into the intermediate `.h3t` file, two utility tools for post-processing and final format translation can be applied to it. The `V4Converter.exe` utility lets you translate `.h3t` data to `.m3g` data. A `V4Examiner.exe` tool allows you to view and verify the contents of `.h3t` and `.m3g` files.

All of the plug-ins, tools, and documentation necessary to manage the export/translation process are available from Mascot Capsule web site at http://www.mascotcapsule.com/M3G/download/e_index.html. These tools are provided by HI Corporation and all feedback and trouble reports should be submitted to this web site.

4.6. Tips and Tricks

At the level of compiling and testing 3D-capable MIDlets, things look fairly equal between the two 3D APIs. As you might expect, when you start examining the situation at the code level, the differences are larger. It must be emphasized that the explanation of these differences does not imply a criticism of a particular API or implementation. The paper is simply stating the capabilities of each API and the occasional implementation detail that a developer needs to be aware of when writing a 3D MIDlet using either of them.

4.6.1. 3D object descriptions for immediate mode

Both of the example programs use immediate mode operations to draw the cube and apply a texture map. The example MIDlet that uses JSR 184 APIs invokes the `render` method on an array of vertices that contain the cube's geometry data. Similarly, the example Micro3D version 3 MIDlet calls its `renderPrimitives` method on an array of vertices carrying geometry data. Although these techniques seem identical, they are not.

One common way to describe a 3D object is to use an assembly of triangles that approximate the object's surface in space. The example programs store the cube's information as triangle primitives, where a *primitive* is a basic geometric entity within a data structure. For JSR 184, a class called a `TriangleStripArray` stores the vertex information for these triangles. The Micro3D version 3 example code provides `renderPrimitives` with a buffer that contains the vertex data for a set of triangles.

Since both APIs use triangle primitives to describe the cube's surface, it's tempting to consider sharing the same vertex array data between the two programs. Unfortunately, the way `TriangleStripArray` stores vertex information thwarts this reasonable attempt at data reuse. That's because triangle strips (they're also sometimes termed meshes) store geometric information a different yet memory efficient way.

A *triangle strip* consists of a series of triangles that share vertices. Stated another way, a new triangle is placed on the strip by specifying a new vertex while reusing two vertices from a previous triangle. This arrangement has the advantage in that adding a new triangle to the strip only needs the description of one extra vertex, not three. Specifying a strip of n triangles therefore requires $n + 2$ vertex descriptions. The economy of this scheme becomes evident when you consider that a typical representation of a geometric model might require hundreds of triangles. For example, a triangle strip containing 100 triangles requires 102 vertex descriptions, whereas a conventional arrangement of 100 triangles, where each primitive contains all three vertices, needs 300 vertex descriptions.

Micro3D version 3's `renderPrimitives` method doesn't use triangle strips, and as a consequence you must specify every vertex of every triangle in the 3D object. The result is that attempting to use a JSR 184 `TriangleStripArray` as input into Micro3D version 3's `renderPrimitives` method means that too few vertices are being supplied. Conversely, feeding the buffer of triangles suitable for use by Micro3D version 3 into JSR 184's `render` method provides too many vertices.

Note that Micro3D version 3's `renderPrimitives` limits you to a maximum of 255 primitives. For a complex 3D model with hundreds of triangles, you'd first partition its vertex data into sections consisting of 255 vertices apiece, and then call `renderPrimitives` on each section.

4.6.2. Texture map differences

One item that jumps out at you when dealing with textures maps is that the Micro3D version 3 API uses 8-bit BMP images for its texture maps, while JSR 184 uses PNG images. A quick glance at the images themselves reveals yet another detail: Micro3D version 3 can manage a texture map with arbitrary dimensions (up to 256 pixels per side), while the JSR 184 implementation requires that the width and height the image be a power of two.

Another thing to watch out for is the order in which you specify the vertex texture coordinates. (This is the array `tex` in both the JSR 184 and Micro3D version 3 example MIDlets.) These coordinates are used to extract pixels from a section of the texture image for mapping onto a 3D object's surface. While this scheme provides great flexibility in controlling the texture's orientation onto a 3D model, such freedom can produce maddening results when the image spans two or more triangles. In the demo program, the square image spans the two triangles used to construct each cube face. Check carefully how the vertex texture coordinates are specified for each triangle-pair that represents a face of the cube. Better yet, try changing the order of these coordinates, rebuilding the MIDlet, and see what happens.

4.6.3. Keep immediate mode and retained mode operations straight

One of JSR 184's advantages is that it supports both an immediate mode and a retained mode, and that you can intermix their operations. However, be aware that retained mode *methods* have no effect on immediate mode *graphics* and vice-versa. The best way to explain this pitfall is by an example. Suppose you're using `render` on a 3D model composed of `TriangleStripArrays`, and you're using a `Light` to illuminate the scene. Changing the `Light`'s color and position should change how the model is lighted, right?

Wrong. The use of `render` on a mesh object (the triangle strip) is an immediate mode operation, while `Light` is normally a node object that's used in a retained mode operation. So, changing the `Light` has no effect on the model. Use the `Graphics3D` methods `setLight` or `addLight` to associate a `Light` with immediate mode rendering, so that it has an effect on the graphics. A `setCamera` method manages a `Camera` node the same way.

4.6.4. Don't mix LCDUI and 3D graphics

Once you attach the 3D rendering engine to a graphics object, say, a `Canvas`, do NOT execute any LCDUI graphics methods on the object until you complete the 3D graphics drawing and release it. If you happen to mix LCDUI graphics output with the 3D output, the results can be unpredictable. As developers know, mobile platforms are literal-minded devices and so creating an unpredictable result is going to generate a very predictable crash. Crashes, needless to say, are guaranteed to upset the user. Stated more concisely, the rule becomes: once `bind` (Micro3D version 3) or `bindTarget` (JSR 184) is invoked, only 3D APIs should be used for drawing until `release` (Micro3D version 3) or `releaseTarget` (JSR 184) is called to free the graphics object.

4.6.5. Design within limits

A developer writing a 3D application for a mobile phone must ensure that its code stays within the device's performance ceiling. Even a modest 3D scene can tax a mobile phone to the limit, particularly since the rendering is accomplished without the benefit of hardware graphics accelerators. The game play for a mobile 3D application must therefore be carefully designed not to overwhelm its internal computer.

With that in mind, we suggest that you start by benchmarking a few 3D-capable mobile phones before designing a game. Kishonti Informatics LP's `Jbenchmark` can rate a mobile phone's performance, and they have a 3D graphics benchmark in the works. Such performance numbers should help you understand the pros and cons of each mobile phone. While JSR 184 was designed to support a large range of mobile devices, be aware that the implementations can differ significantly. The differences aren't in the interfaces (the specification mandates that all

interfaces be supported), but rather that the performance of certain API features may be so poor that it isn't worth using them.

Once you start the game design, first develop high-polygon (detailed) artwork and models. Then lower the polygon count of the 3D models and textures. This reduces the complexity of the objects and therefore the number of computations required rendering them. The smaller screen on the mobile phone actually works to an advantage here: The fewer pixels the phone has to draw, the fewer computations required. Recall that each pixel's appearance is determined by dozens of computations that involve an object's visibility, color, shading, fog, and other 3D attributes. The smaller screen also tends to mask the fact that the 3D models are using fewer polygons. However, keep the high polygon models handy so that the application can scale with future devices that have hardware acceleration and/or larger screens.

Use multiple levels of detail in a game to reduce the processor's load. For example, nearby incoming asteroids in a shooter game would be rendered using 3D models, while those in the distance could be rendered faster using sprites.

Where you can, load balance the application's execution so that it reduces the context switches between the Java runtime and the native environment. As an example of this, if the API—such as JSR 184—supports a retained mode, have it render the 3D world all at once, rather than issuing numerous immediate mode commands. Put another way, generate the 3D scene using several large rendering tasks rather than with lots of small ones. The concept behind this rule is to reduce the context switches between the Java environment and the native rendering algorithms, which degrades performance. Note that this rule also works for those 3D APIs, such as Micro3D version 3, that don't have a retained mode.

One advantage with using JSR 184's retained mode is that it reduces the amount of code that the developer must write. Less code simplifies testing efforts and costs, plus there's the bonus of reducing the 3D application's memory footprint.

Reduce the depth of field in the 3D world to minimize the computations required. A fog effect is available in JSR 184, but its use is computationally expensive for software-only rendering engines.

Finally, where possible, know the mobile phone's hardware and work with it, as much as Java allows. Again, benchmark testing is essential to gain these insights. Here are some possibilities:

- **Understand the phone's video subsystem.** In particular, know how much VRAM the device has, and how you can cache images in it. For example, on some phones the order in which you access images can determine whether they become cached or not.
- **Use features unique to the phone.** You could flash the phone's backlight during say, a virtual thunderstorm. Or, activate the vibrator mechanism and backlight when an explosion occurs. You might use the phone's MIDP 2.0 network interaction features to enable socialization activities, or to support location-based games.

Of course, if you finely tune a 3D game for a particular mobile phone, it might not execute as quickly on a different phone. Since the responsiveness of the game is everything, you may have to support several versions of the game for different phones for it to be a commercial success.

5. Summary

As this paper shows, the capabilities of the mobile platform have evolved to the point to where it can support real-time 3D graphics. In addition, the platform's Java run-time environment, OTA download capability, and 3D APIs offers the means of mass-distributing 3D applications to a wide audience. A large audience allows both developers and network operators alike to make a profit at selling 3D applications, along with any follow-on programs or content.

Sony Ericsson provides two ways to leverage this nascent 3D application market. First, the company offers six 3D-capable mobile phones, the K500, K700, S700, and Z500 series, and the Vodafone-exclusive F500i and V800 mobile phones. These are excellent platforms for users to download and execute 3D content.

Second, through its J2ME SDK, Sony Ericsson provides developers with the means to write 3D applications, fully emulate them, and perform On-Device Debugging. Notably, both the SDK and the mobile phones offer two 3D API implementations: HI Corporation's Mascot Capsule Micro3D version 3, as well as JSR 184. The availability of Micro3D version 3 gives developers the ability to write and deploy 3D-capable applications now, and then migrate to the JSR 184 implementation once tools capable of generating content for it become widely available on the market.

6. Resources

6.1. Abbreviations

Acronym	Meaning
API	Application Programming Interface
CLDC	Connected Limited Device Configuration
IDE	Integrated Development Environment
J2ME	Java 2 Micro Edition
J2SE	Java 2 Standard Edition
JAD	Java Application Descriptor
JAR	Java Archive
JSR	Java Specification Request
MIDP	Mobile Information Device Profile
MMAPI	Mobile Media API (JSR 135)
SDK	Software Development Kit
SMS	Short Message Service
URL	Uniform Resource Locator
WAP	Wireless Application Protocol
WMA	Wireless Messaging APIs (JSR 120)

6.2. Further Information and Links

Item	Link
Sony Ericsson Developer World	http://www.sonyericsson.com/developer
J2ME Mobile 3D Graphics API for J2ME (JSR-184) specification	http://www.jcp.org/en/jsr/detail?id=184
J2ME MIDP 2.0 specification	http://www.jcp.org/en/jsr/detail?id=118

Sony Ericsson phone White Papers	www.sonyericsson.com/developer/site/global/docstools/phonespecs/p_phonespecs.jsp
Sony Ericsson J2ME SDK and J2ME Developers' Guidelines	www.sonyericsson.com/developer/site/global/docstools/java/p_java.jsp
Jbenchmark Web site	http://www.jbenchmark.com/
Sun One Studio 4 update 1 Mobile Edition	http://www.sun.com/software/sundev/previous/studio_me/index.html
JBuilder X Mobile Edition	http://www.borland.com/mobile/jbuilder/