High Performance Graphics

Graphics Performance Improvements in the Java™ 2 SDK, version 1.4

THE NETWORK IS THE COMPUTER™

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303 USA
650 960-1300 fax 650 969-9131

Part No.: 8xx-xxxx-xx
Revision 01, August 2001

Please
Recycle

High Performance Graphics

Introduction

Performance is always an issue for graphics programs. Rendering complex shapes and patterns on a display requires iterating through the pixels and calculating values for each of them. Loading an image and performing a filter operation on it sometimes requires allocating a large block of memory. When using these complex shapes, patterns, and images in an animated application, fast and reliable rendering can be difficult to acheive.

Sun Microsystems first released the 2D graphics framework, Java 2D with the Java 2 SDK, version 1.2. The graphics framework that Java 2D provides is much more powerful than the limited set of features offered by the JDK 1.1. For example, Java 2D includes support for creating arbitrary shapes, text, and images and provides a uniform mechanism for performing transformations, such as rotation and scaling, on these objects. However, these new features required more code to implement them and extra data to be stored for their operations, both of which caused additional overhead between the calls to the API and the rendering of pixels, all culminating in a performance degradation compared to the JDK 1.1. Another factor contributing to the performance degradation is that the hardware acceleration available in the JDK 1.1 was no longer accessible when using the SDK, version 1.2 because offscreen images were moved into software memory for the purpose of cross-platform consistency.

Now that Java 2D provides such a sophisticated graphics framework for the SDK, the Java 2D team has focussed on improving the performance of the features offered by this framework. Many changes made in the pipeline architecture have improved the performance of a wide range of Java 2D operations that have been habitually slow since the introduction of Java 2D. In addition to these general improvements, version 1.4 introduces the `VolatileImage` API, which allows you to take advantage of the hardware acceleration now accessible through the Java 2D API. The `VolatileImage` API allows you to create hardware-accelerated offscreen images, resulting in better performance of Swing and gaming applications in particular and faster offscreen rendering in general.

New Pipeline Architecture

For version 1.4 of the SDK, the Java 2D graphics pipeline architecture has been almost completely rewritten to solve many general performance problems. The key enhancements made in the new architecture are:

- less pipeline validation overhead
- faster copying
- better code sharing and code reduction

Before reading about the details of these three general enhancements, it's essential to understand a little of the background on the rendering process used in Java 2D, which the next section explains.

Background on Java 2D Graphics Pipelines

In the Java 2D API, a `Graphics2D` object encapsulates information about the current rendering state, such as the transform to use for converting from user space to device space and the color to use when rendering to the device. All of this state information is represented as a set of rendering attributes on the `Graphics2D` object. For example, when you set the color attribute of the `Graphics2D`, all of the text and shapes you render with this `Graphics2D` will be drawn in that color.

The Java 2D implementation uses different modules of code, called *pipelines*, when determining what to render to the device. Which pipeline Java 2D uses depends on what it is rendering and how it is rendering it. The pipeline that Java 2D uses to fill a shape with a complex paint is different from the one it uses to fill the same shape with a simple Color. When filling a shape with a complex paint, Java 2D must query the `Paint` object every time it needs to assign a color to a pixel whereas a simple color fill only requires iterating through the pixels and assigning the same color to all of them.

The Java 2D implementation has many pipelines for handling different graphics primitives and rendering attributes. The process of inspecting the rendering attributes, determining which pipeline to use, and deciding how to store the data necessary for rendering is called *pipeline validation*.

Less Pipeline Validation Overhead

For version 1.4, the new pipeline architecture reduces the performance overhead produced by pipeline validation with several implementation changes that:

- Improve the way that data is shared by the rendering pipelines.
- Reduce the amount of code executed when responding to changes in the rendering attributes.
- Use better heuristics to determine when a pipeline needs to be revalidated.

In the Java 2 SDK, version 1.2 and 1.3, common operations on a `Graphics` object often invalidated the rendering data cached for this `Graphics` object. This invalidation interrupted the rendering process by causing continuous re-creation of rendering information for the `Graphics` object. The reason for this invalidation is that the rendering primitive objects often contained both the rendering logic and the attribute data that the rendering instructions needed to modify the destination. For example, whenever `setColor` was called, the pipeline was invalidated and the rendering primitive object had to be re-created with the new color information.

Another problem with the old architecture is that the code for the inner loops that modified the pixels was often bundled up with the code to access the memory for the pixel data. Since pixels could be stored in Java arrays or in the VRAM of a framebuffer and both of these memory types needed different code to set up access to the pixels, the implementation would have to create multiple primitives to draw a line in a particular pixel format--one for each type of memory storage in which that pixel format could occur. In addition, the code to set up the memory access was quite large and was duplicated in every primitive designed to modify pixels stored in that type of memory.

The new pipeline architecture eliminated these problems by improving the way data is cached and code is shared. The attribute data associated with a `Graphics2D` object is now cached so that it can be shared between objects. In particular, color and clip information has been removed from rendering primitives and is stored in the `Graphics2D` object itself where all rendering primitives can access it. Now that rendering primitives can share data, those primitives that used to be instantiated with the data have been deleted.

In the new architecture, a common interface is used by the primitives to access the memory where the pixels are stored so that a single primitive can modify pixels of a given format regardless of what type of memory those pixels are stored in. The code to access the various types of memory is collected into a group of shared modules that these primitives invoke through a common interface. Now all primitives that access a certain type of memory can share the code that sets up access to that memory, and all objects that write to destinations with the same kind of pixel data can share code that modifies that kind of pixel data. This new ability to share code among the rendering primitives greatly reduces the code in the implementation and in turn reduces the runtime footprint.

In some cases, revalidation is necessary, such as when an opaque color is changed to a transparent color. However, the algorithm to render one opaque color is the same as rendering any other opaque color on a particular destination. So replacing one opaque color with another should not invalidate a pipeline. The implementation now only invalidates the pipeline when an attribute is changed to a different type of value, rather than when an attribute is changed to a different value of the same type.

Benefits of New Pipeline

The changes in the new pipeline architecture have improved all areas of Java 2D and other APIs that use Java 2D, not only in terms of performance but also in terms of rendering quality and reliability.

More Reliable Rendering

The more intelligent data caching and code reuse available in the new pipeline architecture have resulted in more reliable rendering in many areas, including:

- Rendering text that has attributes
- Drawing lines with large scale factors
- Performing `copyArea` operations

Rendering Text That Has Attributes

The Java 2D font engine produces either a bitmap or an outline path from a glyph. The font engine can cache a bitmap in the bitmap glyph cache but cannot cache an outline path. If the text is small, copying glyphs from the cache to the destination is a lot faster than rendering the glyphs using an outline path, and the quality of the rendering is also better. When the glyphs are large, the quality and the rendering performance gains are less noticeable when using a cached bitmap. In the old architecture, the implementation rendered solid text from the glyph cache bitmap unless the text had rendering attributes, in which case it used the outline path pipeline. Not only was this process slow, but it also produced inconsistent results. For example, if the opaque color was changed to a gradient, the implementation had to switch from the bitmap pipeline to the outline path pipeline to render the text. If this pipeline switch occurred while a program was running, the text appeared to jiggle because of the sudden change in text quality.

In the new architecture, a more predictable heuristic is used to determine when the bitmap pipeline or the outline path pipeline should be used. If the text is smaller than a certain size, the implementation renders it from the glyph cache bitmap regardless of what rendering attributes the text is rendered with. If the text is as large or larger than the threshold size, the implementation uses the outline path pipeline, which produces about the same performance as the bitmap pipeline because larger text would require more memory allocation in the cache without much of an improvement in quality.

Drawing Lines With Large Scale Factors

Drawing lines with large scale factors produced integer overflows when the internal code to render the lines tried to represent the lines as integers. If these numbers fell outside the 16-bit range, the code would go into an infinite loop trying to clip the lines. This problem has been resolved, and now Java 2D can handle arbitrary scale factors.

Performing `copyArea` Operations

Many copying routines have been improved by eliminating the use of an intermediate buffer. In particular, the `copyArea` operation, which involved an overlapping source and destination, such as used in Swing scrolling components, can often be performed in place rather than relying on an intermediate buffer. Avoiding an intermediate buffer means less memory allocation and garbage collection is needed.

The new image-scaling operations also avoid extra copying by bypassing an intermediate buffer. Now DirectDraw scale operations on Win32 and software scale operations on Solaris and Linux are able to blit the scaled image directly to the destination. Note that hardware-accelerated scaling is currently disabled on Win32 because of quality problems, but you can enable it with a *runtime flag*.

Image Scaling Performance Improvements

Rendering a scaled image using `drawImage` with a scale other than 1:1 was often much slower after JDK 1.1. The `drawImage` implementation created a scaled image in an intermediate buffer and then copied the buffer to the destination. The memory allocation required for the intermediate buffer and the extra copies to and from the buffer were two factors contributing to the performance degradation. Another reason for the performance loss was that any accelerated image-scaling operations available on the display hardware could not access the memory that stored the image data.

The new architecture has a software-scaling operation that scales an image directly into the destination in a single copy operation. This new scaling operation benefits all platforms. If you are rendering a scaled image in the Win32 environment, you can take advantage of the access to DirectDraw surfaces that the new architecture provides if both the source and destination of the scaling operation are in DirectDraw surfaces. The DirectDraw scaling operation blits the scaled image directly to VRAM. Note that the hardware-accelerated image scaling on Win32 is currently disabled because of quality and consistency problems, but you can enable it with the runtime flag, `sun.java2d.ddscale=true`. See the section *Flags for Controlling Performance and Quality* for more information. If hardware-accelerated scaling is not available to you, you still benefit from the new software-scaling operation. The new software-scaling operations and the access to hardware-accelerated scaling obviate the need for extra memory allocation and additional copy operations.

The performance of scaling to and from an accelerated image improved significantly between the Beta 2 and Beta 3 releases because of a new heuristic that, based on the frequency of read operations performed on an image, can store the image to the type of memory that will allow the best performance. See the next section for more information on the new heuristic.

Accelerated-Image Reading Performance Improvements

Translucent and scaling operations to and from an accelerated image were very slow in the Beta 2 release of the SDK 1.4. This performance degradation was a result of frequently reading the image from VRAM, which is a slow process compared with reading an image from system memory. Translucent operations require frequent reads because alpha compositing performs read-modify-write operations on the destination. Scaling from an accelerated image requires reading from the source or destination image in order to perform the operation successfully. The Java 2D implementation available in the Beta 3 release of the SDK 1.4 can transfer the surface to system memory if the image is experiencing frequent reads. If reads occur less frequently, Java 2D can transfer the surface back to VRAM. If you

are working in a UNIX environment, you can override this heuristic by using the J2D_PIXMAPS environment flag. See *Environment Flag for Solaris and Linux* for more information.

Remote X Server Performance Improvements

When working with graphics in a UNIX or Linux environment, you have the option of performing your graphics computations on a remote client by running an X Server from your machine. However, if you used offscreen images with Java 2D while running the X Server you would experience poor performance because the offscreen image was created on the client side. Every time you needed to re-render the image to the destination, the image had to be copied from the remote X client to the display. If you were using the offscreen image for double buffering, you would have to endure a network copy of the image whenever the screen was repainted.

Because accelerated offscreen images are available in 1.4, the offscreen image is now created on the server side, local to the screen. Java 2D uses X protocol requests, which tell the X server what and how to render to the offscreen image located on the server side of the network. Only X protocol requests are sent over the network; the image itself stays on the server side. This particular improvement also has positive implications for Swing performance because Swing uses double-buffering. The Swing application does not have to wait for the backbuffer to be copied over the network for every screen refresh.

One drawback with both remote X and DirectDraw is that neither antialiasing nor alpha-blending can be accelerated. In fact, antialiasing and alpha blending operations on remote X are usually much slower compared to version 1.3 because the image must be copied to the X client to perform one of these operations and then the new image must be copied back to the server. The Java 2D team is looking into solutions to this problem for future releases. Since most Swing applications do not use alpha blending or antialiasing anyway, this problem should not cause serious concern for developers using Swing.

Local X Server Performance Improvements

In J2SE 1.4 releases prior to Beta 3, `drawImage` performance when rendering to the screen in some cases was worse than `drawImage` performance in J2SE 1.3. The reason for the performance degradation was that Java 2D was not using the Shared Memory Extension on Solaris and Linux in the local display environment. The Shared Memory Extension allows an X server and X client running on the same machine to jointly access shared memory, which enables faster data transfers.

For Beta 3, Shared Memory Extension is used on Solaris and Linux in the local display environment, resulting in better performance when rendering to the screen and to the images. When DGA is not available, toolkit images and images created with `Component.createImage` or `GraphicsConfiguration.createCompatibleImage` are

stored in pixmaps instead of system memory, enabling faster copies to the screen using X protocol requests. You can override this behavior with the pmoffscreen runtime flag, described in the section *Runtime Flag For Solaris and Linux*.

Pixmaps that are not in shared memory might be stored in VRAM, which allows for fast copies to the screen, but reading from these images is very slow, as described in the section *Accelerated-Image Reading Performance Improvements*. Since Shared Memory Extension is now accessible in the local display environment, images that experience frequent reads can be stored in Shared Memory Pixmaps, which always reside in system memory. Java 2D can detect if an image is read from more frequently or copied from more frequently and can transfer the image to the appropriate memory. You can control which memory is used by setting the `J2D_PIXMAPS` environment variable described in section *Environment Flag for Solaris and Linux*.

Swing API Benefits

When using Swing in the SDK 1.4, you should notice better performance in many cases because Swing uses many of the Java 2D operations that were targeted for performance improvements for version 1.4. Because the rendering of Swing hierarchies relies heavily on common operations like `setColor` and `Graphics.create`, the invalidation and re-creation of rendering data caused poor repaint performance for many Swing applications. For this reason, Swing is a primary beneficiary of the reduced overhead resulting from the graphics pipeline improvements. Now that image data, such as color, is stored in a central, shared location, the many calls to `setColor` that Swing performs do not invalidate the pipeline, and thus do not result in worse performance for Swing applications. The process of creating and disposing of `Graphics` objects is also faster as a result of the general pipeline improvements. Since each subcomponent of a `JComponent` draws itself with a new `Graphics` object, a particular Swing application could create and dispose of many `Graphics` objects. The create and dispose operations now are much faster and produce less memory overhead.

Some Swing components allow you to scroll over a piece of data, such as an image or text. Often the scrolling destination overlaps the source, and in this case a copy is performed. Prior to version 1.4, the information would be copied to an intermediate buffer and then copied to the screen. These copies now occur in place, thereby avoiding the extra memory allocation and garbage collection associated with an intermediate buffer.

The new access to hardware acceleration for offscreen images has also dramatically improved Swing performance. Since Swing uses double buffering by default, Swing applications will generally perform better because of offscreen image acceleration. Swing performance on remote X is also improved because the backbuffer that Swing uses for double buffering is stored on the server side, thereby avoiding a network copy of the backbuffer every time the screen is repainted. For more information on the remote X improvements see the *Remote X Server Performance Improvements* section. See the next section for more information on hardware acceleration for offscreen images.

Hardware Acceleration for Offscreen Images

Java 2D provides access to hardware acceleration for offscreen images, resulting in better performance of rendering to and copying from these images. On win32 hardware acceleration is achieved through DirectDraw, an API designed for giving applications direct access to video hardware memory and acceleration. On Linux and Solaris, offscreen images might be stored in a pixmap, which is an area of memory that is stored on the X server side of the network.

In the case of DirectDraw, any image stored in accelerated memory could be "lost" at any time: certain events can cause the Windows operating system to clear the image out of VRAM, thus destroying the contents of the image.

A `VolatileImage` represents an offscreen image whose content can be lost at any time. However, a `VolatileImage` offers performance benefits over other kinds of images because a `VolatileImage` allows an application to take advantage of hardware acceleration, thus enabling the application to run at the native speed of the underlying platform.

The new hardware acceleration has improved the performance of Swing applications because Swing double-buffering now uses `VolatileImage`. For more information on hardware acceleration, see the VolatileImage API User's Guide.

Tips for Achieving Better Performance

Although the new performance improvements eliminate a lot of problems, they still don't optimize everything. In particular, alpha blending and anti aliasing will adversely affect performance because calculating alpha values for each pixel takes more work than rendering opaque values. In addition, Java 2D cannot hardware accelerate images with alpha at this time. Only opaque images or images with 1-bit transparency can be hardware accelerated. Acceleration support for 1-bit transparency enables you to accelerate sprites with transparent pixels. You can use 1-bit transparency to make the background color of a sprite rectangle transparent so that the character rendered in the sprite appears to move through the landscape of your game, rather than within the box represented by the sprite rectangle's background color.

When creating an image whose data is copied to the screen, you should create the image with the same depth and type of the screen. This way, no pixel format conversion between the image and the screen is necessary before copying the image. To create such an image, use either `Component.createImage` or `GraphicsConfiguration.createCompatibleImage`, or use a `BufferedImage` created with the `ColorModel` of the screen. The pre-defined types of `BufferedImage` have their own color models.

Rectangular fills--including horizontal and vertical lines--tend to perform better than arbitrary or non-rectangular shapes whether they are rendered in software or with hardware acceleration. If your application must repeatedly render non-rectangular shapes, consider drawing the shapes into 1-bit transparency images and copying the images as needed.

If you are still experiencing low frame rates, try commenting out pieces of your code to find the particular operations that are causing problems, and use the tips in this section to replace these problem operations with something that might perform better.

Another way to identify performance problems in your application is to run it with the new trace flag introduced in Beta 3 of the SDK, version 1.4. See the section *Performance-Tuning Flag For All Platforms* for more information on the trace flag.

If you still are experiencing performance problems, submit a bug with your test case.

Flags for Controlling Performance and Quality

In any graphics program, performance and quality can sometimes seem mutually exclusive. Most of the changes that the Java 2D team has made in the pipeline architecture have greatly improved performance without compromising quality, but in some situations you might want to decide whether you prefer high quality over high performance. You might also be working in an environment that does not allow your application to derive a performance benefit from a feature designed to improve performance in other environments. This list of flags allows you to disable or enable certain features so that you can obtain the best possible performance and rendering quality available in your environment.

Environment Flags

These environment flags are for use on a UNIX environment. You set these flags by using the setenv command. For example, to turn off DGA support and hardware acceleration, enter the following at the command line:

```
setenv NO_J2D_DGA
```

Environment Flag for Solaris and Linux

With the release of Beta 3 of the SDK, version 1.4, Java 2D uses Shared Memory Pixmaps in the local display environment for storing images that experience frequent reads. You can override this default behavior by using the J2D_PIXMAPS environment flag:

```
J2D_PIXMAPS=shared/server
```

If you set this flag to “shared”, all images will be stored in Shared Memory Pixmaps if you are working in a local display environment. Conversely, if you set this flag to “server”, all images will be stored in normal pixmaps, not Shared Memory Pixmaps. The normal pixmaps can be stored in VRAM at the discretion of the device driver.

Environment Flags for Solaris Sparc

- NO_J2D_DGA
  Set this environment flag to turn off DGA support and hardware acceleration, which sometimes helps to reduce rendering artifacts on Solaris Sparc.
- USE_DGA_PIXMAPS
  Set this environment flag to turn on DGA acceleration for pixmaps. The use of DGA with pixmaps is disabled by default because it was causing system freezes on some video boards, such as PGX32.

Runtime Flags

You set these flags when you are running your application. For example, to turn off DirectDraw, run your application like this:

```
java -Dsun.java2d.noddraw=true <yourapp>
```

Performance-Tuning Flag For All Platforms

If your application is experiencing less-than-desirable performance, the new trace runtime flag can help you determine the source of the problem. The flag is specified with a list of options:

```
-Dsun.java2d.trace=<optionname>,<optionname>,...
```

The options are:

log	print out the name of each primitive as it is executed
timestamp	precede each log entry with the `currentTimeMillis()`
count	at exit, print out a count of each primitive used
out:<filename>	send output (logging and counts) to the indicated file
help	print out a short usage statement
verbose	print out a summary of the options chosen for this run

If you use the log option, the Java runtime will print the executed primitives' names, most of which will be in this format:

```
<classname>.<methodname>(<src>,<composite>,<dst>)
```

The methodname represents the basic graphics operation that is used to do the actual rendering work of a Graphics method invocation. These method names will not necessarily map directly to methods on the Graphics object, nor will the number of calls made on the Graphics object map directly to the number of primitive operations performed.

The src and dst represent the type of surfaces or source data involved in the operation.

The composite names match the names in the AlphaComposite class fairly closely with the suffix "NoEa" meaning that the AlphaComposite instance had an "extra alpha" attribute of 1.0. The "SrcNoEa" type is the most commonly used composite type and represents the simplest way of transferring pixels with no blending required. "SrcNoEa" is often used behind the scenes even though the default composite is "SrcOver" when opaque colors and images are rendered because the two operations are indistinguishable for opaque source pixels.

Platform rendering pipelines are sometimes used for doing opaque operations on surfaces accessible by a platform renderer, such as X11, GDI, or DirectDraw. Their names currently use a simplified naming format, which has a prefix for the platform renderer and the name of the operation but without any classname or operand type list. Examples are "X11DrawLine", "GDIFillOval", and "DXFillRect". In the future these names should more closely approximate the names of the other primitives.

Runtime Flag For Solaris and Linux

Starting with the Beta 3 release of the SDK, version 1.4, Java 2D stores images in pixmaps by default when DGA is not available, whether you are working in a local or remote display environment. You can override this behavior with the pmoffscreen flag:

```
-Dsun.java2d.pmoffscreen=true/false
```

If you set this flag to true, offscreen pixmap support is enabled even if DGA is available. If you set this flag to false, offscreen pixmap support is disabled. Disabling offscreen pixmap support can solve some rendering problems.

Runtime Flags For Win32

- -Dsun.java2d.noddraw=true
  Setting this flag to true turns off DirectDraw usage, which sometimes helps to get rid of a lot of rendering problems on Win32.
- -Dsun.java2d.ddoffscreen=false
  Setting this flag to false turns off DirectDraw offscreen surfaces acceleration by forcing all createVolatileImage calls to become createImage calls, and disables hidden acceleration performed on surfaces created with createImage.
- -Dsun.java2d.ddscale=true
  Setting this flag to true enables hardware-accelerated scaling. Hardware scaling is disabled by default to avoid rendering artifacts in existing applications. These rendering artifacts are caused by possible inconsistencies between the scale method that the software

scaling operation uses (nearest neighbor) and the different scale methods that video cards use. Certain events that occur while an application is running might cause a scaled image to be rendered partially with hardware scaling operations and partially with software scaling operations, resulting in an inconsistent appearance. For now, you can enable acceleration by setting the `ddscale` flag to `true`.

Examples Demonstrating Performance Gains

Since the graphics performance gains you experience will depend partially on the hardware and operating system environment that you have, we have provided, in place of charts and data, this set of examples that you can run to see the performance improvements for yourself. Try running them with both version 1.3 and 1.4 of the Java 2 SDK.

CopyAreaPerf

This example scrolls the screen a certain number pixels at a time, demonstrating how the reduction in memory allocation and garbage collection can really speed up your copying operations. When you run the example, specify how many pixels you want the application to scroll at a time.

FadeTextSplash

The FadeTextSplash example shows fading text using alpha composites. If you run it with version 1.4, you will see the bitmap glyph cache in action. Notice that the text doesn't jiggle and you will not see any metrics changes when you run the application with version 1.4. When you run the example, you can specify what text you want the application to display.

PhotoMesa

This application is a digital image library browser and demonstrates the image-scaling improvements in Java 2D by allowing you to zoom in on an image in the library. The application was developed by Dr. Ben Bederson, assistant professor of computer science, and director of the Human-Computer Interaction Lab at the University of Maryland. When you run this example, make sure you are using the Beta 2 or later release of the Java 2 SDK, version 1.4.

See the `VolatileImage` API User's Guide for examples demonstrating the `VolatileImage` API.

THE NETWORK IS THE COMPUTER™

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
650 960-1300

For U.S. Sales Office locations, call:
800 821-4643
In California:
800 821-4642

Australia: (02) 413 2666
Belgium: 32 2 716 7911
Canada: 416 477-6745
Finland: 358-0-502 27 00
France: (1) 30 67 50 00
Germany: (0) 89-46 00 8-0
Hong Kong: 852 802 4188
Italy: 039 60551
Japan: (03) 221-7021
Korea: 822-563-8700
Latin America: 415 688-9464
The Netherlands: 033 501234
New Zealand: (04) 499 2344
Nordic Countries: +46 (0) 8 623 90 00
PRC: 861-831-5568
Singapore: 224 3388
Spain: (91) 5551648
Switzerland: (1) 825 71 11
Taiwan: 2-514-0567
UK: 44 (0)1252 420000

Elsewhere in the world,
call Corporate Headquarters:
650 960-1300
Intercontinental Sales: 650 688-9000