Getting Started with Mobile 2D Graphics for J2ME

By Michael Powers, August 2005

Download the source code of the M2GDemo MIDlet, the origin of the code samples in this article.

Introduction

This article introduces the Scalable 2D Vector Graphics API, an optional package you can use with Java 2 Platform, Micro Edition (J2ME) profiles. This friendly API for rendering and transforming compact, scalable two-dimensional (2D) graphics is specified as JSR 226 in the Java Community Process (JCP). I'll present an overview of the API, highlight typical use cases, and demonstrate those use cases in code.

Overview

JSR 226 brings rendering and playback of vector-based 2D graphics to Java-powered mobile applications. The expert group, headed by Nokia, comprises the major players in the mobile industry, all of whom have an interest in promoting vector-based multimedia content on their devices. At this writing, the specification is recently approved, and a reference implementation eagerly anticipated.

Vector graphics have two big advantages for developers of mobile applications: they're both compact and scalable. Consider this simple image:

![Hello brave world.](image)

Figure 1: A Simple Sample Image

In GIF format, this image uses up 7,386 bytes. In a vector format, the size is only 693 bytes, less than a tenth as large. How is this reduction possible?

Raster-based image formats like GIF encode the color contents of each pixel of the rectangular region that comprises the image. Vector-based images instead contain only the drawing instructions that determine how the pixels should be colored. The vector representation of an image can be much more compact, a big plus in resource-constrained mobile devices.

Scalability is another important advantage. Vector images can be transformed clearly because their drawing instructions are resolution-independent. Figure 2 shows the previous image scaled down, flipped, and rotated:
The units of measurement in a vector image are arbitrary and relative, and they can be multiplied or divided by some constant factor to transform the image to fit the device screen. Applying simple mathematical operations to the coordinates, you can create virtually limitless effects, including flipping, rotation, stretching, and skewing. Using vector graphics, application developers no longer need to create separate sets of images and icons to support devices of varying screen resolution.

The compactness and scalability of vector graphics really shine in the area of animation. Raster-based animations, like animated GIFs in web pages or animated sprites in MIDP games, laboriously include the entire image in each frame of the animation. By contrast, vector-based animations include only the instructions that indicate which elements of the image should be changed and when to change them. If vector images are an order of magnitude smaller than their raster counterparts, then vector animations are two orders smaller than raster animations, making them ideal for transmitting and presenting multimedia content on a mobile device.

While vector graphics and animations based on the proprietary Macromedia Flash player have become popular on the web, the Scalable Vector Graphics (SVG) file format has emerged as the open-standard and patent-free choice for the rest of the industry. It’s standardized by the World Wide Web Consortium (W3C). SVG-Tiny offers a subset of the full capabilities of SVG, geared toward implementations on mobile devices. JSR 226 adopts version 1.1 of SVG-Tiny as the official file format for J2ME vector graphics. W3C calls SVG-Tiny a “profile”, but I’ll use “format” to avoid confusion with J2ME profiles.

The SVG-Tiny standard also enables implementations to support animation. Instructions embedded in the image file itself can modify the positions and attributes of the image elements in response to timer events and user input.

For more complex interactivity, JSR 226 adopts an API that is compatible with microDOM, a subset of SVG 1.2’s full Document Object Model (DOM). Java-based programs can use this API to create and modify vector graphics and animations on the fly.

SVG is an eXtensible Markup Language (XML) format, well-documented and codified with a public schema. SVG-Tiny files may be small but they’re still SVG files, and they contain XML, as you’d expect. Here’s the full SVG-Tiny representation of the first “brave world” image:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "/W3C/DTD SVG 1.1 Tiny/EN" 
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11-tiny.dtd">
<svg preserveAspectRatio="xMidYMid meet"
    viewBox="10 10 130 55" width="150.0" height="75.0">
    <rect x="25" y="25" transform="translate(75 36.375)
        rotate(15) translate(-70 -35.375)"
        fill="#00007E" width="100" height="22.75" stroke="#000000" stroke-width=".5"/>
    <rect x="25" y="25" fill="#FF9800" width="108.5"
        height="22.75" stroke="#000000" stroke-width=".5"/>
    <text x="30" y="40" fill="#00007E" stroke="#FFFFFF"
        stroke-width=".33" xml:space="preserve">Hello brave world.</text>
</svg>
```

This document declares two rectangles and some text. The first rectangle is rotated 15 degrees around its center by the series of transformations contained in the corresponding rect tag’s transform attribute. The second rectangle is drawn on top of the first, and the text is drawn on top of the second rectangle.

Even though raw XML text is not a particularly compact format, this document is still much smaller than even a compressed raster representation, and human-readable to boot. Because XML text compresses well, SVG documents take up relatively little of the space in an application’s JAR file. JSR 226 requires implementations to support SVG documents compressed with gzip. The extension .svgz denotes a gzip-compressed
The Mobile 2D API

A primary objective of JSR 226 is to define the Mobile 2D Graphics (M2G) API, a small set of classes that align closely with the capabilities of the SVG-Tiny format. It’s not intended to be a generalized 2D drawing toolkit like the one J2SE’s Graphics2D class provides. The M2G API is plainly focused on the playback and runtime manipulation of SVG content.

M2G consists of high-level classes for creating and rendering vector images and low-level classes for manipulating the XML components of a vector image as parts of a Document Object Model (DOM) tree. Vector images are instances of ScalableImage; you obtain them from the static createImage() method of that class. You use an instance of ScalableGraphics to draw ScalableImages onto a MIDP Graphics context. SVGImage is a subclass of ScalableImage that provides hooks for event handling and access to the underlying DOM document.

To make the common case of rendering animated SVG content more convenient, the API provides the SVGAnimator class. For MIDP applications, SVGAnimator creates and controls a Canvas object that automatically handles screen updates in response to animation events and programmatic modifications to the image. It provides a Player-like interface for controlling the playback of the animation.

These classes are defined in the javax.microedition.m2g and org.w3c.dom.svg packages, summarized in Table 1, and documented fully in the JSR 226 specification. The SVG DOM classes extend the base DOM classes and interfaces defined in the standard org.w3c.dom and org.w3c.dom.events packages.

### Table 1: Class or Interface Descriptions

<table>
<thead>
<tr>
<th>Class or Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.microedition.m2g</td>
<td></td>
</tr>
<tr>
<td>ScalableGraphics</td>
<td>This is the fundamental class for 2D rendering.</td>
</tr>
<tr>
<td>SVGAnimator</td>
<td>This class handles automatic rendering of updates and animations in an SVGImage to a target user interface component.</td>
</tr>
<tr>
<td>SVGEventListener</td>
<td>This interface is used to forward platform-specific events to an application.</td>
</tr>
<tr>
<td>SVGImage</td>
<td>This class represents an SVG image conforming to the W3C SVG Tiny 1.1 Profile.</td>
</tr>
<tr>
<td>ExternalResourceHandler</td>
<td>This interface is used to load synchronously any external resources needed for loading SVG content.</td>
</tr>
<tr>
<td>org.w3c.dom.svg</td>
<td></td>
</tr>
<tr>
<td>SVGAnimationElement</td>
<td>This interface represents an animation element, and includes methods to control the timing of animations.</td>
</tr>
<tr>
<td>SVGElement</td>
<td>This interface represents an SVG element in the document tree.</td>
</tr>
<tr>
<td>SVGLocatableElement</td>
<td>This interface represents a drawable SVG element, typically a shape, image, or text.</td>
</tr>
<tr>
<td>SVGMatrix</td>
<td>This interface represents an &quot;SVG matrix&quot; data type, identified by an affine transformation &amp;equivalent to a linear transformation followed by a translation.</td>
</tr>
<tr>
<td>SVGPath</td>
<td>This interface represents an &quot;SVG path&quot; data type used to define the path geometry.</td>
</tr>
<tr>
<td>SVGPoint</td>
<td>This interface represents an &quot;SVG point&quot; datatype, identified by its x and y components.</td>
</tr>
<tr>
<td>SVGRect</td>
<td>This interface represents an &quot;SVG rectangle&quot; datatype, consisting of a minimum X, minimum Y, width, and height values.</td>
</tr>
<tr>
<td>SVGRGBColor</td>
<td>This interface represents an &quot;SVG RGB color&quot; data type, composed of red, green, and blue components.</td>
</tr>
<tr>
<td>SVGSVGElement</td>
<td>This interface represents an element in an SVG document tree.</td>
</tr>
</tbody>
</table>

Creating M2G Applications

To create an M2G application, you need a working implementation of JSR 226. At this writing the specification is only recently finalized; no devices implement it and no official reference implementation is available. You can download a limited, experimental implementation as an add-on package for the mpowerplayer developer toolkit.

You also need some SVG-Tiny content. This article uses the "brave world" image above, but more SVG-Tiny examples are available from mobility development toolmaker TinyLine. You can also create your own content...
with a commercial Java-based application called Sketsa, which works like an illustration tool and uses SVG as its file format.

There are a few approaches to get your SVG content onto the screen. The simplest is to create an SVGAnimator for your image and put its Canvas on screen. Another approach is to create a ScalableImage instance and use an instance of ScalableGraphics to draw it onto the Graphics context of your own Canvas or CustomItem. You'll have more control over the presentation, but you'll be responsible for handling input events and repaints as a result of animation or user interaction.

**Using an SVG Animator to Display 2D Graphics**

You can obtain an instance of SVGAnimator from its static createAnimator() method, providing your own SVGImage. Using SVGAnimator is a good choice because the animator handles all user interaction and animation for dynamic content. The play(), pause(), stop(), and setTimeIncrement() methods give you programmatic control over the playback of an animation. The implementation may provide a native user interface for directing any dynamic behavior as well.

Once you obtain the SVGAnimator, embedding it in your application is easy. Calling getTargetComponent() returns an instance of Canvas that can be placed on screen. Here's an example:

```java
public void animationTest() {
    // create an animator to load the content
    SVGAnimator animator = SVGAnimator.createAnimator( image );

    // add our custom event listener
    animator.setEventListener( new CustomEventListener( animator ) );

    // get the Canvas for this player; requires a cast
    Canvas canvas = (Canvas) animator.getTargetComponent();

    // add a "back" command
    canvas.addCommand( backCommand );
    canvas.setCommandListener( this );

    // show it
    Display.getDisplay(this).setCurrent( canvas );

    // start it
    animator.play();
}
```

In a MIDP environment, the return value from getTargetComponent() is a MIDP Canvas. In a non-MIDP environment, the result would be appropriate to the native windowing system; for example, a Component in an AWT environment. If the environment has more than one suitable choice, provide the name of the desired class as a second argument to createAnimator().

Figure 3 shows what the form looks like when displayed. Depending on the implementation, and on the availability of a stylus or other pointing device, the animator may allow the user to interact with the content. There is no dynamic content in the "brave world" sample, but note that the text is selectable.
Interacting With the Animator

Although the "brave world" image is not an animation, a little programming can add interactivity.

JSR 226's DOM support allows a program to alter the structure of an image. Just like any other XML document, an SVGImage is represented as a DOM object tree. The M2G API lets you use familiar XML manipulation techniques to modify the image. Each node in the tree is an SVGElemet and provides methods for navigating to the adjoining parent and child nodes. You modify nodes by adding or removing child nodes, or by casting to an appropriate element class and calling the methods of that class to make changes.

The SVGEventListener interface enables your application to receive user input events from the SVGAnimator's Canvas. This example uses the DOM API to scale and rotate the image in response to input from the user's arrow keys:

```java
private void rotate( final float delta )
{
    // put ourselves on the animator's thread
    animator.invokeLater( new Runnable()
    {
        public void run()
        {
            // execute the transformation
            Document document = svgImage.getDocument();
            SVGSVGElement root = (SVGSVGElement)...
        }
    });
}
```

```java
{        int gameAction = canvas.getGameAction( keyCode );
        switch ( gameAction )
        {
            case Canvas.UP:
                scale( 0.10f );
                break;
            case Canvas.LEFT:
                rotate( -10.0f );
                break;
            case Canvas.DOWN:
                scale( -0.10f );
                break;
            case Canvas.RIGHT:
                rotate( 10 );
                break;
            default:
                // leave unchanged
                }
        }
```
private void scale(final float delta) {
    // put ourselves on the animator's thread
    animator.invokeLater(new Runnable() {
        public void run() {
            // execute the transformation
            Document document = svgImage.getDocument();
            SVGSVGElement root = (SVGSVGElement) document.getDocumentElement();
            root.setCurrentScale(root.getCurrentScale() + delta);
        }
    });
}

SVGImage.getDocument() provides a reference to the document that the animator is rendering. Because the player may be running an animation or otherwise updating dynamic content, you must take care to synchronize your changes with the animation thread, or unspecified bad things may happen. SVGAnimator's invokeAndWait() and invokeLater() methods behave just like their AWT counterparts to ensure your code is executed on the animation thread.

Rendering 2D Content Directly

If you want absolute control over where and how your SVG image is rendered, you can use the ScalableGraphics class to draw a ScalableImage directly onto the Graphics object passed into the paint() method of a Canvas, Layer, or CustomItem instance.

You create an instance of ScalableGraphics by invoking its static createInstance() method. Before drawing, you must first use the bindTarget() method to bind the scalable graphics instance to the graphics object that was passed into the paint method. Take care to call releaseTarget() when you're done. Once the binding is established, you draw each of your images using the render() method.

This example draws the "brave world" image repeatedly at different sizes by calling the setViewportHeight() and setViewportWidth() methods before each call to render().

private static class M2GCanvas extends Canvas {
    // retain a reference the specified image
    ScalableImage scalableImage;
    // retain an instance of a scalable graphics
    ScalableGraphics scalableGraphics;

    public M2GCanvas(ScalableImage inImage) {
        scalableImage = inImage;
        // create the scalable graphics instance
        scalableGraphics = ScalableGraphics.createInstance();
    }

    public void paint(Graphics g) {
        // clear the display
        g.setColor(255, 255, 255);
        g.fillRect(0, 0, getWidth(), getHeight());

        // bind our scalable graphics to the given graphics
        scalableGraphics.bindTarget(g);

        // render at fixed position and size
        scalableImage.setViewportWidth(50);
        scalableImage.setViewportHeight(75);
        scalableGraphics.render(5, 50, scalableImage);
    }
}
This simple subclass of Canvas displays an image three times, at different positions and sizes. The third time, the image fills the size of the canvas. On devices that support resizable screens, the image scales appropriately. Figure 4 shows the results when the canvas is displayed. In the implementation provided, the larger image scales as the window is resized.

If the images you're rendering are dynamic, you should allow the user to interact with them. When you're rendering without an SVGAnimator, you need to detect the user's pointer events, discover whether and where the image was clicked, and call dispatchMouseEvent() on the SVGImage to trigger any scripted behavior. For a CustomItem, form traversal should allow activation and focus navigation through the elements within an image, using SVGImage's activate() and focusOn() methods respectively.

Not only can your program manipulate the DOM of an existing SVGImage, it can also build an image from scratch. The static createEmptyImage() method of SVGImage returns an image with an empty document skeleton ready to be populated with programmatically created SVGElements representing shapes and transformations. This technique can be used in conjunction with ScalableGraphics to effect a generalized 2D drawing toolkit, but as I noted earlier such usage wasn't intended, and it's not recommended.

Conclusion

The Scalable 2D Vector Graphics API for J2ME provides capabilities for rendering and manipulation of vector-based images and animation. The compactness and scalability of vector images make them a natural fit for mobile applications. JSR 226's Mobile 2D Graphics API enables applications to create, display, and modify SVG content using the MIDP and AWT toolkits.

About the Author

Michael Powers is Chief Technology Officer of mpowerplayer inc., a provider of software solutions for mobile application vendors and developers. He has been working with Java technology in its various incarnations since its inception. The mpowerplayer MIDP emulator is a free download.