



Java and the Digital Image Processing Application Package

Mariusz Jankowski

Department of Electrical Engineering
University of Southern Maine
mjkcc@usm.maine.edu

Introduction

The *Digital Image Processing* application package, a relatively recent addition to the *Mathematica* application library, enhances *Mathematica* with over 160 functions in the area of digital processing of images (see [1]). The introduction of the package supported by significant performance improvements in versions 4.0 and 4.1 have made *Mathematica* an effective platform for image processing and analysis. Now, thanks to the introduction of *J/Link* (see [2]), the *Mathematica* user gains direct access to an extensive native Java imaging functionality found in the Java 2D and Java Advanced Imaging (JAI) application programming interfaces (APIs) and excellent graphical user interface (GUI) design features thanks to Java Swing.

These developments have the potential to profoundly impact the future of desktop image processing in general and within the *Mathematica* environment in particular. Issues that will be addressed in the presentation include Java imaging functionality, *Mathematica* and Java image data formats, performance and trade-offs. Examples will be shown of translation between *Mathematica* and Java image data objects and scripting of Java image operators found in Java 2D and JAI.

Java 2D API

The Java 2D API introduced in JDK 1.2 provides enhanced two-dimensional graphics, text, and imaging capabilities for Java programs through extensions to the Abstract Windowing Toolkit (AWT) (see [3]). This comprehensive rendering package supports line art, text, and images in a flexible, full-featured framework for developing richer user interfaces, sophisticated drawing programs and image editors.

The Java 2D API implements an imaging model that supports the manipulation of fixed-resolution images stored in memory. A new `Image` class in the `java.awt.image` package, `BufferedImage`, can be used to hold and to manipulate image data retrieved from a file, URL or an appropriate *Mathematica* expression. The class `BufferedImage` enables you to perform a variety of image-filtering operations, such as blur and sharpen, affine spatial transformations and image resampling/resizing operations.

```
In[1]:= << JLink`
```

```
In[2]:= InstallJava[];
```

This reads an image using the *Digital Image Processing* function `ImageRead`.

```
In[3]:= << ImageProcessing`
```

```
In[5]:= img = ImageRead["cameraman.tif"]
```

```
Out[5]= -ImageData-
```

Mathematica and Java image data objects

The *Digital Image Processing* package introduced a convenient new *Mathematica* object called `ImageData` which encapsulates information about an image including the pixel values, the particular organization of the data and the color format. The `BufferedImage` Java object introduced in JDK 1.2 plays a similar role in the Java environment. Here we show how to translate grayscale image data between the two environments.

This determines the dimensions of the image and instantiates a `BufferedImage` object of appropriate size and data type.

```
In[6]:= {h, w} = ImageDimensions[img];
```

```
bi = JavaNew["java.awt.image.BufferedImage", w, h, BufferedImage`TYPEUBYTEUGRAY]
```

```
Out[7]= «JavaObject[java.awt.image.BufferedImage]»
```

This populates the `BufferedImage` with raw data extracted from the *Mathematica* `ImageData` expression `img`.

```
In[8]:= r = bi@getRaster[];
```

```
r@setPixels[0, 0, w, h, Flatten[Reverse[RawImageData[img]]]]];
```

Here I display the image in a Java window.

```
In[10]:= frm = JavaNew["com.wolfram.jlink.MathJFrame"];
```

```
canvas = JavaNew["com.wolfram.jlink.MathCanvas"];
```

```
frm@setSize[300, 300];
```

```
pane = frm@getContentPane[];
```

```
canvas@setImage[bi];
```

```
pane@add[canvas];
```

```
JavaShow[frm];
```

Alternatively, given a `BufferedImage` we wish to convert to an `ImageData` object. This creates an array to hold the raw image data.

```
In[17]:= r = bi@getData[];
```

```
w = r@getWidth[];
```

```
h = r@getHeight[];
```

```
pix = JavaNew["I", w * h * r@getNumBands[]];
```

The following returns a grayscale `ImageData` object. Note that image orientations in Java and *Mathematica* are reversed.

```
In[21]:= ToGrayLevel[Reverse[Partition[r@getPixels[0, 0, w, h, pix], w]]]
```

```
Out[21]= -ImageData-
```

```
In[22] := Show[Graphics[%]];
```



The data translation operations shown above for grayscale images have been generalized to color images and collected in a `BufferedImage` package. This loads the package for use in the remainder of the presentation.

```
In[23] := << ImageProcessing`BufferedImage`
```

Image processing example: filtering

The following example demonstrates an image smoothing operation using Java 2D functionality. It also shows the ease with which *Mathematica* may be used to script a sequence of Java operations.

This defines a simple smoothing filter using the *Digital Image Processing* function `BoxFilter`.

```
In[24] := ker = JavaNew["java.awt.image.Kernel", 5, 5, Flatten[BoxFilter[5, 5]]];
```

Here we define the operator and instantiate the destination `BufferedImage` object.

```
In[25] := cop = JavaNew["java.awt.image.ConvolveOp", ker];
          dst = JavaNew["java.awt.image.BufferedImage",
                      bi@getWidth[], bi@getHeight[], bi@getType[]];
```

This performs the operation and returns a *Mathematica* expression containing the image data.

```
In[27] := cop@filter[bi, dst];
          blur = BufferedImageToImageData[dst];
```

This displays the result.

```
In[29] := Show[GraphicsArray[{Graphics[img], Graphics[blur]}, ImageSize -> {540, 256}];
```



Java Advanced Imaging API

The Java™ Advanced Imaging API enables developers to easily incorporate high-performance, network-enabled, scalable, platform-independent image processing into Java technology and *Mathematica*-based applications (see [4]).

The Java Advanced Imaging API provides a set of object-oriented interfaces that support a simple, high-level programming model which allows developers to manipulate images easily. Examples of image processing techniques available in JAI range from simple operations such as contrast enhancement, cropping, and scaling to more complex operations such as advanced geometric warping and frequency domain processing. There are approximately 80 operators, many with native support for optimized performance.

Here are a few examples of image processing operations using JAI. Here I load the JAI interface.

```
In[30] := jai = JavaNew["javax.media.jai.JAI"];
```

Here I load a new image and convert to a `BufferedImage` Java object.

```
In[31] := img = ImageRead["peppers.tif"];
         bi = ImageDataToBufferedImage[img];
```

Image processing examples:

Convolution

Here we define the convolution kernel, the desired operation, and return the result. Note that `BufferedImage` objects are fully compatible with JAI and thus may be used as source image data with any of the JAI operators.

```
In[33]:= ker = JavaNew["javax.media.jai.KernelJAI", 5, 5, Flatten[BoxFilter[5, 5]]];
pb = JavaNew["java.awt.image.renderable.ParameterBlock"];
pb@addSource[bi];
pb@add[ker];
dst = jai@create["convolve", pb]
```

```
Out[37]= «JavaObject[javax.media.jai.RenderedOp] »
```

The following converts the result to a Java BufferedImage object for subsequent translation to a *Mathematica* expression using BufferedImageToImageData.

```
In[38]:= dst@getAsBufferedImage[]
```

```
Out[38]= «JavaObject[java.awt.image.BufferedImage] »
```

This shows the result.

```
In[39]:= Show[GraphicsArray[{Graphics[img], Graphics[BufferedImageToImageData[dst]]}],
ImageSize -> {540, 256}];
```



Most other JAI image operators may be implemented in a similar manner.

Morphological erosion

Erosion and dilation are two fundamental morphological operators. Many higher-level image processing operations (edge detection, noise removal, line thinning and more) may be defined using repetitive applications of these two operators. The following example demonstrates erosion, including a method for padding of image data to minimize the edge effect visible in earlier examples.

```
In[40]:= pb = JavaNew["java.awt.image.renderable.ParameterBlock"];
ker = JavaNew["javax.media.jai.KernelJAI", 5, 5, Table[0, {25}]];
pb@addSource[bi];
pb@add[ker];
```

This defines a method of extending the border and adds it to the erode operator in the form of rendering hints.

```
In[44] := LoadClass["javax.media.jai.BorderExtender"];
         extender = BorderExtender`createInstance[BorderExtender`BORDERUCOPY];
         hints = JavaNew["java.awt.RenderingHints", JAI`KEYUBORDERUEXTENDER, extender];
         dst = jai@create["erode", pb, hints];
```

This shows the result.

```
In[48] := Show[GraphicsArray[{Graphics[img], Graphics[BufferedImageToImageData[dst]]}],
              ImageSize -> {540, 256}];
```



Rotation

Here is an example of a simple geometric operation. First we define the parameterblock and the interpolation scheme.

```
In[49] := pb = JavaNew["java.awt.image.renderable.ParameterBlock"];
         inter = JavaNew["javax.media.jai.InterpolationBilinear"];
```

This defines the rotation parameters and adds them to the parameter block. The required parameters are the coordinates of the pivot point and the angle of rotation (in radians/second).

```
In[51] := pb@addSource[bi];
         pb@add[JavaNew["java.lang.Float", 100.]];
         pb@add[JavaNew["java.lang.Float", 100.]];
         pb@add[JavaNew["java.lang.Float", N[17 Degree]]];
         pb@add[inter];
         dst = jai@create["rotate", pb];
```

This shows the result.

```
In[57]:= Show[GraphicsArray[{Graphics[img], Graphics[BufferedImageToImageData[dst]]}],
  ImageSize -> {540, 256}];
```



Performance

I conclude this presentation with a brief performance comparison. With one exception, the functionality of the Java 2D and JAI API's is a subset of the Digital Image Processing package. Therefore the primary reason for moving native *Mathematica* implementations to Java would be to gain a performance advantage. Such indeed is the case with some, but not all, of the functions. Here is a tabular summary of selected comparisons (all values in seconds, Pentium III @ 500 MHz):

Operation	<i>Mathematica</i>	Java2D	JAI
convolution	0.344	0.7	0.7
medianfilter	2.353	<i>n/a</i>	0.16
colorconversion	0.594	1.8	3.0
grayscale erosion	8.172	<i>n/a</i>	0.95
rotation	8.75	0.6	1.3

Summary

The imaging functionality of Java 2D and JAI APIs was briefly described. Examples were shown of how to smooth an image using convolution operators available in both of the APIs. A translation mechanism between Java and *Mathematica* image data objects was also demonstrated. *J/Link* served as the all-important link between the Java and *Mathematica* environments with profound potential benefits to the image processing professional.

References

- [1] URL: <http://www.wolfram.com/products/applications/digitalimage/>
- [2] URL: <http://www.wolfram.com/solutions/mathlink/jlink/>
- [3] URL: <http://java.sun.com/products/java-media/2D/>

[4] URL: <http://java.sun.com/products/java-media/jai/>