Java Imaging Technology

Diksha Verma, Anjli Tyagi, Deepak Sharma
Department of Information Technology, Dronacharya college of Engineering

Abstract- **Java Advanced Imaging (JAI) is a Java platform extension API that provides a set of object-oriented interfaces that support a simple, high-level programming model which allows developers to create their own image manipulation routines without the additional cost or licensing restrictions, associated with commercial image processing software.**
with rapid increase on requirement of the cross- platform distributed imagin technology and the
common stable imaging lib, SUN provided a complete solution based java technology
this paper has tried to analyse the evolution of java imaging technology

JAI is provided as a free download directly from Oracle Corporation for the Windows, Solaris, and Linux platforms. Apple Inc. provides a Mac OS X version of the API from their website for Mac OS X v10.3; Mac OS X v10.4 and later ships with it preinstalled.

While the API is provided in Java, platform-specific implementations can either use the pure Java implementation or provide an implementation that takes advantage of native technology on the platform to provide better performance.

I. INTRODUCTION

Java Advanced Imaging helps developers compete in today's heterogeneous, highly distributed networked environments by simplifying the creation of imaging applications that support a broad range of systems, from thin clients to powerful workstations. The imaging technology greatly improves developers' ability to implement portable image processing applications and its flexible, scalable design meets the demands of the geospatial, medical, commercial, network and government imaging markets. Benefits of JAI include:

* A standard interface for cross-platform imaging.
* Enabling the same application to be deployed on multiple platforms without source code changes.
* Simplified programming tasks and increased code reuse.
* Reduced time to market by enabling solutions to be deployed faster and at lower costs.
* Enhanced client/server imaging through a network-centric architecture and complementary technologies.
* Rapid adoption of highly optimized implementations that use the media capabilities of CPUs.
* Integration with the Java Media Application Programming Interface (API) family, enabling deployment of media-rich applications.

Java Advanced Imaging (JAI) and the associated **Image I/O Tools** packages are now community source projects on java.net. Please see our Project Structure and Mailing List page for more information.

JavaBeans Integration

BeanStalk's inspiration came from the promise ofrapid prototyping ofinteractive illustrations using JavaBeans in a visual authoring environment -hence the *Bean* in BeanStalk. The idea was to provide a Bean component library especially tailored to the needs of illustration developers so that they could focus on content creation instead of software engineering busywork. While key players in the industry were quick to endorse the JavaBeans component model, none of them at the time of this writing have yet produced a solid, shippable product that fully supports Beans as they are described in the JavaBeans specification. Because of this, BeanStalk was not able to take on the Bean features that were originally intended for it. Still, JavaBeans conventions were used in BeanStalk wherever possible with the hope that someday they could be turned into Bean components. Once commercial JavaBeans development environments become mature enough to allow developers to create, test, and package Beans without ever having to resort to a command line tool, then BeanStalk can begin to reveal its true nature. For the time being, interactive illustration developers will have to use it in the traditional, non-visual approach to application development.

The Evolution of Imaging in Java

Early versions of the Java AWT provided a simple rendering package suitable for rendering common HTML pages, but without the features necessary for complex imaging. The early AWT allowed the generation of simple images by drawing lines and shapes. A very limited number of image files, such as GIF and JPEG, could be read in through the use of a Toolkit object. Once read in, the image could be displayed, but there were essentially no image processing operators.

The Java 2D API extended the early AWT by adding support for more general graphics and rendering operations. Java 2D added special graphics classes for the definition of geometric primitives, text layout and font definition, color spaces, and image rendering. The new classes supported a limited set of image processing operators for blurring, geometric transformation, sharpening, contrast enhancement, and thresholding. The Java 2D extensions were added to the core Java AWT beginning with the Java Platform 1.2 release.

The Java Advanced Imaging (JAI) API further extends the Java platform (including the Java 2D API) by allowing sophisticated, high-performance image processing to be incorporated into Java applets and applications. JAI is a set of classes providing imaging functionality beyond that of Java 2D and the Java Foundation classes, though it is compatible with those APIs.

JAI implements a set of core image processing capabilities including image tiling, regions of interest, and deferred execution. JAI also offers a set of core image processing operators including many common point, area, and frequency-domain operators.

JAI is intended to meet the needs of all imaging applications. The API is highly extensible, allowing new image processing operations to be added in such a way as to appear to be a native part of it. Thus, JAI benefits virtually all Java developers who want to incorporate imaging into their applets and applications.

1.2 Why Another Imaging API?

Several imaging APIs have been developed - a few have even been marketed and been fairly successful. However, none of these APIs have been universally accepted because they failed to address specific segments of the imaging market or they lacked the power to meet specific needs. As a consequence, many companies have had to "roll their own" in an attempt to meet their specific requirements.

Writing a custom imaging API is a very expensive and time-consuming task and the customized API often has to be rewritten whenever a new CPU or operating system comes along, creating a maintenance nightmare. How much simpler it would be to have an imaging API that meets everyone's needs.

Previous industry and academic experience in the design of image processing libraries, the usefulness of these libraries across a wide variety of application domains, and the feedback from the users of these libraries have been incorporated into the design of JAI.

JAI is intended to support image processing using the Java programming language as generally as possible so that few, if any, image processing applications are beyond its reach. At the same time, JAI presents a simple programming model that can be readily used in applications without a tremendous mechanical programming overhead or a requirement that the programmer be expert in all phases of the API's design.

JAI encapsulates image data formats and remote method invocations within a re-usable image data object, allowing an image file, a network image object, or a real-time data stream to be processed identically. Thus, JAI represents a simple programming model while concealing the complexity of the internal mechanisms.

1.3 JAI Features

JAI is intended to meet the requirements of all of the different imaging markets, and more. JAI offers several advantages for applications developers compared to other imaging solutions. Some of these advantages are described in the following paragraphs.

Cross-platform Imaging

Whereas most imaging APIs are designed for one specific operating system, JAI follows the Java run time library model, providing platform independence. Implementations of JAI applications will run on any computer where there is a Java Virtual Machine. This makes JAI a true cross-platform imaging API, providing a standard interface to the imaging capabilities of a platform. This means that you write your application once and it will run anywhere.

Distributed Imaging

JAI is also well suited for client-server imaging by way of the Java platform's networking architecture and remote execution technologies. Remote execution is based on Java RMI (remote method invocation). Java

RMI allows Java code on a client to invoke method calls on objects that reside on another computer without having to move those objects to the client.

Object-oriented API

Like Java itself, JAI is totally object-oriented. In JAI, images and image processing operations are defined as objects. JAI unifies the notions of image and operator by making both subclasses of a common parent.

An operator object is instantiated with one or more image sources and other parameters. This operator object may then become an image source for the next operator object. The connections between the objects define the flow of processed data. The resulting editable graphs of image processing operations may be defined and instantiated as needed.

Flexible and Extensible

Any imaging API must support certain basic imaging technologies, such as image acquisition and display, basic manipulation, enhancement, geometric manipulation, and analysis. JAI provides a core set of the operators required to support the basic imaging technologies. These operators support many of the functions required of an imaging application. However, some applications require special image processing operations that are seldom, if ever, required by other applications. For these specialized applications, JAI provides an extensible framework that allows customized solutions to be added to the core API.

JAI also provides a standard set of image compression and decompression methods. The core set is based on international standards for the most common compressed file types. As with special image processing functions, some applications also require certain types of compressed image files. It is beyond the scope of any API to support the hundreds of known compression algorithms, so JAI also supports the addition of customized coders and decoders (codecs), which can be added to the core API.

Device Independent

The processing of images can be specified in device-independent coordinates, with the ultimate translation to pixels being specified as needed at run time. JAI's "renderable" mode treats all image sources as rendering-independent. You can set up a graph (or chain) of renderable operations without any concern for the source image resolution or size; JAI takes care of the details of the operations.

To make it possible to develop platform-independent applications, JAI makes no assumptions about output device resolution, color space, or color model. Nor does the API assume a particular file format. Image files may be acquired and manipulated without the programmer having any knowledge of the file format being acquired.

Powerful

JAI supports complex image formats, including images of up to three dimensions and an arbitrary number of bands. Many classes of imaging algorithms are supported directly, others may be added as needed.

JAI implements a set of core image processing capabilities, including image tiling, regions of interest, and deferred execution. The API also implements a set of core image processing operators, including many common point, area, and frequency-domain operations.

High Performance

A variety of implementations are possible, including highly-optimized implementations that can take advantage of hardware acceleration and the media capabilities of the platform, such as MMX on Intel processors and VIS on UltraSparc.

Interoperable

JAI is integrated with the rest of the Java Media APIs, enabling media-rich applications to be deployed on the Java platform. JAI works well with other Java APIs, such as Java 3D and Java component technologies. This allows sophisticated imaging to be a part of every Java technology programmer's tool box.

JAI is a Java Media API. It is classified as a Standard Extension to the Java platform. JAI provides imaging functionality beyond that of the Java Foundation Classes, although it is compatible with those classes in most cases.

A Simple Java Advanced Imaging Program

Before proceeding any further, let's take a look at an example JAI program to get an idea of what it looks like. Listing 1-1 shows a simple example of a complete

JAI program. This example reads an image, passed to the program as a command line argument, scales the image by 2x with bilinear interpolation, then displays the result.

Listing 1-1 **Simple Example JAI Program**

```
import java.awt.Frame;
import java.awt.image.renderable.ParameterBlock;
import java.io.IOException;
import javax.media.jai.Interpolation;
import javax.media.jai.JAI;
import javax.media.jai.RenderedOp;
import com.sun.media.jai.codec.FileSeekableStream;
import javax.media.jai.widget.ScrollingImagePanel;
/**
 * This program decodes an image file of any JAI supported
 * formats, such as GIF, JPEG, TIFF, BMP, PNM, PNG, into a
 * RenderedImage, scales the image by 2X with bilinear
 * interpolation, and then displays the result of the scale
 * operation.
 */
public class JAISampleProgram {
  /** The main method. */
  public static void main(String[] args) {
    /* Validate input. */
    if (args.length != 1) {
      System.out.println("Usage: java JAISampleProgram " +
                         "input_image_filename");
      System.exit(-1);
    }
    /*
     * Create an input stream from the specified file name
     * to be used with the file decoding operator.
     */
    FileSeekableStream stream = null;
    try {
      stream = new FileSeekableStream(args[0]);
    } catch (IOException e) {
      e.printStackTrace();
      System.exit(0);
    }
    /* Create an operator to decode the image file. */
    RenderedOp image1 = JAI.create("stream", stream);
    /*
     * Create a standard bilinear interpolation object to be
     * used with the "scale" operator.
     */
    Interpolation interp = Interpolation.getInstance(
        Interpolation.INTERP_BILINEAR);
    /**
     * Stores the required input source and parameters in a
     * ParameterBlock to be sent to the operation registry,
     * and eventually to the "scale" operator.
     */
    ParameterBlock params = new ParameterBlock();
    params.addSource(image1);
    params.add(2.0F);       // x scale factor
    params.add(2.0F);       // y scale factor
    params.add(0.0F);       // x translate
    params.add(0.0F);       // y translate
    params.add(interp);     // interpolation method
    /* Create an operator to scale image1. */
    RenderedOp image2 = JAI.create("scale", params);
    /* Get the width and height of image2. */
    int width = image2.getWidth();
    int height = image2.getHeight();
    /* Attach image2 to a scrolling panel to be displayed. */
    ScrollingImagePanel panel = new ScrollingImagePanel(
                                    image2, width, height);
    /* Create a frame to contain the panel. */
    Frame window = new Frame("JAI Sample Program");
    window.add(panel);
    window.pack();
    window.show();
  }
}
```

Overview of the Java 2D API Concepts

The Java 2D API provides two-dimensional graphics, text, and imaging capabilities for Java programs through extensions to the Abstract Windowing Toolkit (AWT). 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. Java 2D objects exist on a plane called user coordinate space, or just

user space. When objects are rendered on a screen or a printer, user space coordinates are transformed to *device space coordinates*. The following links are useful to start learning about the Java 2D API:

- Graphics class
- Graphics2D class

The Java 2D API provides following capabilities:

- A uniform rendering model for display devices and printers
- A wide range of geometric primitives, such as curves, rectangles, and ellipses, as well as a mechanism for rendering virtually any geometric shape
- Mechanisms for performing hit detection on shapes, text, and images
- A compositing model that provides control over how overlapping objects are rendered
- Enhanced color support that facilitates color management
- Support for printing complex documents
- Control of the quality of the rendering through the use of rendering hints

These topics are discussed in the following sections:

- Java 2D Rendering
- Geometric Primitives
- Text
- Images
- Printing

Morphological Filtering

Edge detection can be achieved by convolution or by morphological filtering. Morphological filtering is far more accurate as it produces an edge that is only a single pixel-width wide. Morphological filters manipulate the shape of an object and work better for binary or grey-level images.

Morphological filtering is like convolution, except that it uses sets as opposed to multiplication and addition. The centre of a kernel is moved around the image one pixel at a time and a set operation is performed on every pixel that overlaps this kernel. This is best performed on a grey image using a kernel of odd dimensions (so that the centre can be found).

To find the inside edges of an object using morphological filtering it is necessary to use an erosion filter (to find the outside edge a dilation filter is required), this reduces the size of an object by eroding the boundary of it (a dilation filter would add to the boundary) and subtract the image from itself.

$$A = \{a_0, a_1, \ldots a_{n-1}\}$$

Figure 3.7 – Two Dimensional integer space

At this point it useful to discuss some set theory[6], for images we must restrict the sets to point sets in a two-dimensional integer space, Z^2 (figure 3.7).

$$A + x = \{a + x : a \in A\}$$

Figure 3.8 – Point Translation

Each element in *A* is a two-dimensional point consisting of integer co-ordinates. If the points contained in *A* were drawn they would show an image. It is possible to translate a point set by another point *x* (figure 3.8).

After defining a structuring element *B* (which can be used to measure the structure of *A*) the equation shown in 3.8 is used to translate *B* inside of *A*, thus creating a new set *C*. *C* consists of a translation of the elements in *A* by the elements in *B.*

$$C = A[-]B = \cap(A - b : b \in B)$$
$$C = \{x : B + x \subset A\}$$

Figure 3.9 – Set Erosion

Thus erosion is defined as (figure 3.9):

In computing terms this is achieved by eroding the boundaries of each object:

- For every pixel in the image
- Store in the current pixel the minimum value of all the pixels surrounding it. (During erosion all the surrounding pixels carry an equal weighting, using the symmetrical structuring element shown in figure 3.10).

1	1	1
1	1	1
1	1	1

Figure3.10 – Symmetric Structuring Element

To reduce the image further (instead of just being areas

of black and white) to single lines around an object an outlining filter must be used, this is quite simply done by subtracting the image from itself to create the inside contour (figure 3.11).

$$insideContour(A,B) = A[/](A[-]B)$$

Figure3.11 – Outlining Filter

Subtracting the arrays of data obtained via this method from themselves results in a black image, with white edges of objects (an example is shown in figures 3.12 & 3.13).

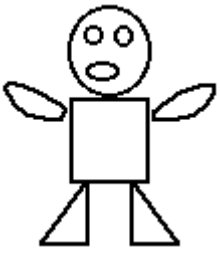


Figure 3.12 – Input Image

Figure 3.13 – Image after subtraction and erosion

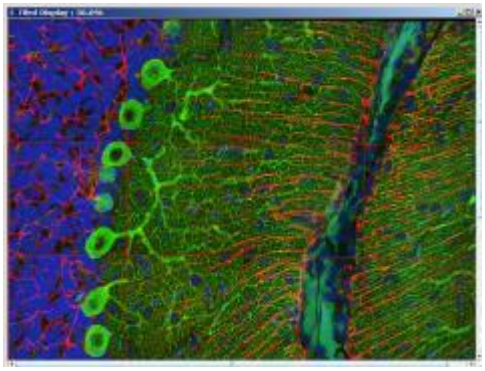
Java Advanced Imaging -- National Center for Microscopy and Imaging Research

Automated Mosaicking, Tomographic Dataset Reconstruction, and Remote Collaborative Telemicroscopy

The National Center for Microscopy and Imaging Research (NCMIR) at the University of California San Diego (UCSD) develops state-of-the-art 3D imaging and analysis technologies to help biomedical researchers understand biological structure and function relationships in cells and tissues. For more information, please visit http://ncmir.ucsd.edu/.

ImageJ Mosaic plugins

We have developed an automated mosaic acquisition and processing solution around the integration of a Nanomotion® -II motorized stage (Applied Precision LLC) and a Bio-Rad RTS 2000 2-photon confocal microscope. The processing involved in aligning and assembling the individually acquired images into a mosaic was developed using the ImageJ plugin framework with the core of the processing routines implemented using the JAI libraries. Navigation of the assembled mosaics was implemented with the JAI and the JadeDisplay libraries. Without JAI's paradigm of processing images in the form of discrete tiles, the assembling of the thousands of images involved in creating a mosaic would be inefficient if not impossible. By utilizing JAI's facilities, datasets of up to 60 gigabytes have been processed and future improvements are planned to push this limit even further.



Jinx

Jinx was developed to aid in the 3D reconstruction of tomographic datasets acquired with one of the various electron microscopes available at the resource. Tomographic datasets consist of a series of 2D images from which objects of interest are segmented out for the 3D reconstruction. Jinx offers the user a graphical interface to step through each image of the series and facilities to manually trace out objects of interest. It relies on JadeDisplay to support the display of large images with graphical overlays and the JAI libraries for histogram functionality and other types of image manipulations. Jinx is currently under active development and future releases will offer semi-automated segmentation algorithms based on fuzzy logic, level set, and watershed algorithms.

EMWorkspace

The remote operation of the various electron microscope instruments available at the center has long been a focus of our engineering and software development. EMWorkspace is our most recent software effort to assist in remote and collaborative telemicroscopy sessions. It offers numerous tools for user collaboration, microscope control, image acquisition, and displaying of a live video stream during a session. To accommodate the different instruments, EMWorkspace dynamically changes the user interface to adjust for the functionalities offered by the different instruments. Future development of this software will include improvements to the live video subsystem and the integration of the Cell Centered Database (CCDB) for image storage.

CONCLUSION

The Java Advanced Imaging API (JAI) provides a set of object-oriented interfaces that supports a simple, high-level programming model which allows images to be manipulated easily in Java applications and applets. JAI goes beyond the functionality of traditional imaging APIs to provide a high-performance, platform-independent, extensible image processing framework.

REFERENCES

1. Sun Microsystems. Programming in Java Advanced Imaging, Release 1.0.1, Palo Alto, CA (November 1999)

2. Santos, R.: J Java Advanced Imaging API: A Tutorial. RITA XI(1) (2004),

http://www.inf.ufrgs.br/~revista/docs/ritall/rita_v11_n11_n1_p93a124.pdf

3. Java AWT Imaging, http://www2.hs-fulda.de/caelabor/inhalte/java/j3d/j3d_seminar/19/JAI%20Guide%20von%20Sun/J2D-concepts.doc.html

4. Lesson: Overview of the Java 2D API Concepts (The Java™ Tutorials > 2D Graphics), https://docs.oracle.com/javase/tutorial/2d/overview/