Interactive Textbooks; Embedding Image Processing Operator Demonstrations in Text

Robert B. Fisher and Konstantinos Koryllos
Dept. of Artificial Intelligence, Univ. of Edinburgh

Short title: Interactive Image Processing Textbooks

Abstract

Traditional image processing teaching has used materials where the theory and drill are separated into textbooks and image processing packages. HTML and JAVA might allow easier construction of an integrated teaching resource. Such a resource would have widespread, platform-independent accessibility. This paper reports our assessment of this potential, which is explored through extensions of the HIPR teaching materials. Our conclusions are that the approach is feasible and attractive to students, that a few standard programming protocols reduce development time, and that use of compiled JAVA is essential.

Keywords: teaching image processing, interactive teaching resources, JAVA-based image processing

1 Introduction

People are usually far better at remembering interactive rather than static material. Interaction with knowledge develops deep learning rather than textbook memorization, thus students can benefit greatly from the introduction of interactive technology. In the past few years many universities and colleges have produced their own World Wide Web pages with links to teaching material, papers, exercises *etc.*, so that the rest of the world can benefit from their work. This idea has had a great response, resulting in huge amounts of information becoming available to any user with Internet access and a browser. The typical educational WWW page is a hypertext document written in HTML (HyperText Mark-up Language), sometimes containing multimedia objects such as embedded images, sounds, demonstrations, *etc.* However, what is needed is a real hands on application, for which JAVA is ideal. What we ought to have are "interactive textbooks".

The project described here explores the potential for JAVA to implement and link a variety of image processing operators within a HTML hypertext document, for the purpose of creating "interactive textbooks". (Image processing is here defined as the transformation of (at least) one image to another.) The project explored functional (*i.e.* "Does JAVA have enough programming facilities?"), practical (*i.e.* "Is JAVA fast enough and does it have enough computational resources?") and educational (*i.e.* "Is the combination usable, attention-getting and informative?") issues. The final product can be viewed using a JAVA-enabled web-browser and the operators communicate by passing on output data to each other.

The idea of an interactive teaching package is well-accepted, and has even been applied to image processing [19, 7, 12], however, projects are usually limited to use on the platforms for which they were designed, which, as in the case of [19], may become obsolete very rapidly. The advantage of using JAVA is that it is platform independent and seems likely to be publically available for a long time. The DIP course [7, 12] is an on-line HTML-based package. It also has an active exploration element (in conjunction with the use of the KHOROS [8] image processing package), but has limited explanatory text and requires the use of KHOROS. Thus, it really supplements text-book based teaching. The Pennsylvania State University approach is partly similar, in that

it supplies executable on-line operators and image libraries, and is also in JAVA, which improves usability, but also has only limited explanatory text. The project described here incorporates all of the desirable features:

- online operators for active exploration,
- portable platform, and
- direct integration of the demonstration materials with the instructional text.

To investigate the suitability of JAVA for use in interactive image processing textbooks, representatives of the main classes of computations typical to image processing were chosen for implementation. They include:

Point Operators are applied to individual pixels and are, therefore, position independent operators. Two popular representatives of this category are thresholding and gamma correction.

Image Arithmetic combines two or more images to produce a single one as an output. Image subtraction, and logical AND/OR are common examples.

Geometric Operations *e.g.* rotation, translation and scaling - affecting the position but not the content of the image data.

Morphology take as input a binary image and a structuring element, and output a function of the two - typically linked by the relative spatial distribution of the pixels, rather than their values.

Digital Filters are often used for smoothing or enhancing features in images and are based on a two-dimensional convolution operation (which expresses a linear filtering process applied to an image). The convolution of two functions f and g is defined by

$$g(x,y) \ * \ f(x,y) = \int \int f(\alpha,\beta)\, g(x-\alpha, y-\beta)\, d\alpha\, d\beta$$

where $f(x,y)$ represents the image and $g(x,y)$ the kernel (see [17]).

These classes of operators were chosen because they exemplified many of the different algorithmic paradigms found in image processing, and thus allowed us to explore both implementation and usability issues.

Our main conclusions are that, in general, JAVA is suitable, integration with HTML is straightforward, the execution time of interpreted JAVA is sub-optimal but compiled JAVA performs satisfactorily. One has to take care not to misuse the internal thread resources of JAVA as the browser implementing the JAVA Virtual Machine may become overloaded.

2 Background

JAVA [18], is an object-oriented language developed by Sun Microsystems, modeled after C++ [11], but designed to be small, simple and machine independent. JAVA programs belong in two categories: applets and applications. Applets are programs that are referenced through an HTML document and are down-loaded over the WWW and executed by the Web browser on the user's machine. Applications are simply conventional stand-alone programs written in the JAVA language. Due to the educational focus of this project, we used applets rather than applications.

When JAVA code is compiled, the result is not directly executable, but must be interpreted by each computer (using a web browser or other tools). This first 'compilation' produces *bytecodes* from the JAVA source code. (Bytecodes are a set of machine-independent instruction codes that can be executed using a special-purpose, machine-dependent interpreter.) So that every computer in the world can run the same bytecodes, JAVA has assumed the existence of a virtual machine which the bytecode-interpreter (*e.g.* a WWW Browser) will implement. This scheme works fine for most programs but if more speed is required then two solutions are available:

- Use of native C/C++ code.
- Just-in-Time Compilers.

JAVA has mechanisms that will allow compiled C/C++ to be used in order to improve performance where required. While this solution is fine for applications executed locally, it runs into the

portability constraints that all binary executables have. As we are interested in educational materials usable on many platforms, the C/C++ option is impractical. The better solution is Just-in-Time compilers which incrementally translate JAVA bytecodes into native machine code. This will enable JAVA programs to run at almost the same speed as C. A Just-in-Time compiler is scheduled to be available in the next release of SUN's Solaris operating system, but other commercial compilers are also available.

HIPR or Hypermedia Image Processing Reference [2] is a project undertaken by the Machine Vision Unit in the Department of Artificial Intelligence at the University of Edinburgh. Its purpose is to help students learn about image processing by aiming for a balance between too few examples and too much technical information. The majority of textbooks in image processing and machine vision typically err on the side of not providing sufficient intuition-building examples, while image processing software packages supply the user with little theory. HIPR is an HTML based teaching package designed for local network use (and sold under site license by John Wiley and Sons). The *Worksheets* section of HIPR includes a series of 42 topics starting from image arithmetic to morphology and image transforms. These worksheets contain a brief summary of the theoretical background on each topic plus a number of examples of applications, including numerous before-and-after (processing) example images. This way the student can comprehend the exact effects of the particular operator as well as review its theoretical basis. An example of a worksheet on the image addition operator can seen at URL:
`http://www.dai.ed.ac.uk/daidb/staff/personal_pages/rbf/HIPR/hiprdemo/html/pixadd.htm`.
What this package is missing is an interactive element and that's where JAVA fits in.

An initial investigation of JAVA for image processing was undertaken in the MARBLE Project [4] (see Figure 1) to determine if JAVA was basically suitable. Amongst other issues, the project investigated a "thresholding" applet with a slider to set the threshold value acting on a given source image. This demonstration, coupled with the relevant background theory (in HTML) made for an entertaining and educational experience.

The project described here made use of some of the JAVA classes written for the Marble project.

The most useful class was *ImageCanvas()* whose purpose is to "tie an image to a canvas, wait until its size is known, resize the canvas and later update the on-screen image".

Murdoch [15] experimented with the development of a greatly simplified Khoros-like [8] environment for image processing that would allow users with little experience in the field to experiment with various operations and their effects. The project developed pull-down instances of operators that could be connected to form longer image transformation processes.

3 Java's Image Processing Support

JAVA offers a variety of functions dedicated to the manipulation of images, as well a *Graphics* class which implements the usual set of drawing primitives. The support for using images is spread out between the *java.applet*, *java.awt* and *java.awt.image* packages. The latter package contains the support for manipulating images that have been already loaded. The loading of an image is achieved through the *getImage()* method (*i.e.* a subroutine or function) of an Applet instance (*i.e.* the Applet might be attached to execute in more than one place independently) by supplying a URL (Uniform Resource Locator) parameter (*i.e.* the path name of the image to be loaded).

To display an image one invokes the *drawImage()* method of the Graphics object which is passed to the applet's *update()* or *paint()* method. One can keep track of the state of the image(s) using an instance of a *MediaTracker* class and use the methods provided. The *ImageObserver* interface can be used for even closer monitoring of an image.

3.1 The Java Color Model

JAVA has a class named *Color*. In order to create a Color object, the red, green and blue components must be specified as floating point numbers between zero and one. Alternatively one can specify hue, saturation and brightness and the equivalent color will be created. Internally JAVA stores color components as 8-bit values between 0 and 255. An RGB color is a 32-bit integer of the form:

0xAARRGGBB

where AA represents the image transparency value (*i.e.* the extent to which any images displayed behind the current image are visible, which is not relevant here), and RR, GG, BB the red, green and blue values. The JAVA image class allows the programmer to access the image as a one-dimensional array. Pixel values are stored in each element of the array in the above form.

3.2 Pixel Independent Operations

The *ImageProducer* interface represents an image source and defines the methods which must be implemented by classes wishing to communicate with *ImageConsumer* classes. The *ImageConsumer* interface defines a set of methods which must be implemented by any class that desires to consume image data from another class that produces it. These methods should only be called by the *ImageProducer* that wishes to pass image (and other) data to the *ImageConsumer*. Behind the scenes image data is created according to Figure 2. JAVA allows the user to modify an image by inserting an image filter between the producer and the consumer of the image. The producer then sends the data which will be modified by the filter before reaching the consumer.

3.3 Single Pixel *vs.* Neighborhood Operations

JAVA has optimized single-pixel operations by giving the user the option to directly filter the colormap (a fast and global operation) instead of working with image pixels. However, in neighborhood operations we can no longer simply modify the colormap as with pixel operations. A more general solution is to produce new data for a standard colormap. The most straightforward way to achieve this is to convert the image into an array of numbers which can then be processed as desired. For this purpose, the *PixelGrabber* class is used. This class implements the *ImageConsumer* interface and is used to extract a requested rectangular array of pixels from an *Image* object. These pixels are stored in a one-dimensional array of integers in the RGB format described earlier.

4 Applets

Appendix A describes the generic JAVA framework used in all of the applets. This section presents some of the individual applets developed during the project. The choice of operators implemented was based on two criteria: (1) the set of operators could be linked into a sequence and (2) the operators would explore a variety of interactive facilities and image processing algorithmic paradigms. These facilities included: text string inputs to specify test images, scalar parameter inputs, convolution kernel entry, chaining of operators, pointwise, local neighborhood and global operators, in-place and translated results, and single and multiple image inputs.

In the sections below, in some cases we go into the details of the operator algorithms more than would ordinarily be expected (given they are textbook material now), because the details are helpful for understanding the timing results presented in Section 6.

4.1 Gamma Correction

The gamma transform [3] is an instance of a single pixel operator in which the input and output pixels are related by a function $out\_pixel = f(in\_pixel)$. Figure 3 shows the original image on the left. The gamma value is entered in the text field and pressing the "return" key initiates the operation.

4.2 Rotation

Rotation is a non-local operation that affects the pixel locations but not the values themselves. Simple rotation [17] (see Fig 4) fills in the destination image in raster order by point-sampling the source image. The optimization lies in the fact that the expensive matrix multiplication is done once outside the inner loop. The inner loop itself contains step additions for going from one pixel to the next (in the source image) as well as some conditional statements to check for pixels which map to positions outside the boundaries of the source image.

4.3 3x3 Convolution

Convolution (see Figure 5) [6] is a local neighborhood operator that uses a user-defined mask.

The kernel values are entered in each of the nine text areas of the applet, each one representing the corresponding value of a 3x3 kernel. Floating point values are allowed. By pressing the "Apply Convolution" button, the applet will commence its execution.

4.4 Noise Generation

We experimented with salt-pepper and gaussian noise [2].

Salt-pepper noise is produced by corrupting the original image so that individual pixels are randomly flipped to black or white (0 or 255 for 8-bit gray-scale) with some low probability.

Gaussian noise is described by a gaussian distribution with a given mean and standard deviation.

Both types of noise can be created as separate images and then combined with the original image, using image arithmetic, to produce a corresponding noise image. The applet is displayed in Figure 6 and it accepts three parameters:

1. The percentage of Salt-Pepper noise to be produced.
2. The standard deviation value of the gaussian noise.
3. The mean value of the gaussian noise.

After these parameters have been set, the user may select the type of noise desired by selecting the desired button. After a noise image has been generated (like the middle one in Figure 6) it can be added to the original image by pressing the "Add Images" button.

In order to produce salt-pepper noise one requires the use of a random number generator. We used the one supplied with JAVA.

This process is repeated for each element of the image array and with a 256x256 image the time taken is several seconds as calls to random number generators are slow. This delay is multiplied

by a factor of three when a call to a gaussian-distributed random number generator is made. This is not a serious problem when the applet is running on its own.

4.5 Noise Reduction

We implemented four noise-reduction algorithms, which also evaluated standard local neighbor operations.

- Mean smoothing [5] by a 3x3 or 5x5 convolution kernel.
- A median [13] filter replaces a pixel by the median value of its neighborhood.
- Gaussian smoothing [16] implemented by using discrete 3x3 and 5x5 convolution kernels.
- The "Mean & Median" [9, 10] button: Pixel values in a 3x3 neighborhood excluding salt-pepper noise are averaged together.

The operation of this applet is straightforward. The user chooses the type of smoothing by pressing on the corresponding button. No example is shown for brevity.

4.6 Histograming

An intensity histogram is constructed by counting all the occurrences of each grey-scale value in an image. A graph is then plotted which on the horizontal axis contains the different pixel values, and on the vertical the number of times they occur (see middle image of Figure 7).

4.7 Thresholding

Thresholding is a straightforward and quite fast operation [1] to implement. All one has to do is retrieve the pixel value of an image, compare it to a preselected threshold value and replace the original pixel by either a one (or 255) or a zero depending on whether the threshold is greater than the pixel value or not. Figure 7 shows this operator.

4.8 Thinning

Thinning used a hit-and-miss transform [2] implemented as follows:

1. Sweep the image with one of the structuring elements shown in Figure 8.
2. If the 3x3 image pattern matches the structuring element (blanks denote don't-care points) then put a one on the corresponding location of the result image, otherwise put a zero.
3. Invert the resulting image and perform a binary AND of it with the initial image. This removes the points produced by the first structuring element.
4. Repeat processes 1-3 until all structuring elements have passed over the image (each element takes as input the output of the previous one).
5. Repeat processes 1-4 until the image doesn't change anymore.

Thinning takes a long time. In interpreted JAVA code a straightforward convolution of a 256x256 image with a 3x3 kernel takes approximately five seconds. As thinning requires 8 passes (*i.e.* one pass with each of the 8 different masks) for each iteration, and may require an unknown number of iterations, then this operator is too slow for interactive use and tutorial use. For example, five thinning passes would thus require about 200 seconds. Assuming a more efficient implementation gave an improvement factor of 5-10, this reduces the computation time to 20-40 seconds which is still too slow for tutorial use.

While this paper discusses only sample applets that illustrate different image processing schemas, we have already successfully implemented 11 operators (conservative smoothing, convolution, gaussian smoothing, histogramming, average smoothing, median smoothing, inverse, log transform, Roberts' cross, Sobel and thresholding) with work on about another 10 in progress.

5 Communicating Applets

The independent operators described in the previous section can be embedded and interact in an HTML document for display in a web page. This can be achieved by the following piece of HTML

code:

```
<APPLET CODE="AddNoise.class" WIDTH=800 HEIGHT=400>
<PARAM NAME=images  VALUE="images/holesquare2.gif">
</APPLET>
```

which adds the applets to an HTML page.

If the parameter named `images` is given a value (as in this case) then it is used, otherwise a default image is loaded.

The output image from each applet is linked to the next one down the page so that it can be further processed. The way this is achieved is by giving each applet the ability to communicate with the next one.

The procedure which JAVA employs to grant applets the ability to communicate with each-other is simple and works as follows:

- The recipient applet must be given a name using the NAME parameter in the relevant HTML document *e.g.*

  ```
  <APPLET CODE="AddNoise.class" WIDTH=800 HEIGHT=400 NAME="addnoise">
  ```

- Then, the sending applet must use the *getApplet()* method from the applet context with the name of the recipient applet as a parameter. This will return a reference to the recipient applet. One can use the reference applet as if it were an object and manipulate it accordingly. For example, if one wished to obtain a reference to the applet named "addnoise" belonging to the AddNoise class, the sending applet would need to declare an AddNoise variable and cast the applet obtained by *getApplet()* to that class:

  ```
  AddNoise addnoise_applet;

  addnoise_applet = (AddNoise)getAppletContext().getApplet("addnoise");
  ```

- All one needs to do now is define the functions that will implement the communication protocol.

  Each applet has been equipped with two extra functions. One of these functions is *send_image()*, which activates when the "Forward Results" button has been pressed, thus sending the whole

integer array of the final image produced by this applet to the next applet on the page. For example, consider Figure 9. The applets pictured are called "gamma" and "addnoise" respectively. After "Forward Results" has been pressed and the gamma operation repeated, the resulting image is passed on to the "addnoise" applet (Figure 9).

This is achieved by a call to *send_image()* which, in turn, calls the function *set_src_image()* of the "addnoise" applet in the usual object-oriented way:

```
addnoise_applet.set_src_image(img_1d);
```

This function's purpose is to set the source image of the receiving applet to be the image sent by the sending applet.

Now each applet can pass its image to the next applet. To make the communication procedure more general, a parameter has been added which can be set from the relevant HTML document. This parameter is called `receiver_applet` and is used to set the destination applet of the resulting image. If this parameter is missing, the default is to send the image to the next applet in the manner described above.

6 Performance

Image processing is generally a computationally expensive operation, and the usability of JAVA for educational purposes requires that students do not face unreasonable delays in the operator demonstrations. Further, performance in the demonstrations should be comparable to actual implementations of the operators so as to give students an understanding of the rate of performance as well as the capabilities of the operators.

This section makes a brief comparison of the time it takes for the JAVA applets to perform a particular image processing operation compared to the equivalent Visilog operation. Visilog [20] is a standard C-based commercial image processing package intended for educational use. Visilog's operators are not strictly equivalent as the underlying algorithms can differ (and Visilog timings also include overheads associated with data-structure management and screen displays)

but nevertheless should help provide a feel for the difference in performance. Our study showed that interpreted JAVA is about five to ten times slower than C, but compiled JAVA is comparable.

The machine on which the interpreted JAVA (and Visilog) tests were run was a lightly loaded Sparc 4 (110MHz). The timing was performed using one of JAVA's built in functions for retrieving the current time in milliseconds. We used Visilog's built-in timer (also in milliseconds). The applets were individually run through the *appletviewer* program and all the images used were 256x256 greyscale images. The machine on which compiled JAVA was tested on was an Intel Pentium running at 133MHz. The web-browser was Netscape version 3, running under MS-Windows 95, which includes a JIT compiler for JAVA.

One should bear in mind that running compiled JAVA on a different machine than the one Visilog was tested on is liable to create surprising results. A P133 processor is approximately 1.8 times faster than the Sparc 4 processor, when comparing pure processing power in terms of floating point and integer operations (see Table 2. These figures are taken from `http://www.maths.lth.se/bengtl/horna/spectable.html`). There are many other factors, however, which cannot be accounted for such as memory speed, caching strategy, *etc.*

6.1 Performance Comparison

Table 1 summarizes the timings of the operators presented here. In several cases, comparable operators did not exist, but we attempted to find operators that had a similar function. The implementation of the algorithm is the same between the two JAVA comparisons, but we do not have the Visilog source code so can only assume that the algorithm is similar.

We make some special notes here about the comparison:

- Rotation

  The JAVA rotation operator can be compared to Visilog's 'Nearby pixel' rotation although the algorithms are slightly different.

- Gamma correction

Visilog does not have a gamma correction transform for comparison.

- Thinning

  The JAVA program is approximately 50 times slower than Visilog. Some of this can be attributed to the inefficient algorithm implemented (even compiled JAVA is 4-5 times slower than Visilog).

Based on the average results in the table, we can conclude compiled JAVA is broadly comparable in speed with compiled C, but interpreted JAVA is about 5-20 times slower for most operations and thus is too slow for many tutorial applications.

7 Discussion

The purpose of this project was to investigate the suitability, or otherwise, of JAVA teaching image processing. The answer to this question became reasonably apparent during the early stages of the project, and we have concluded that JAVA is sufficiently fast and flexible for image processing both for creating proper applications (such as Visilog or XV) as well as for providing interactive teaching material for students via the Internet.

One of our experiences with teaching image processing is that a suitable set of example images is not easy to derive, and textbooks do not have the space to allow an exhaustive presentation. On the other hand, when the example images can be derived on the spot, the learner has a much broader range of experiences. The benefits of the learner actually creating the examples is clear, in that the percentage of material retained in "active learning" (*i.e.* learning while doing something), is much higher than in "passive learning" (*i.e.* where the learner only reads or hears the material). The ability to embed JAVA more-or-less seamlessly into HTML documents also makes the teaching materials intellectually pleasing and easy use, whereas skipping between several different media or presentation windows can cause one to lose one's place or interest (or increase the intellectual "energy barrier" to actually doing the exploration).

The teaching resources described in this article were used as part of a more extensive course

on machine vision. Only two of the lectures in that course discussed the sorts of image processing operators used here. This is typical of the experience other lecturers have of the various forms of packaged teaching resources (*e.g.* videos and books) — the contents of the resource package intersects with the topics that they wish to teach, but does not contain all the desired material.

While it is possible to develop more-or-less complete on-line courses, our experience of student opinion is that the students prefer to have some lectures. Thus, because of both their preferences and the desire of teaching staff to select portions of the teaching materials, we have been developing these materials in the form of a "resource package", rather than as a stand-alone package.

To help the materials to become part of the students' repertoire of actually used learning resources, we developed an exercise that was used in on-line small-group "tutorial" sessions. The drill sheets assigned a few simple exercises designed to familiarize the students with the use of the hypertext and JAVA style of the resources, and also the overall layout of the whole HIPR package.

Overall, the materials were integrated into an existing traditional course (lectures, small groups, lab exercises) in a way that did not change much the way that the course was run, nor the way that we interacted with the students. (Whether this is the best approach is subject to much debate, of course). None-the-less, the introduction of the package did increase the amount of *active* contact that the students have with the material.

We did no formal assessment of the benefits of the JAVA enhancements, although, in informal student feedback, the students said that they liked exploring the operators and found the single focussed interface less distracting than the many options of Visilog. They also commented that seeing how the operator performance varied with different parameter values was helpful. They made no specific comments about the supporting text, but made many comments about how the online access was a great advantage as compared to having to locate/buy textbooks.

Figure 9 shows the embedding of text and interactive demonstrations in rudimentary form (see `http://vision.dai.ed.ac.uk/kotsDEMO/demo.html`), whereas a proper application of the approach would have more tutorial material and discussion of how to interpret the results (see

http://www.dai.ed.ac.uk/daidb/staff/personal_pages/rbf/HIPR/hiprdemo/html/newthr.htm
and
http://www.dai.ed.ac.uk/daidb/staff/personal_pages/rbf/HIPR/hiprdemo/html/newgsm.htm).
Further development of the online materials is continuing. Appendices B.1 - B.3 shows the HTML and JAVA for the complete threshold applet given at the URL above.

One of the major reasons for JAVA's popularity arises from the fact that JAVA comes with many built-in methods specifically for image manipulation, such as for image loading and pixel retrieval. Implementing a function to load a gif or jpg image in C would be a tedious and difficult task. JAVA gives this and much more for free. Furthermore, it is easy to learn and use, debugging is usually straight-forward and is platform independent (but is subject to occasional bugs as new JAVA compilers and interpreters are being developed).

The development time for the applets presented here was about 3 person-months. However, factored into this time was considerable learning about the facilities and use of JAVA, and the development of several different operator templates (*e.g.* single point single input image operators, single point two input image operators, convolution), including display and file interfaces. The project also went through several iterations of display and inter-applet communication protocols.

With this advance work done, it is possible to develop and integrate new operators in something like 2-4 hours provided that the display and algorithm structure is nearly identical to one of the previously developed operators. When new algorithms need to be substituted into an existing applet framework, then development time is still reduced.

The question of performance which is raised in the case of interpreted JAVA is eliminated by compiled JAVA as we have seen in the previous section. JAVA compilers and JIT compilers already exist and new products are under rapid development.

The only development difficulties encountered were due mainly to the heavy use of threads during the initial approach to the development of the applets. It turned out that the WWW browser that the applets were being tested on (Netscape version 2.02 for Unix) could easily become overloaded which in turn caused it to halt for unreasonably long periods of time. These difficulties

were eliminated by restricting the use of threads and by keeping the computation within them down to a minimum.

Based on the general ImageProducer and ImageConsumer paradigm, we showed that a wide variety of classes of image processing operators could be implemented in JAVA, that their execution time is reasonable when JAVA is compiled, and the Applets can be embedded in HTML text for use in teaching materials.

We are now using this methodology to add interactive exploration to the HIPR package [2]. (The non-JAVA version of HIPR is currently being sold by John Wiley and Sons.) As well as the individual operators as discussed here and shown in the online examples, we are also investigating a prototype Khoros-like [8] operator pull-down workbench (as an extension of [15]) that allows sequences of operators to be connected together and explored as a group. Further in the future, we are investigating how to give real-time commentary and feedback to students, based on the choice of operators and parameters used when attempting to solve problems set on specified test images.

Enhancing the current JAVA operators to give "critical" feedback on the parameters used to solve a problem is a future possibility. We believe that doing this well would require substantial work – in part to develop the repertoire of responses in relation to a fixed set of tutorial images, and in part to integrate a process capable of developing a model of the student's intellectual development, so that appropriate feedback can be given. These developments could perhaps be done in two independent stages, but then some sort of "off" switch would be necessary to disable the repetition of elementary feedback. Adding both levels of capability are clearly desirable features and are probably feasible at least at an elementary level. However, the cost of this additional work is likely to be close to the rule of thumb for the development of computer-based teaching materials: one person-year of development for one hour of high-quality meaningful student interaction.

References

[1] M. P. Ekstrom (Ed). *Digital Image Processing Techniques.* Academic Press, Inc., 1984.

[2] R. B. Fisher, S. Perkins, A. Walker and E. Wolfart. *HIPR: Hypermedia Image Processing Reference*, CDrom published by John Wiley and Sons, Chichester, 1996.

[3] A. S. Glassner (Ed). *Graphics Gems.* Academic Press Professional, 1988.

[4] Heriot Watt University, Napier University and the University of Edinburgh. *Marble Interactive Vision.* http://www.marble.ac.uk/marble/, 1996.

[5] V. Hlavac, M. Sonka and R. Boyle. *Image Processing, Analysis and Machine Vision.* Chapman & Hall, Cambridge University Press, 1993.

[6] B. Jähne. *Digital Image Processing.* Second Edition, Springer-Verlag, 1993.

[7] R. Jordan and R. Lotufo, *Interactive Digital Image Processing Course on the World Wide Web.* IEEE International Conference on Image Processing, ICIP-96 pp 433-436, Volume II, Sept. 16-19, 1996. See `http://tularosa.eece.unm.edu/dipcourse/` for the materials.

[8] Khoros is a sophisticated visual programming and scientific software development environment, sold by Khoral Research (Khoral Research Inc, 6001 Indian School Rd, Ne, Suite 200, Albuquerque, NM 87110, USA; Tel:(505)837-6500; Fax:(505)881-3842).

[9] K. Koryllos and R. Fisher. "Interactive textbooks: embedding image processing operator demonstrations in text". Online proceedings of IEEE Computer Society Workshop on Undergrad Education & Image Computation (Ed. Kevin Bowyer), Puerto Rico, June 20, 1997. See `http://marathon.csee.usf.edu/education.html` for the online proceedings.

[10] K. Koryllos. "On-line image processing operator demonstrations in Java". Master's thesis, Department of Artificial Intelligence, Edinburgh University, 1996.

[11] L. Lemay and C. L. Perkins. *Teach Yourself Java in 21 Days.* Sams.net Publishing, 1996.

[12] R. Lotufo and R. Jordan, "Hands-on Digital Image Processing". IEEE Frontiers in Education - 26th Annual Conference, FIE-96 pp 1199-1202, Volume 3, Nov. 6-9, 1996. See `http://tularosa.eece.unm.edu/dipcourse/` for the materials.

[13] A. Low. *Introductory Computer Vision and Image Processing.* McGraw-Hill, 1991.

[14] S. Moscariello, R. Kasturi, O. Camps "Image processing and computer vision instruction using Java". Online proceedings of IEEE Computer Society Workshop on Undergrad Education & Image Computation (Ed. Kevin Bowyer), Puerto Rico, June 20, 1997. See `http://marathon.csee.usf.edu/education.html` for the online proceedings.

[15] A. Murdoch. "An image processing workbench". Batchelor's degree dissertation, Department of Artificial Intelligence, Univerity of Edinburgh, 1996.

[16] W. Niblack. *An Introduction to Digital Image Processing.* Prentice-Hall International, 1986.

[17] J. C. Russ. *The Image Processing Handbook.* CRC Press Inc, 1995.

[18] The Java Development Team. *The Java Tutorial and The Java API Documentation.* Sun Microsystems, java.sun.com, 1.0 edition, 1996.

[19] D. Unwin and M. Langford. "The life and death of LIPS: some lessons from the design and use of courseware for teaching digital image processing". *Digest of IEE*, (190):1.1–1.4, 1993.

[20] Visilog is a GUI (Graphical User Interface) based image processing package produced by Noesis (Noesis, Immeuble Nungesser, 13 Avenue Morane Saulnier, 78140 Velizy, France; Tel: (33-1)34-65-08-95.

A The Generic Image Processing Applet

In each of the applets described in Section 4, certain important steps are common. While one does not usually include code in a paper, here it is useful to be precise.

Step 1

The first step is to obtain the image to be processed. This is achieved using

```
Image_src = getImage(getCodeBase(), image_name);
```

where `Image_src` is an *Image* object, *getCodeBase()* obtains the path where the compiled program resides and `image_name` is a *String* object which contains the name of the image which can either be obtained from an HTML document or by default (using `getParameter`)

```
String image_name = getParameter("image");
if (image_name == null) image_name="images/simon.gif";
```

The method *getParameter()* obtains the value of the `image` parameter in the HTML document that holds the applet. If no such parameter exists then a default value compiled with the applet is used. The combined code above results (if no HTML parameter is found) in the image `simon.gif` to be loaded. This image must reside in the sub-directory `images` of the directory holding the executable applet code.

Step 2

Next, the image must be tied to a canvas so that its size can be obtained. *ImageCanvas* is a subclass of *Canvas* (written by Andrew Fitzgibbon). An instance of the *ImagesCanvas* class is first created:

```
ImageCanvas src_canvas = new ImageCanvas(src);
```

and to retrieve the image width and height the following piece of code is used:

```
int i_w = src_canvas.getImageWidth();
int i_h = src_canvas.getImageHeight();
```

Step 3

The following piece of code produces a one-dimensional array of pixels (stored in `src_1d`) from the image contained in `src`.

```
PixelGrabber pg1 = new PixelGrabber(src,0,0,i_w,i_h,src_1d,0,i_w);
try {
  pg1.grabPixels();
} catch (InterruptedException e) {
  System.err.println("InterruptedException!");
  return;
}
if (pg1.status() & ImageObserver.ABORT) return;
```

`src_1d` will contain the pixel values required provided that the operation executes smoothly.

The method *grabPixels()* initiates the pixel acquisition process.

Step 4

The pixel values can now be extracted and operated upon from the array `src_1d`.

Step 5

The code

```
dest = createImage(new MemoryImageSource(i_w,i_h,dest_1d,0,i_w));
```

creates an *Image* object from a one-dimensional array of pixels of a particular size. This image can be, in turn, tied to an *ImageCanvas* and displayed by:

```
grid.add(dest_canvas = new ImageCanvas(dest));
```

B HTML and JAVA for Threshold Applet

This appendix shows the complete HTML and JAVA code used for the applet demonstrated on the extended HIPR worksheet at:

`http://www.dai.ed.ac.uk/daidb/staff/personal_pages/rbf/HIPR/hiprdemo/html/newthr.htm`

B.1 HTML Added to HIPR Pages

```
<!-- applet to load image -->
<p><APPLET CODE="GetImage.class" CODEBASE="javacode/"
WIDTH=500 HEIGHT=100>
<PARAM NAME=id VALUE="threshload">
</APPLET>

<!-- applet to threshold image -->
<p><APPLET CODE="Thresh.class" CODEBASE="javacode/"
WIDTH=810 HEIGHT=300>
<PARAM NAME=id VALUE="thresh">
</APPLET>
```

B.2 JAVA Source to Load Image

```
//  File: GetImage.java
//  Author: Konstantinos Koryllos
//  Date: September 1996
//  Purpose: To load an image via a URL and send it to a specific
//           applet.
//  Bugs: Must have. Must use Nestcape 3 for error during URL parsing.
//

import java.applet.*;
import java.awt.*;
import java.awt.image.*;
import java.io.*;
import java.net.*;
import java.util.*;
import java.lang.*;

import ImageCanvas;

public class GetImage extends Applet {

  Panel pan1 = new Panel();
  Choice applet_menu = new Choice();
  GridBagLayout gridbag = new GridBagLayout();
  GridBagConstraints c = new GridBagConstraints();

  // define static panel
  Button load_image = new Button("Load Image");
  TextField input = new TextField(70);
  String image_url = "brg1.gif";        // default loadable image
  Label status = new Label("Status:");
  TextField error_mesgs  = new TextField(70);
  URL theURL;                           // URL of image to load

  public void destroy() {}

  public void stop() { destroy(); }
```

```
public void start() {}

// set up display panel
public void init() {
  input.setText(image_url);
  error_mesgs.setEditable(false);
  this.setLayout(new FlowLayout());
  pan1.setLayout(gridbag);
  pan1.setBackground(Color.white);
  pan1_layout_components();           // place components
  this.add(pan1);
  this.setBackground(Color.white);
}

// specify panel component layout
private void pan1_layout_components() {

  c.fill = GridBagConstraints.BOTH;   // load image button
  gridbag.setConstraints(load_image,c);
  pan1.add(load_image);

  c.gridwidth = GridBagConstraints.REMAINDER; // input box
  gridbag.setConstraints(input,c);
  pan1.add(input);

  c.anchor = GridBagConstraints.CENTER;   // status message label
  c.fill = GridBagConstraints.BOTH;
  c.gridwidth = 1;
  gridbag.setConstraints(status,c);
  pan1.add(status);

  c.anchor = GridBagConstraints.WEST;     // status message box
  c.fill = GridBagConstraints.BOTH;
  c.gridwidth = GridBagConstraints.REMAINDER;
  gridbag.setConstraints(error_mesgs,c);
  pan1.add(error_mesgs);
}

// panel event handler
public boolean action(Event evt, Object arg) {

  // load image button pushed
  if (evt.target == load_image) {

    image_url = input.getText();      // pull text out of window
                                      // name of image file or full URL
    // see if a URL
    try {
      theURL = new URL(image_url);    // parse supplied URL
      error_mesgs.setText("Image "+image_url+" to be used");
    } catch (MalformedURLException e) {

      // if a file rather than a URL, then add document base and try again
```

```
      // If still not a URL and not a file then report error
      try {
              theURL = new URL(getDocumentBase(),"../images/"+image_url);
              error_mesgs.setText("Image "+getDocumentBase()+
                      ", "+"../images/"+image_url+" extended to use");
        } catch (MalformedURLException e4) {

         error_mesgs.setText("Error: Not a valid URL");
      }
    }

    // spool off image viewer - ie pop up an XV
    getAppletContext().showDocument(theURL);

    // search thru applet context to find destination process
    // assume only one other applet and assume it's not the one
    // whose name ends with the applet parameter id="...load"
    // (which is assumed to be this applet). Schema could be improved.
    Enumeration enum = getAppletContext().getApplets();
    Receiver receiver = (Receiver) null;
    while (enum.hasMoreElements()) {
        Applet applet = (Applet) (enum.nextElement());
        String name = applet.getParameter("id");
        if (name != null && !name.endsWith("load")) {

            receiver = (Receiver) applet;
        }
      }

    // load and send image to receiver
    try {

      // get file name
      String filename = theURL.getFile();

      // make sure entered file has correct suffix
      if (!filename.endsWith("jpg") && !filename.endsWith("gif"))
      {
            throw (new NullPointerException());
      }
      Image image = this.getImage(theURL);  // link to image
      apply_send_image(receiver, image);    // send image to receiver applet
    }
    catch (NullPointerException e)
    {
        error_mesgs.setText("Invalid image type or other access error");
    }
  }

  return true;
}

// load image & send to processor applet
private void apply_send_image(Receiver applet, Image image){
```

```
    // set up memory for use
    ImageCanvas image_canvas = new ImageCanvas(image);
    int i_w = image_canvas.getImageWidth();
    int i_h = image_canvas.getImageHeight();
    int[] src_1d = new int[i_w*i_h];

    // load in pixels
    PixelGrabber pg1 = new PixelGrabber(image,0,0,i_w,i_h,src_1d,0,i_w);
    try {
      pg1.grabPixels();
    } catch (InterruptedException e) {
      error_mesgs.setText("InterruptedException at apply_send_image");
      return;
    }
    if ((pg1.status() & ImageObserver.ABORT) != 0) {
      error_mesgs.setText("Trouble at apply_send_image");
      return;
    }

    // send image to processor applet
    applet.set_src_image(src_1d, i_w, i_h);
  }

}
```

B.3 JAVA Source to Threshold Image

```
// File: Thresh.java
// Author: Konstantinos Koryllos
// Date: August-September 1996
// Purpose: Thresholding & thinning
// Bugs: Couldn't possibly say...

import java.applet.*;
import java.awt.*;
import java.awt.image.*;
import java.awt.Color;
import java.net.*;

import ImageCanvas;
import TPrint;

public class Thresh extends Applet implements Runnable, Receiver {

  // structures for input/output images
  Image src, dest;
  ImageCanvas src_canvas, dest_canvas;
  int i_w=0, i_h=0;
  int[] src_1d, dest_1d;

  // structures for histogram
  Image hist;
  ImageCanvas hist_canvas;
```

```
int[] hist_1d;
private final int grey_scales = 256; //num of grey-scale values
int hist_w = 0; //width of histogram. Either 256 or 3*256....

// histogram label.
Label hist_range = new Label("0          64"+
                             "          128"+
                             "          192"+
                             "          256");

Panel grid  = new Panel(); //images
Panel panl  = new Panel(); //textfield and label

// define static panel items
//threshold
TextField threshold  = new TextField(10);
Label instructions = new Label("Enter threshold value: ");

//for calculating execution time
long time_msec;
TextField time_taken = new TextField(10);
Label execution_time = new Label("Execution time: ");

//Flip pixels 0->255, 255->0
Button flip_pixels = new Button("Flip Pixels");

//Component.reshape() parameters
//to place src images, histogram and destination image.
int src_x, src_y, src_w, src_h;
int his_x, his_y, his_w, his_h;
int dest_x, dest_y, dest_w, dest_h;

// set up fixed display items
public void init() {
  this.setLayout(new BorderLayout());

  //dealing with source image
  grid.setLayout(null);
  this.add("Center", grid);
  //GUI stuff
  panl.setLayout(new GridLayout(0,5));
  panl.add(instructions);
  panl.add(threshold);
  panl.add(flip_pixels);
  panl.add(execution_time);
  panl.add(time_taken);
  this.add("North",panl);
  time_taken.setEditable(false);
  panl.setBackground(Color.white);

  //Get image name
  String image_name = getParameter("image");
  if (image_name == null) image_name="../../images/cln1.gif";
  src = this.getImage(this.getCodeBase(), image_name);
```

```
  //set src_canvas
  src_canvas = new ImageCanvas(src);

  //Get size of image and make 1d_arrays
  set_image_dimensions();
  src_1d  = new int[i_w*i_h];

  PixelGrabber pg1 = new PixelGrabber(src,0,0,i_w,i_h,src_1d,0,i_w);
  try {
    pg1.grabPixels();
  } catch (InterruptedException e) {
    System.err.println("InterruptedException at init() pg1.grabPixels()");
    return;
  }
  if ((pg1.status() & ImageObserver.ABORT) != 0) {
    System.err.println("Trouble at init() pg1.grabPixels()");
    return;
  }

  //determine the width of the histogram
  hist_w = grey_scales;

  //create blank destination & histogram.
  create_dest_hist();

  //add images in GUI(source and two blanks: histogram, destination)
  gui_add_images();

  //Component.reshape() Parameters
  set_reshape_params();

  //Absolut positioning... Someone fix this!
  reshape_canvases();

  //Make the histogram of the source image
  make_histogram();

  //add histogram label.
  grid.add(hist_range);
  hist_range.reshape(13+i_w,18+i_h,13+i_w,30);
}

// set image sizes
private void set_reshape_params() {
  src_x = 10;
  src_y = 10;
  src_w = i_w + 10;
  src_h = i_h + 10;
  his_x = 18 + i_w;
  his_y = 10;
  his_w = i_w + 10;
  his_h = i_h + 10;
  dest_x = 26 + i_w + hist_w;
```

```
  dest_y = 10;
  dest_w = i_w +10;
  dest_h = i_h +10;
}

private void reshape_canvases() {
  src_canvas.reshape(src_x,src_y,src_w,src_h);
  hist_canvas.reshape(his_x,his_x,his_w,his_h);
  dest_canvas.reshape(dest_x,dest_y,dest_w,dest_h);
}

// draw histogram
private void make_histogram() {

  double[] hist_values = new double[grey_scales];
  double increment = 1.0/256.0; //histogram "normalisation"

  // process all pixels into histogram
  int blue;
  for(int i=0;i<src_1d.length;i++){
    blue = src_1d[i] & 0x000000ff;
    hist_values[blue] += increment;
  }

  // find maximum histogram peak height
  double max_value = 0.0; //the grayscale value appearing more often.
  for(int i=0;i<grey_scales;i++)
    max_value = (hist_values[i]>max_value)?hist_values[i]:max_value;

  // scale display to spread over 256 vertical values of histogram
  int scale_factor = (int) Math.floor(grey_scales / max_value);
  for(int i=0;i<hist_values.length;i++) hist_values[i] *= scale_factor;

  // draw histogram display
  int l = hist_1d.length;
  for(int i=0;i<hist_values.length;i++){
    int n = 1;
    while (hist_values[i] > 0){
      hist_1d[(l-grey_scales*n)+i] = 0xffff0000;
      hist_values[i] -= 1;
      n++;
    }
  }

  // display new histogram
  grid.remove(hist_canvas);
  hist = createImage(new MemoryImageSource(grey_scales,i_h,hist_1d,0,grey_scales));
  grid.add(hist_canvas = new ImageCanvas(hist));
  hist_canvas.reshape(his_x,his_y,his_w,his_h);
  hist_1d = new int[grey_scales*i_h];

}

private void set_image_dimensions() {
```

```
  i_w = src_canvas.getImageWidth();
  i_h = src_canvas.getImageHeight();
}

private void create_dest_hist() {
  hist_1d = new int[hist_w*i_h];
  dest_1d = new int[i_w*i_h];
  dest = createImage(new MemoryImageSource(i_w,i_h,dest_1d,0,i_w));
  hist = createImage(new MemoryImageSource(hist_w,i_h,hist_1d,0,hist_w));
}

private void gui_add_images() {
  grid.add(src_canvas = new ImageCanvas(src));
  grid.add(dest_canvas = new ImageCanvas(dest));
  grid.add(hist_canvas = new ImageCanvas(hist));
}

public void start() {}
public void stop() {}
public void run() {}

// display event handler
public boolean action(Event evt, Object arg) {

  time_msec = System.currentTimeMillis();

  // threshold value entered
  if (evt.target == threshold) {
    try {
      int thresh=0;
      thresh = Integer.parseInt(threshold.getText());
      if ((thresh < 0) || (thresh > 255)) {
        throw new NumberFormatException();
      }
      threshold(thresh);         // do threshold
    } catch (NumberFormatException e) {
      threshold.setText("");  // reset bad threshold value field
      return true;
    }
  }

  //invert thresholded image
  else if (evt.target == flip_pixels) {
    flip_the_pixels();
  }

  // report processing time
  time_msec = System.currentTimeMillis() - time_msec;
  time_taken.setText(new Long(time_msec).toString()+" msecs");

  // replace result image with new result
  grid.remove(dest_canvas);
  dest = createImage(new MemoryImageSource(i_w,i_h,dest_1d,0,i_w));
  grid.add(dest_canvas = new ImageCanvas(dest));
```

```
    dest_canvas.reshape(dest_x,dest_y,dest_w,dest_h);

    return true;
  }

  // do threshold. Assume greyscale image and threshold using only
  // the blue channel value
  private void threshold(int thresh) {
    int blue;
    for(int i=0;i<src_1d.length;i++){
      blue = src_1d[i] & 0x000000ff;
      dest_1d[i] = (blue>=thresh)?0xffffffff:0xff000000;
    }
  }

  //swaps black and white values
  public void flip_the_pixels(){
    for(int i=0;i<dest_1d.length;i++){
      dest_1d[i] = (dest_1d[i]==0xff000000)?0xffffffff:0xff000000;
    }
  }

  //called by image loading applet
  public void set_src_image(int[] input_img, int w, int h) {
    //reset image size
    i_w = w;
    i_h = h;
    src_1d = new int[i_w*i_h];

    System.arraycopy(input_img,0,src_1d,0,input_img.length);
    grid.remove(src_canvas);
    grid.remove(hist_canvas);
    grid.remove(dest_canvas);
    src = createImage(new MemoryImageSource(i_w,i_h,src_1d,0,i_w));

    hist_w = grey_scales;
    create_dest_hist();
    gui_add_images();
    set_reshape_params();
    reshape_canvases();
    make_histogram();
  }

}
```

Exercise 1: This image is of a white square against a black background. Try to find a threshold which makes the square completely white and the background completely black.

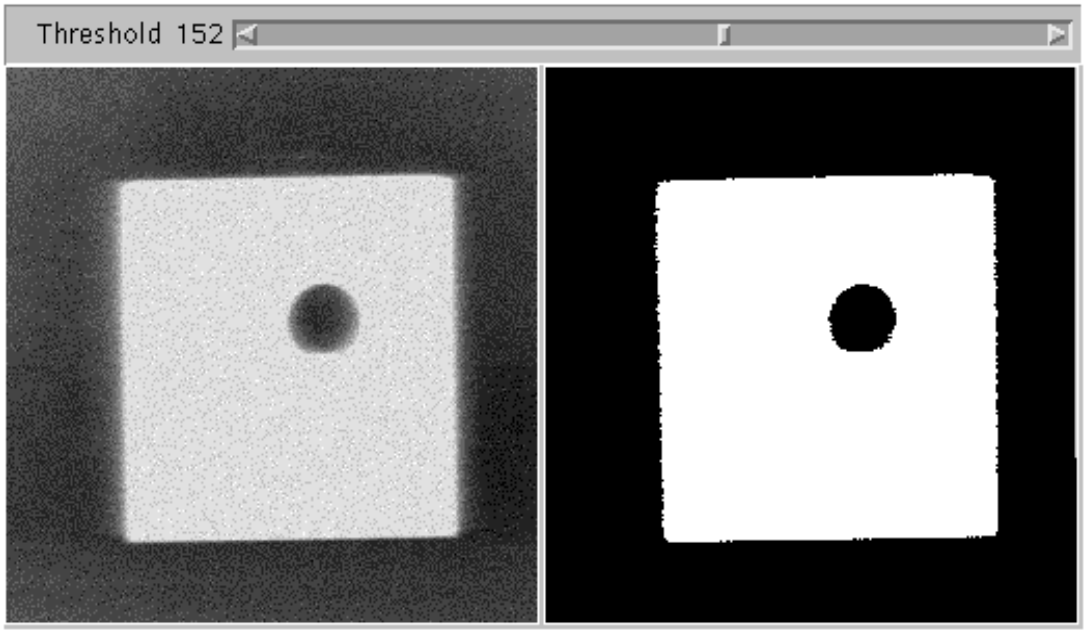

Figure 1: Prototype threshold applet

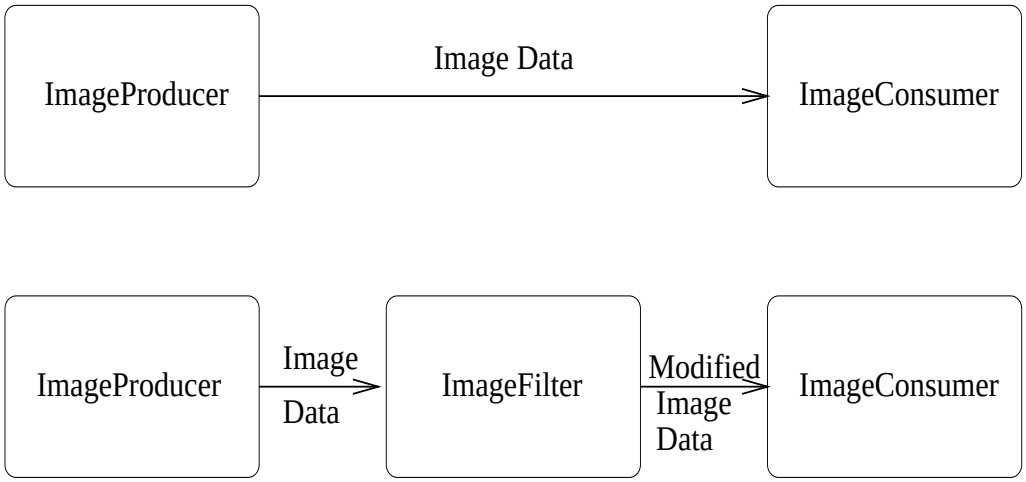

Figure 2: Behind the scenes

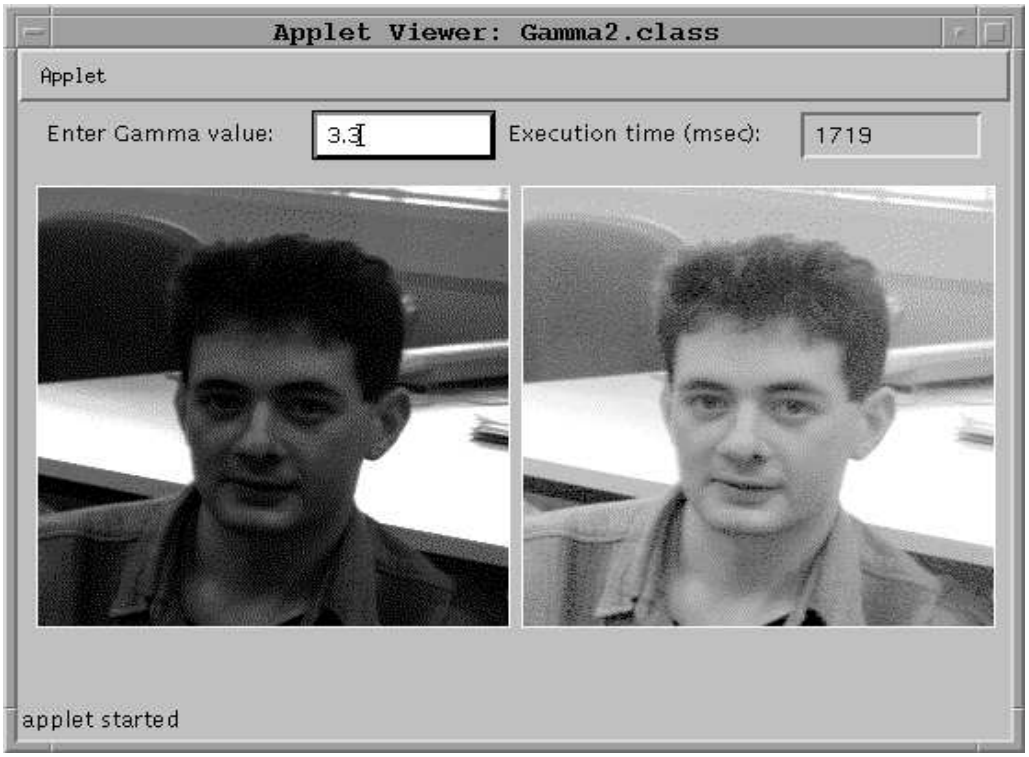

Figure 3: The gamma correction applet

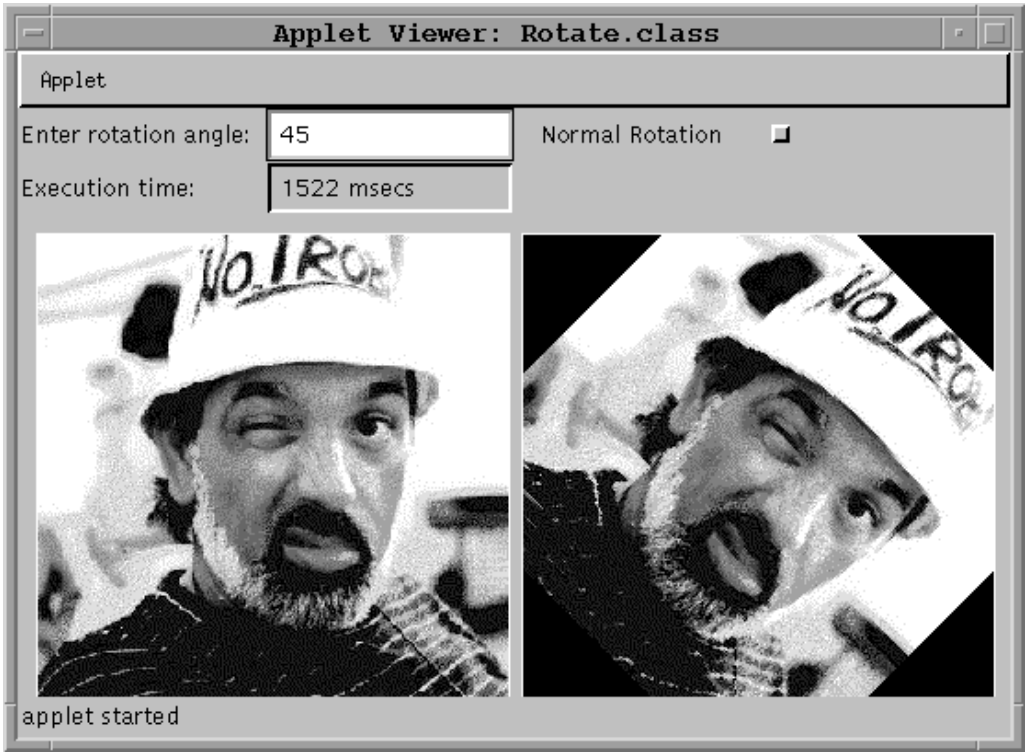

Figure 4: The rotation applet

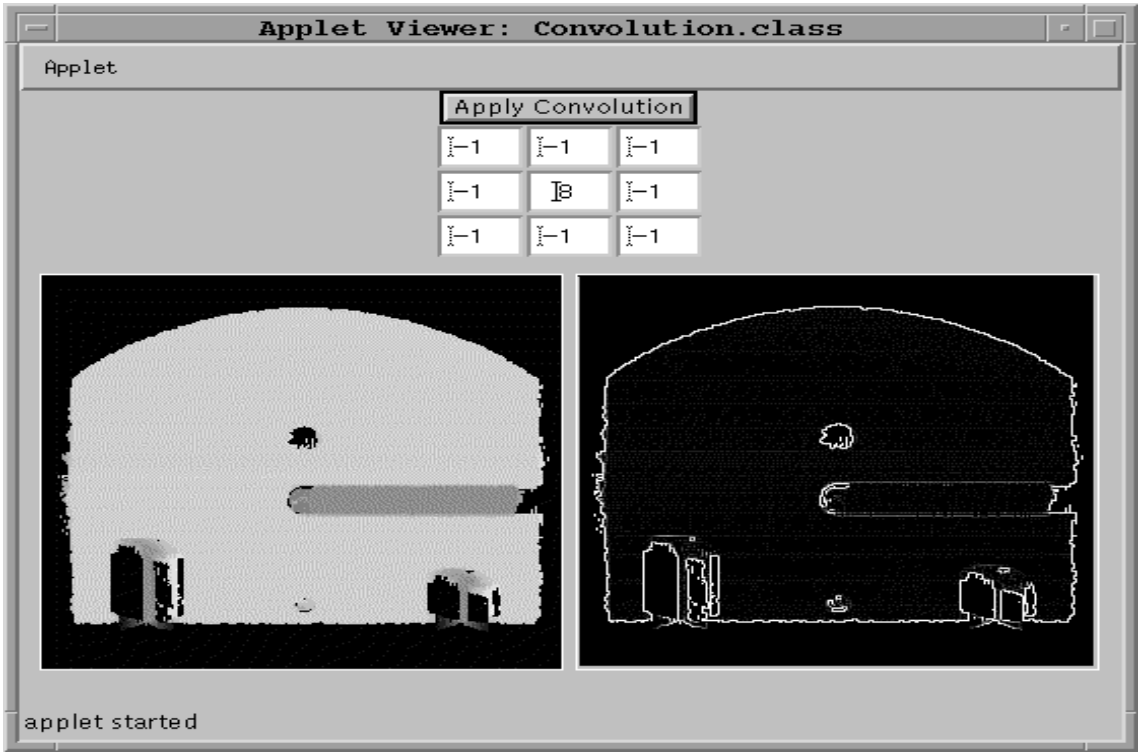

Figure 5: The convolution applet

Figure 6: The noise generating applet

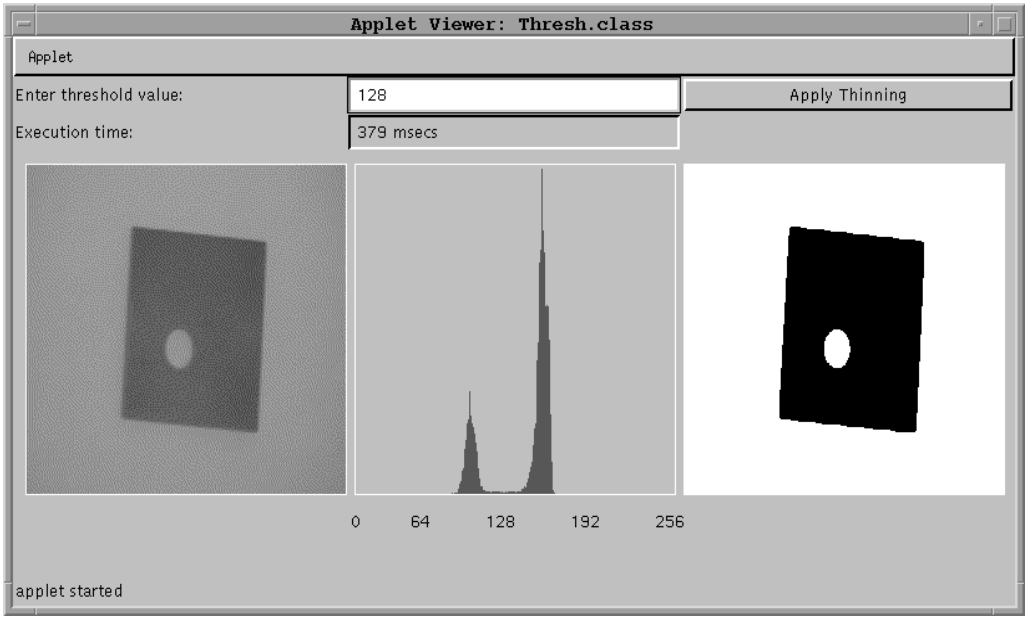

Figure 7: The thresholding applet

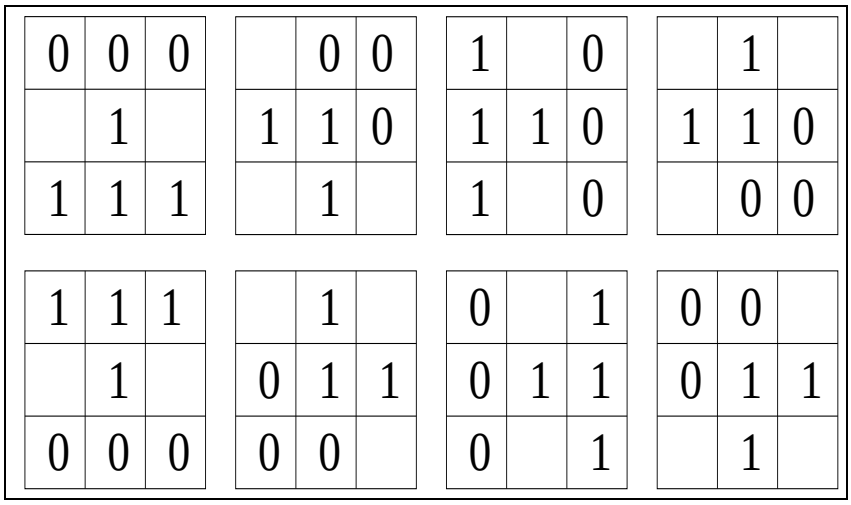

Figure 8: Thinning structuring elements

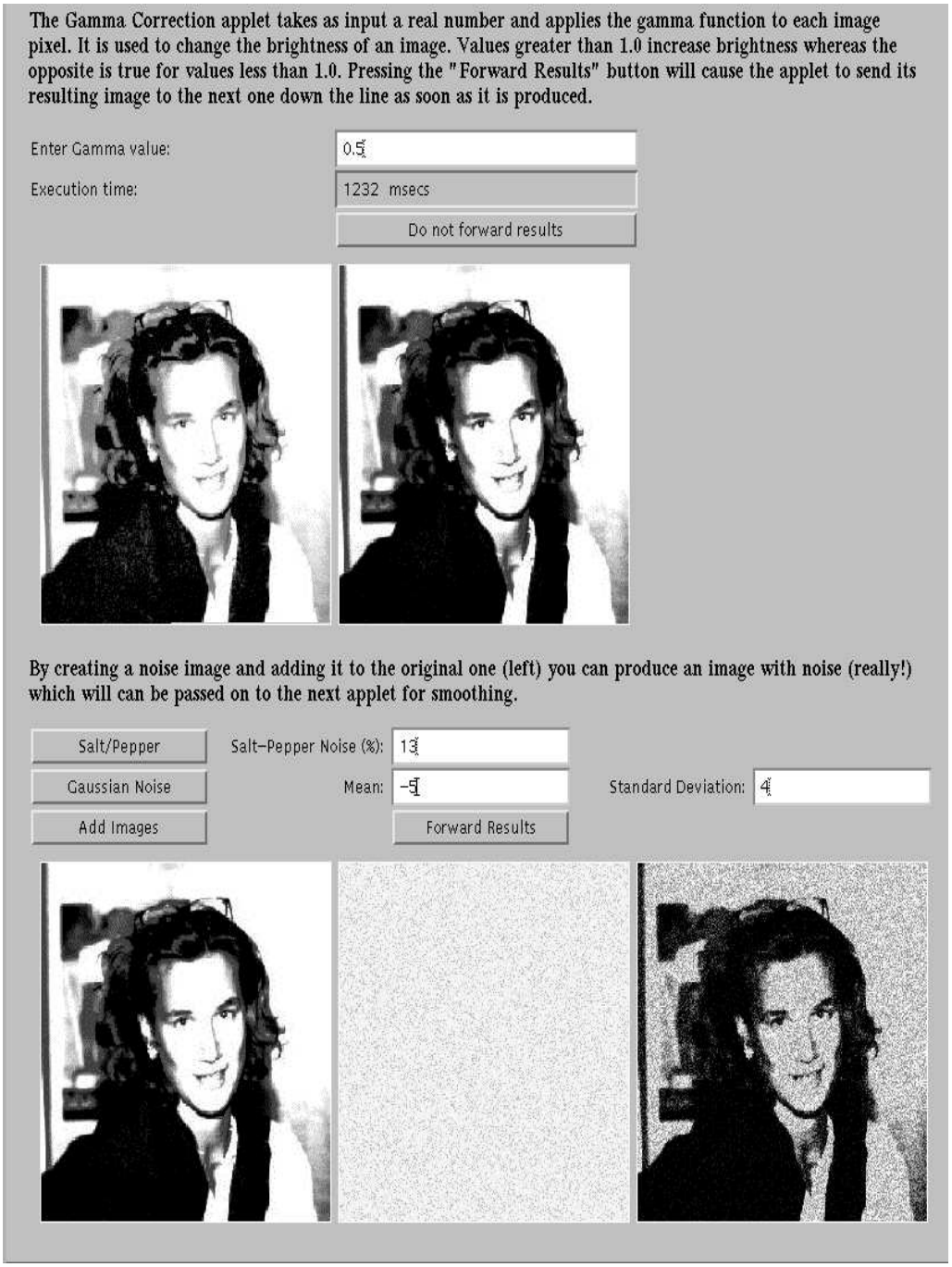

Figure 9: Communicating applets

Table 1: Comparison of execution times in seconds

Operation	Java	Visilog(C)	Java with JIT
Normal rotation	1.1-1.3	—	1.2-1.4
Shear rotation	4.0-5.2	—	0.5-0.6
Nearby pixel rotation	—	0.8-1.0	—
Four neighbor rotation	—	3.0-3.2	—
Gaussian smoothing (3x3)	5.0-6.0	0.8-1.0	0.3-0.4
Mean smoothing (3x3)	4.0-5.0	0.3-0.4	0.3-0.4
Mean smoothing (5x5)	11.0-12.0	0.8-0.9	0.7-0.8
Median smoothing	6.0-7.0	0.4-0.5	0.3-0.4
Median and mean	5.0-8.0	—	0.3-0.4
Salt-Pepper noise generation	0.2-0.3	—	instant
Gaussian noise generation	0.2-0.3	—	instant
Gamma correction	1.0-2.0	—	0.7-0.8
Thresholding	0.3-0.4	0.2-0.3	instant
One thinning iteration	21-23	0.3-0.5	1.7-1.9
Convolution (3x3 Laplacian)	4.5-5.5	0.8-1.0	0.4-0.5
Image addition	0.2-0.3	0.1-0.2	instant

Processor	**Clockspeed**	**SPECint92**	**SPECfp92**
microSparc-II	110MHz	78.6	65.3
Pentium 133	133MHz	147.5	109.6

Table 2: Sparc *vs.* Pentium