On-line Image Processing Operator Demonstrations in Java

Konstantinos Koryllos

MSc Information Technology: Knowledge Based Systems
Department of Artificial Intelligence
University of Edinburgh
1996

Abstract

A new tool for developing interactive software has emerged. It is called Java[1] and is an object-oriented language developed by Sun Micro-systems. Now it is possible to have Java-enhanced Web pages to interactively teach students new material.

This project is aimed at exploring the facilities provided by Java for image processing and implementing a few representative operators from different classes which will operate within a Web-browser environment.

[1]Registered trademark of Sun Microsystems, Inc.

Acknowledgements

Firstly, I am obliged to my supervisor, Robert B. Fisher, for keeping me in track and for making this project interesting, without significantly increasing the amount of grey hair in my head.
Secondly, I would like to thank Pavlos Papageorgiou for proofreading this document as well as for his expert advice on a diversity of topics.
Lastly, credit is due to the SAA[2] for their funding, to my parents who still love me, and to my University for having me around longer than I expected.

[2]Students Award Agency (not South African Airlines), Grant number 7582633

Contents

List of Figures

Chapter 1

Introduction

We are probably quite fortunate to be living in an era where we can communicate with each other via the World Wide Web (WWW) and exchange a very broad variety of information with most of the rest of the world. The ability to be able in seconds to exploit world-wide resources is probably the best justification for the existence of the WWW.

In the Computer Science and Artificial Intelligence fields one of the major exchanges of information is achieved through software programs. It is more than probable that when you are developing some piece of software you will be able to draw upon previous work and source code on the subject. This is where we hit upon a slight snag. With a few tens of widely used languages one would have to be extremely computer-literate to be able make full use of the source code one wishes to elaborate upon.

At this point, many hope, that Java[1] will come to the rescue by allowing all the programmers in the world to communicate using the same language. This is quite an ambitious and powerful vision but given the attention the Internet is receiving at the moment it has reasonable potential for success.

[1]The famous new Object-Oriented Language developed by Sun Microsystems.

1.1 The Internet in Education

In the present day and age the personal computer has become a standard household apparatus and educational software packages[2] are amongst the most popular ones. People are usually far better at remembering interactive rather than static material. Interaction with knowledge is always more fruitful than mere textbook memorisation so children and students in particular can benefit greatly from the new technology.

In the past few years many Universities and Colleges have produced their own WWW pages with several 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. On the hot-list now are the interactive stand-alone programs that occupy a part of a WWW page with educational content. The favourite set-up seems to be composed of some length of theory about a particular subject and then a real hands on (mouses on?) application, for which Java is ideal. This forms the basis of an “interactive textbook”.

A very representative example of such an application can be found in the WWW pages of the Physics Department of the University of Syracuse in New York and is illustrated in Figure 1.1. The purpose of this on-line demonstration is to give the opportunity to students to have a hands-on experience with the way the vector cross-product behaves as the magnitude of the vectors and the angle between them varies. After a couple of minutes of interaction with the program one cannot help wanting to see more such demonstrations.

1.2 Image Processing

Digital image processing is a fairly new field which made its appearance in the 1960's together with third generation computers. One of the first fields to use image processing was space research to compensate for camera distortion of lunar

[2] e.g. Microsoft Encarta etc.

surface images. From the early 1960's to date, image processing has infiltrated numerous fields such as medicine, biology, geography, meteorology, plasma physics, etc.
Image processing, for the purposes of this project, can be defined as the transformation of (at least) one image to another by the use of some local operator. An inventory and a brief description of the major classes of operators this project touches upon follows.

Point Operators The pixel is the smallest unit that can be processed and it represents a single dot on an image displayed on a computer monitor.
Point operators are functions that are applied to individual pixels and are, therefore, position independent operators. The most popular representatives of this category are Thresholding and Gamma correction.

- Thresholding
  An intensity value is supplied by the user above which all pixels values are set to the maximum (or alternatively, the value 1) and below which all pixels value are set to zero. Thus, if the input image contains values in the range 0..255 the output image will only contain pixels with values either zero or 255 (a binary image).
  One application of thresholding is in segmentation whereby an object may be separated from its background.
- Gamma Correction
  An image which is too dark or too bright can be adjusted to preference using the non-linear gamma correction transform which relates closely to a brightness control on a monitor and is used in standard TV cameras.
  The function which relates input to output pixel in the gamma transform is given by:
  $$Out(i,j) = c\, In(i,j)^{\gamma}$$
  A gamma value of 1.0 produces a null transform. For γ less than one pixels with high intensity value are suppressed whereas if γ is greater

than one high intensity pixels are enhanced.
Visually, values in the range 0.0..1.0 brighten the image and values greater than 1.0 darken it.
The normalisation constant c is once again used to make sure that no output pixel value exceeds a preset range (e.g. 0..255). This transform is similar to two well known operators, the logarithmic and the exponential.
The logarithmic is described by:

$$Out(i,j) = c\, log(|In(i,j)|)$$

in which each pixel value is replaced by its logarithm. This has the effect of enhancing low intensity pixel values. The scaling constant c is used to make sure that the output value $O(i,j)$ does not exceed the maximum value (255 in our case). The exponential operator is described by:

$$Out(i,j) = c\, b^{In(i,j)}$$

where c is again the scaling factor, b the basis which has the effect of enhancing high intensity pixel values.

Image Arithmetic Image arithmetic requires two images which are combined to produce a single one as an output. This is a popular class of operators of which subtraction, inversion and logical AND/OR are very widely used.

- Pixel Subtraction

  Pixel subtraction takes two images and produces a third one whose corresponding pixel values have been subtracted, or

$$Out(i,j) = In_1(i,j) - In_2(i,j)$$

  where $Out(i,j)$ represents the output value of the pixel located at (i,j).

It usually forms part of a more complicated series of operations but is also used on its own.

- Inversion

  Inversion produces the effect observed on photographic negatives and is used to change the polarity of an image as part of a larger process. Its equation is simply

  $$Out(i,j) = I_{max} - In(i,j)$$

  where I_{max} is usually equal to 255.

- Logical AND

  Logical AND takes two images and applies the AND operator in a bitwise fashion between corresponding pixels in each image. The effect is to return the intersection of the two images and it can be used to detect changes in consecutive image of a particular scene.

  Similarly, logical OR can be used to obtain the union of two images which is equivalent to merging them.

Geometric Operations These are well known operations which include Rotation, Translation and Scaling. The most interesting of them is rotation whereby a point (i,j) in one image is mapped to another location on a target image using the transformation matrix:

$$M = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix}$$

where θ represents the anti-clockwise rotation angle. The whole transformation can then be written as

$$Out\left(\begin{bmatrix} i \\ j \end{bmatrix}\right) = In\left(M\begin{bmatrix} i \\ j \end{bmatrix}\right)$$

Morphology Under this heading belong operations where the input is usually a binary image plus a structuring element and the output a combination of the two.

A structuring element is usually a 3x3 pixel matrix like the ones illustrated in Figure 4.14. These structuring elements determine the fine details of the effects of an operator on an image using the Hit-and-Miss transform. This transform is the basis for most of the remaining morphological operators.

- Hit-and-Miss

  The hit-and-miss transform operates as follows: The origin of the structuring element is translated to each point of an image in sequence. For each position, all points in the structuring element are compared with the image pixel value they happen to overlap at that point in time. If a match can be made (according to criteria explored later) then the pixel underneath the origin of the structuring element is set to the foreground colour (1 or 255) otherwise to zero.

- Thinning

  This is a useful operation and can be described in terms of the hit-and-miss transform. Its effect is to produce a one-pixel thick version of a binary image (while preserving the overall geometry of the shape). It is described as

  $$thin(I, J) = I - \text{hit-and-miss}(I, J)$$

  where I is the input image and J the structuring element.

Digital Filters These filters are generally 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 described 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, $g(x, y)$ the kernel, α and β are dummy variables for the integration (the range of which is across the entire image) and the symbol $*$ indicates convolution (see [14]).

Later on we shall encounter a more practical discrete convolution implemented for the purposes of this project.
Smoothing is a direct application of the convolution operation as is commonly used to remove noise from an image.
Mean, and Gaussian smoothing are two such techniques which are further discussed in Chapter 4.

Image Synthesis Under this heading one finds noise generation. In order for the effectiveness of a smoothing filter to be measured, one should possess the capability of willfully corrupting an image. Two types of noise are commonly produced:

- Salt and Pepper Noise
  The new image is exactly like the old one except for the fact that several pixels (depending on the percentage of corruption) are replaced by either a maximum or a minimum intensity value. The corrupted image thus has black and white speckles scattered all over it.
- Gaussian Noise
  This type of noise is common in recorded images where the recording medium corrupts the image in a uniform way. This corruption can be modelled by a Gaussian with zero mean and a selected amount of variance.

1.3 Previous Work

Using Java for image processing is quite a novel idea. While this does have advantages its obvious disadvantage is that there is not a lot of previous work to draw upon. A few of the most useful resources are listed below.

1.3.1 HIPR

This has been the single most useful reference for this project. HIPR or Hypermedia Image Processing Reference [13] is a project undertaken by the Machine Vision Unit the Artificial Intelligence Department of the University of Edinburgh. Its purpose is help students learn about image processing by achieving the golden ratio between having too few examples and too much technical information (as is the case with the majority of textbooks in image processing and machine vision). The *Worksheets* section of HIPR includes a series of topics starting from image arithmetic to morphology and image transforms. These worksheets contain theoretical background on each topic plus a number of examples of applications, including numerous before-and-after type of images. This way the student can comprehend the exact effects of the particular operator as well as understand its theoretical basis.
At the end of each section a set of exercises and further references are given for the student to elaborate upon.

1.3.2 Andrew Fitzgibbon

Although image processing is itself a deeply explored field, image processing using Java is not. Using the power of Java as the current Internet platform-independent language one can provide an interactive world-viewable teaching tool for image processing.
The first such demonstration I came across was in the WWW pages of the Marble Project [1] (see Figure 1.2). It was a "thresholding" applet with a slider to set the threshold value and a source and target images. This demonstration, coupled with the relevant background theory (in HTML) made for an entertaining and educational experience.
My project has made use of some of the Java classes written by Andrew Fitzgibbon the most useful of which is *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".

1.3.3 Visilog

Visilog is the standard commercial image processing package used in the Department of Artificial Intelligence in the University of Edinburgh. Its purpose is to provide machine vision students with practical experience in their area of study by implementing a large number of operators such as

- Point (arithmetic, logical, etc.)
- Filtering (for smoothing, sharpening, etc.)
- Morphology (thinning, hit-and-miss, etc.)
- Geometry (rotation, sliding, etc.)

Visilog's features go far beyond the ones mentioned but for the purposes of my project it has been used as a testing and performance benchmark.
Chapter 6 compares the execution speed as well as the output of each operator implemented in Java against Visilog (for speed and correctness).

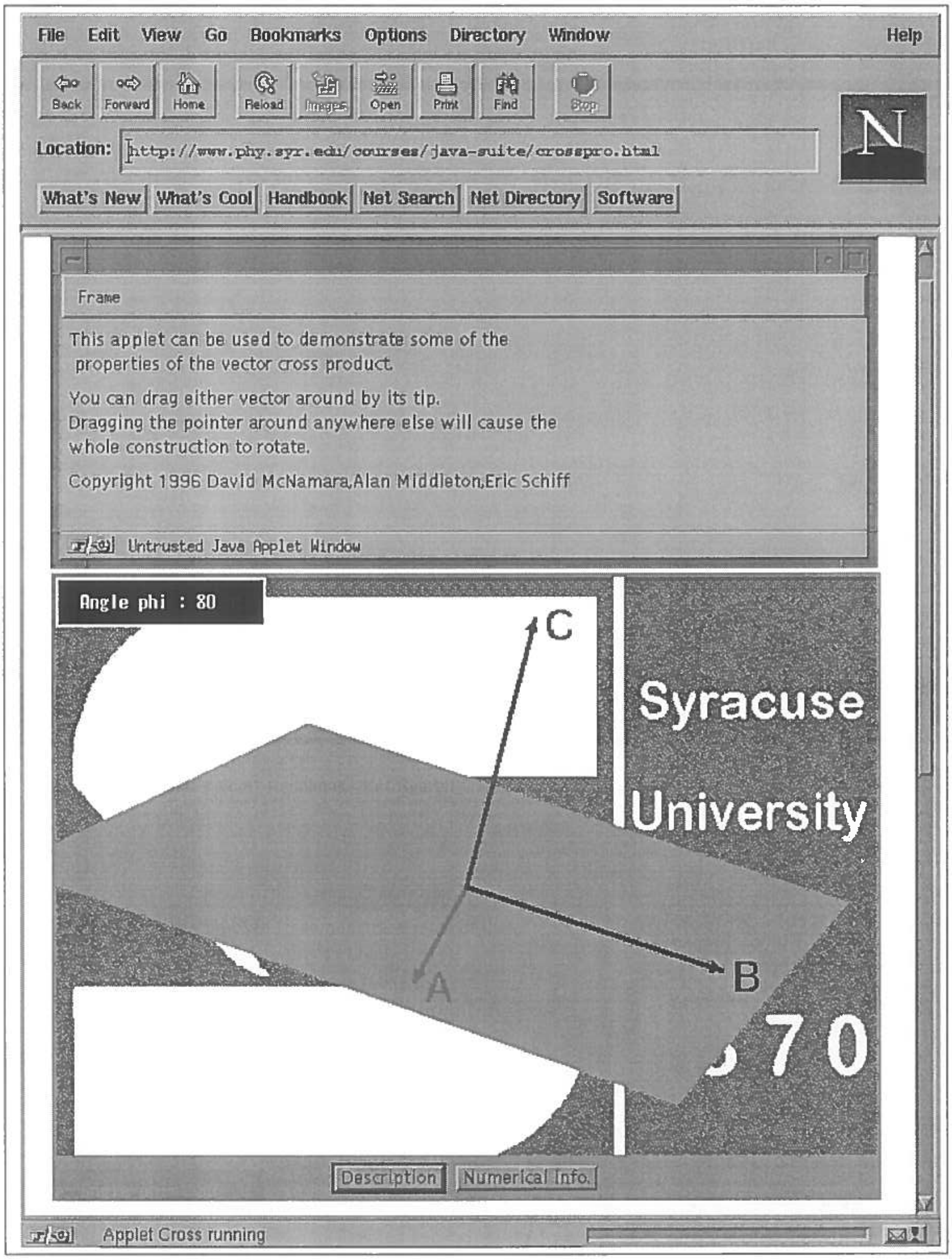
1.4 Aims of this Project

The purpose of this project is to investigate the suitability of Java in image processing and to implement some of the most representative operations in that area.
The final product will be world-viewable via a web-browser and the operators will be able to communicate by passing on output data to each other.

1.5 What follows

Chapter 2 contains an overview of Java and its mechanisms that support image processing.
Chapter 3 briefly describes the generic applet template used in the applets implemented.

Chapter 4 describes the applets implemented in detail and chapter 5 describes the way applets have been made to communicate data with one another.
Chapter 6 briefly compares the performance of Java-based image processing to Visilog.
Finally, chapter 7 briefly lays down the conclusions of this project's work and discusses future plans.



http://www.phy.syr.edu/courses/java-suite/crosspro.html

This applet can be used to demonstrate some of the properties of the vector cross product.

You can drag either vector around by its tip.
Dragging the pointer around anywhere else will cause the whole construction to rotate.

Copyright 1996 David McNamara,Alan Middleton,Eric Schiff

Figure 1.1: Interactive Vector Cross-Product

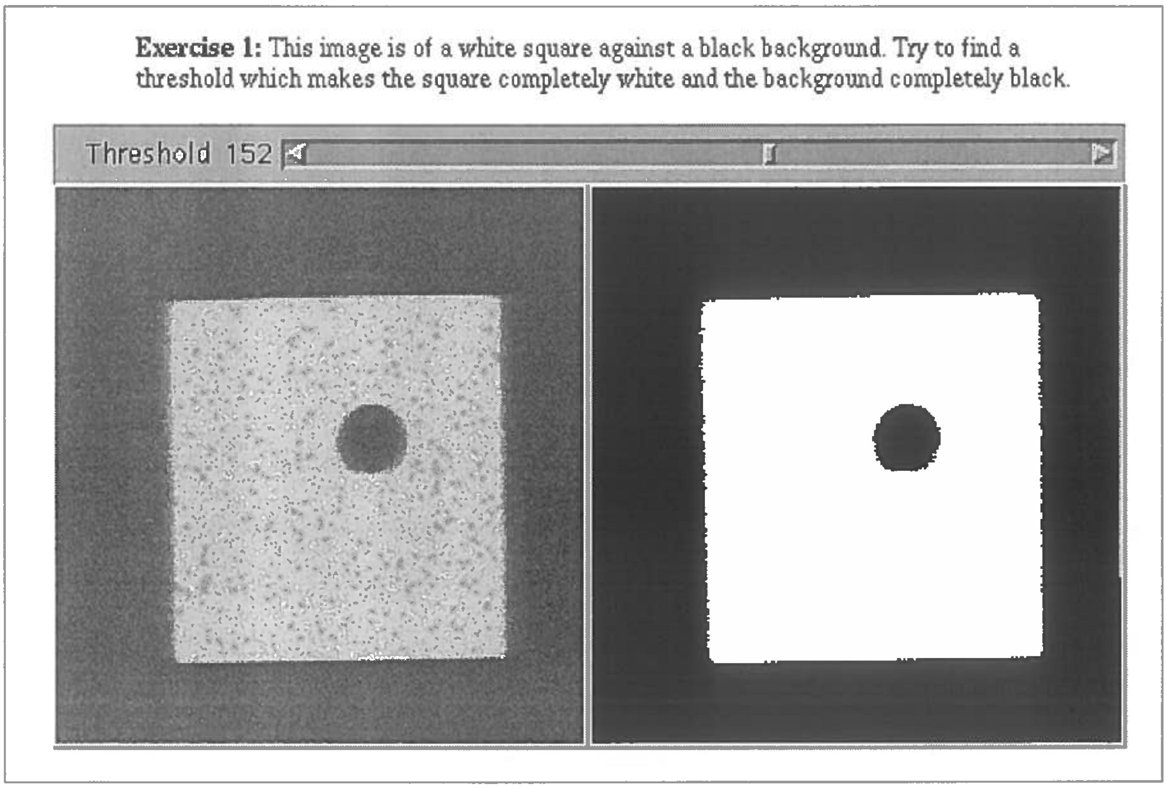

Exercise 1: This image is of a white square against a black background. Try to find a threshold which makes the square completely white and the background completely black.

Figure 1.2: Fitzgibbon's Threshold Applet

Chapter 2

Java

Interesting, from a developers perspective, Java is a compelling technology. It's based on C++, a language that almost all programmers are at least familiar with. Java's creators stripped out most of the Bad Stuff — structs, pointers, unions, multiple inheritance, and more — that are probably responsible for 50% of the C++ programming errors and problems, and added some Good Stuff — automatic memory management, garbage collection, and strings, whose absence are probably responsible for the other 50%.

Steve Mann, PDA Guru.

2.1 Background Information

Java, as mentioned earlier, is an object-oriented language developed by Sun Microsystems. It is modelled after C++ but it is 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[1] document and are down-loaded over the WWW and executed by the Web browser on the reader's machine. Applications are simply conventional stand-alone programs written in the Java lan-

[1]HyperText Mark-up Language

guage. Due to the educational nature of this project, we shall be concentrating on applets rather than applications.

2.1.1 Useful definitions

Class A template for an object. It may contain methods and variables and may also exploit inheritance by extending other classes.

Object An instance of a class.

Method Functions that operate either on instances or classes. No functions outside classes are allowed.

Interface A collection of abstract behaviour specifications that individual classes can implement. This is Java's answer to multiple inheritance. One of the most important interfaces is the *Runnable* which enables the use of threads in applets.

Package A collection of classes and Interfaces. For instance, *java.awt* is the Abstract Windowing Toolkit package which contains all the building blocks for user-interfaces, drawing etc.

2.1.2 Java vs C & C++

This very brief comparison has been adapted from [8].

- Java has no pointers. Instead, variable assignments, arguments passed on to methods etc, are accomplished by reference (for class objects).
- Arrays and Strings in Java are objects, and their boundaries are strictly enforced. Any attempt to access beyond their end means that a compile or run-time error will occur.
- All memory management is automatic so no explicit allocation or deallocation of memory is required (or allowed).

- Errors are objects in their own right and can be handled explicitly. This feature greatly simplifies debugging.
- Multiple inheritance (in the C++ sense) is not allowed.
- All functions must be methods so that no functions outside classes exist.
- Java does not have a preprocessor so no *#defines* or macros exist.
- Java does not support mechanisms for variable-length argument lists to functions.
- The test expression for each control flow statement must return a boolean value (*true* or *false*) and not, for instance, an integer as in C or C++.
- Support for threads is built into the Java language.

2.1.3 Java Bytecodes

When Java code is compiled the result is not directly executable but it must be interpreted by each computer (using a web browser or other tools). This first 'compilation' produces *bytecodes* out of source code. A bytecode instruction consists of a one-byte op code to identify the operation required and a number of bytes which represent the parameters involved.
This system has been employed so that every computer in the world can run the same bytecodes, whatever its architecture. To achieve this, Java has assumed the existence of a Virtual Machine which the bytecode-interpreter (e.g. a WWW Browser) will implement so that the bytecodes can be executed on the host machine. This scheme works fine for most programs but if more speed is required then two solutions are available

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

Java has mechanisms that will allow compiled 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. The better solution is to wait for Just-in-Time compilers[2] which will translate Java bytecodes into native machine code. This will enable Java programs to run at almost the same speed as C.

2.1.4 Applet Methods

Applets share several methods which must be overridden by the user if they are to do anything constructive.

- *init()*. This is the initialisation method and it is called when the applet is first loaded. Usually object creation and image loading is done here.
- *start()*. After initialisation the applet is started. This method is called whenever the user visits the WWW page containing that applet and thus it may be called several times during the lifetime of the applet.
- *stop()*. When the user leaves the page containing the applet this method is called. If the applet contains any threads they must be stopped explicitly though this method.
- *run()*. If the applet implements the *Runnable* interface it may use this method to include anything that is to be run in a separate thread.
- *destroy()*. This method enables the applet to clean up after itself before it is destroyed.
- *paint()*. This method is used if an applets wishes to draw something on screen.

[2]Compilers that will be ready just-in-time for this project!

2.2 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.
This 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 of an Applet instance by supplying a URL[3] parameter.
To display an image one invokes the *drawImage()* method of the Graphics object which is passed onto 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.

2.2.1 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, however, 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 alpha transparency value, and RR, GG, BB the red, green and blue values respectively.
The Java image class allows the programmer to access the pixels of the image as a one-dimensional array. Pixel values are stored in each element of the array in the above form. In order to extract the individual color components from this form, bitwise arithmetic is used:

- R = rgb & 0x00ff0000 $\gg$ 16

[3]Uniform Resource Locator

- G = rgb & 0x0000ff00 $\gg$ 8
- B = rgb & 0x000000ff

2.2.2 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.
Once an *Image* object has been created (using *Applet.getImage()*), the *ImageProducer* for that image can be obtained with the *Image.getSource()* method. If, on the other hand, one is given an *ImageProducer* object, its corresponding *Image* object can be created using the *createImage()* method.
Once an *ImageProducer* object is obtained, it can be manipulated using the classes contained in the package *java.awt.image.*
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 2.1. An image producer is the object which implements the *ImageProducer* interface and which produces the raw data for an *Image* object. This data is sent to an image consumer which is an object that implements the *ImageConsumer* interface. Java allows the user to modify an image is by enabling the user to define and insert an image filter between the producer and the consumer of the image. The producer will then send the data which will be modified by the filter before reaching the consumer.
The filter is an *ImageFilter* object which also implements the the *ImageConsumer* interface since it receives data sent by the *ImageProducer.*
The procedure which follows may be adopted to realize the aforementioned plan The above procedure can then be described as follows:

1. Get an *Image* object using *Applet.getImage()* .

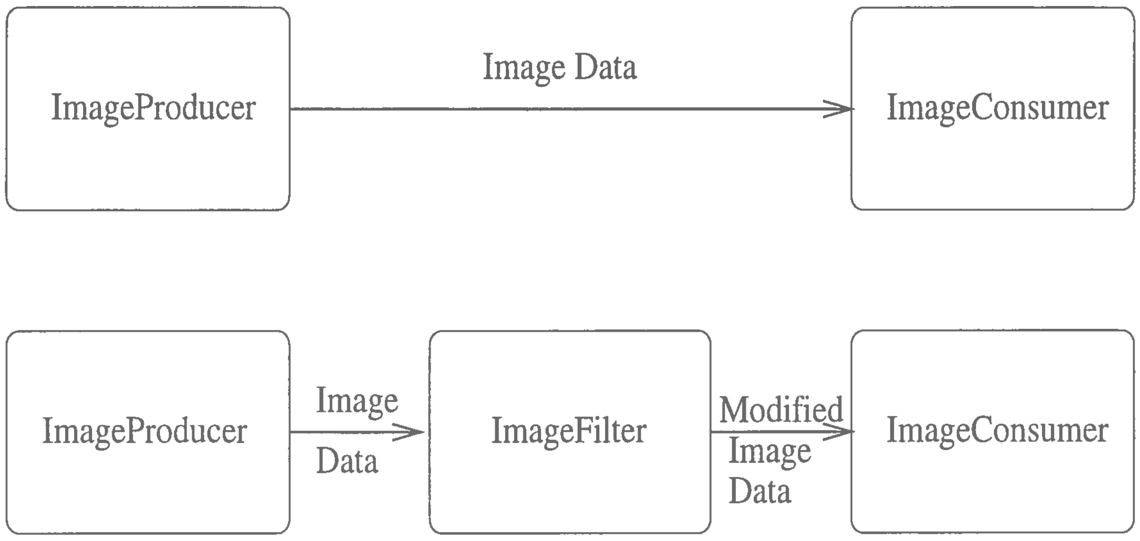


Figure 2.1: Behind the Scenes

2. Using *Image.getSource()* obtain the data source for that image.

3. Create a filter instance.

4. Filter the image using *FilteredImageSource()* which takes the image source and the filter and returns an image producer, and finally

5. Create the new image by using *Component.createImage()* which takes an image producer (created in the previous step).

2.2.3 Single Pixel vs. Neighbourhood Operations

Java has optimised single-pixel operations by giving the user the option to directly filter the colormap instead of working with image pixels.
In neighbourhood operations it is no longer the case that we can simply modify the colormap as with pixel operations. In any case, while colormap modifications are fast and global, they are a bad idea if you want to do anything other than display the image. It is thus a more general solution 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 into a one-dimensional array of integers in the RGB format described earlier.
Most of the applets implemented are neighbourhood operations and the procedure which was adopted is as follows:

- Obtain the source image using *Applet.getImage()*.
- Convert the image into a one-dimensional array using *PixelGrabber.grabPixels()*.
- Extract colors and process as desired.
- Create the target image using the class *MemoryImageSource* and the function *Component.createImage()*.
- Display the target image.

This procedure is described in more detail in the next chapter.

Chapter 3

The Generic Applet

In each of the applets, described individually in chapter 4, certain important steps are common. These steps are described below.

Step 1

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

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

where `src` is an *Image* object, *getCodeBase()* obtains the path where the compiled program resides and `image_name` is a *String* object. This object contains the name of the image which can either be obtained from an HTML document or directly.

```
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 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[1]. 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 is responsible for producing a one-dimensional array of pixels 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) != 0) {
  return;
}
```

`src_1d` will contain the pixel values required provided that the operation executes smoothly.
The method *grabPixels()* initiates the pixel acquisition process. If this process is interrupted then an error occurs.

[1] *ImageCanvas* is a subclass of *Canvas* and was written by Andrew Fitzgibbon.

Step 4

The colors can now be extracted and operated upon from the array `src_1d` containing the image pixels. Section 2.2.1 describes the procedure required to achieve this.

Step 5

The line

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

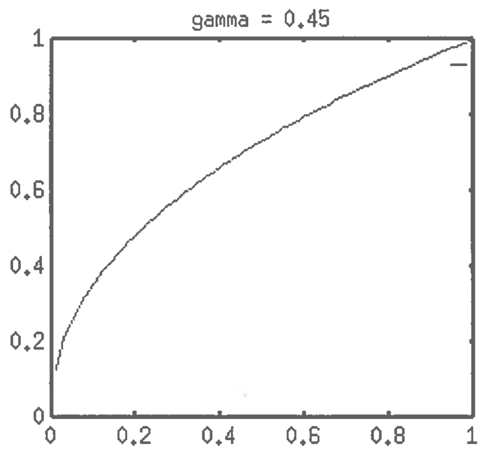
is responsible for creating an *Image* object from a one-dimensional array of pixels of a particular size. This image can be, in turn, tied to an *ImageCanvas* and displayed. The following two lines of code achieve this.

```
dest = createImage(new MemoryImageSource(i_w,i_h,dest_1d,0,i_w));
grid.add(dest_canvas = new ImageCanvas(dest));
```

Chapter 4

The Applets

4.1 The Gamma Correction Applet



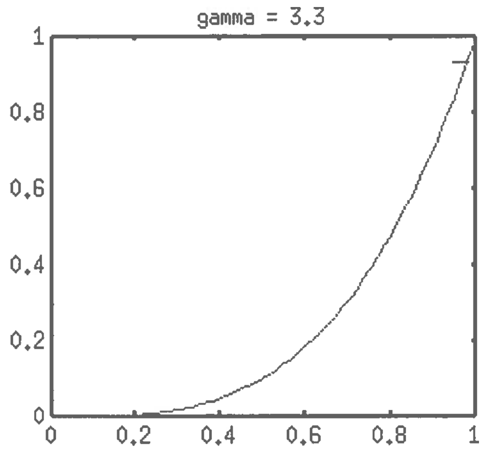


Figure 4.1: 0.45 and 3.3 Gamma Correction transforms

As already mentioned, the gamma transform is a pixel transform in which the input and output pixels are related by the formula:

$$f(x) = c\,x^{\gamma}$$

For the purposes of this project, the formula

$$f(x) = c\,x^{1/\gamma}$$

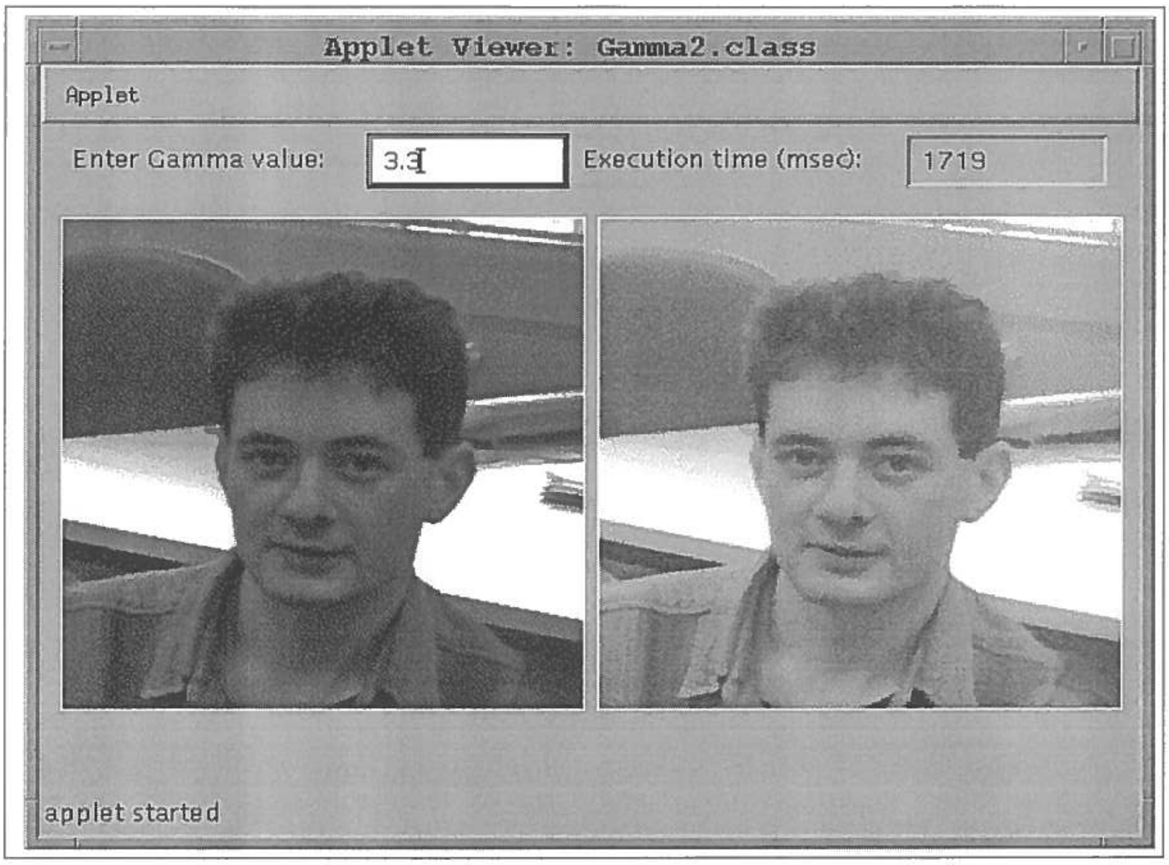


Figure 4.2: The Gamma Correction Applet

was used where x is a pixel value in the range 0.0..1.0.
For γ in 0..1 the output image is darker than the input image whereas for γ greater than one the reverse is true i.e. the output image is brighter. In Figure 4.2 the value of gamma applied to the original image (left) is 3.3. The effect of the function is to considerably brighten the image. The transformation undergone by the pixels is illustrated in the right of Figure 4.1.

4.1.1 Operation

A Gamma value must be supplied in the corresponding text field. By pressing the "return" key the operation of the applet will commence.

Two minor details must be kept in mind here:

1. The pixel values obtained by Java are in the range 0..255 so they have to be normalised to 0.0..1.0 before the transform can be applied and they have to be re-normalised to 0..255 before the image can be displayed.

2. Any values exceeding 255 (due to floating-point imprecision) should be kept at 255.

As a further point of interest, negative gamma values have the effect of producing a photographic negative version of the original image and so they have been allowed.

4.2 The Rotation Applet

Rotation of images is an interesting problem. A two-dimensional anti-clockwise rotation is represented by the corresponding four-element matrix:

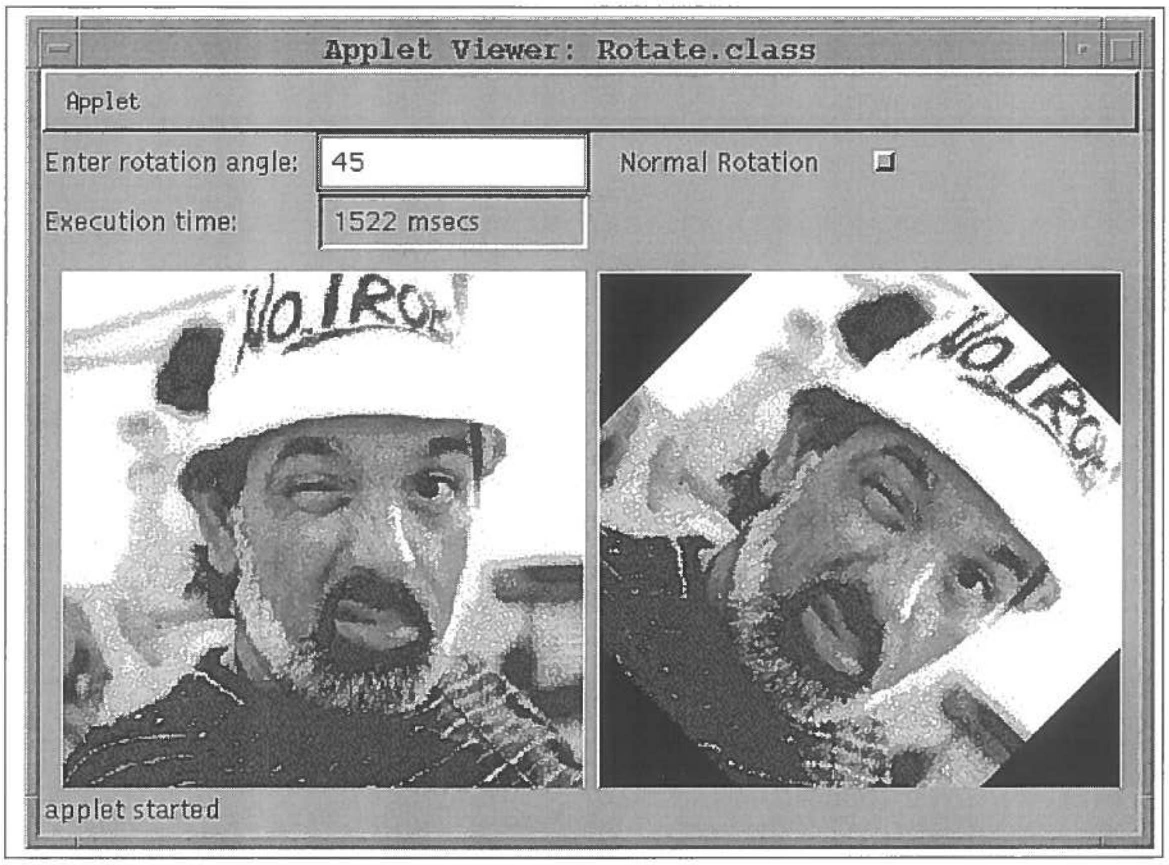
$$M = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix}$$

In image processing a few considerations have to be taken into account before this transformation can be applied. For instance,

- What happens to the parts of the image that are rotated out of the boundaries of the destination image? Do we use a larger frame and rotate the whole image or do some areas get inevitably cropped out?

- Do we represent areas with no source pixels as black areas or do we wrap-around the source image to produce a 'tiling' effect?

- How do we make sure that each pixel in the source image will map to one distinct pixel on the destination image? How can we ensure that not more than one source pixel maps to a destination pixel? This is not possible in general, so we must find a satisfactory approximation.



Enter rotation angle: 45

Execution time: 1522 msecs

Figure 4.3: The Rotation Applet

- Similarly, how can we ensure that no destination pixels are left without a value when they clearly should have one (i.e. are mid-image pixels and not out-of-screen ones).

- Given that rotation can be a computationally intensive operation, how do we cut down on execution time?

In order to address all of the above questions properly, two rotation algorithms have been implemented.

4.2.1 Simple Rotation

This rotation algorithm is a fast implementation of the rotation where we fill in the destination image in raster order by point-sampling the source image. The

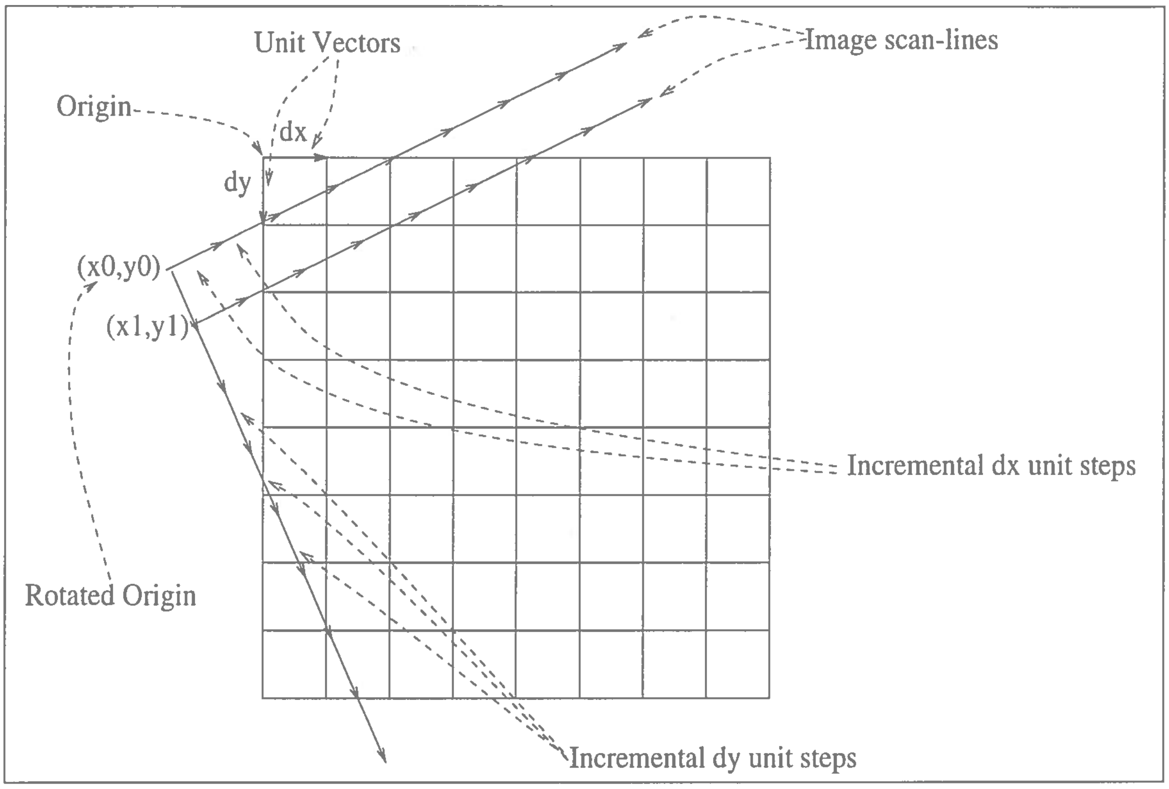


Figure 4.4: Rotation Schema

optimisation 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 outside the boundaries of the source image.
The algorithm operates as follows:

1. Rotate the top left corner of the destination image (origin), about the centroid of the image, employing the above rotation matrix, using the negative value of the desired rotation angle.
   This step occurs once outside any loops. The values for x and y obtained will be used to index into the source image array to retrieve the first pixel of the rotated image (which is placed in location (0,0) of the destination array).

2. Rotate a unit vector on the x-direction and a unit vector on the y-direction using the transformation of step 1 (see Figure 4.4). This rotated unit vector will help us determine which pixel to retrieve from the source image next. This step also takes place once.

3. In order to avoid further expensive matrix multiplications we are going to exploit a property which relates collinear rotated pixels; by adding the rotated unit vector in the x-direction (calculated in step 2) to the rotated origin of step 1 we obtain the next set of coordinates which will be used to index into the source image array. This will return the next pixel to be placed in location (1,0) of the destination array, etc, until we reach the end of the row.

4. The coordinate of the first row have been kept and the coordinates for the next row can now be produced by adding to the ones we have the unit vector in the y-direction.

5. This process is repeated until all locations in the destination image have been filled.

There are several things to note/add here

- Each pixel in the destination array is guaranteed to have a value though not necessarily a distinct one; i.e. it could be the case that with index rounding a pixel in the source image array will be copied in more than one locations in the destination image.

- Pixel coordinates which exceed the limits of the source image are "cropped" out (represented by black color).

- No averaging of pixel values takes place so that for certain angles the effect of aliasing (jaggedness of straight lines) is obvious.

- The speed of execution seems to compensate nicely for the slightly aliased rotated image.

4.2.2 "Shear" Rotation

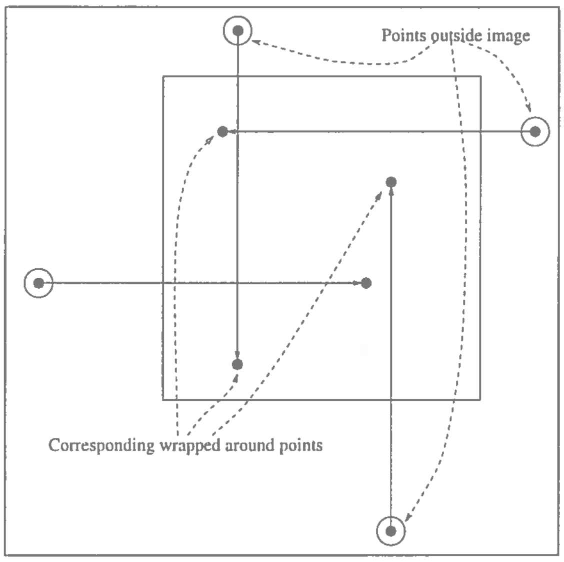


Figure 4.5: Wrap-around of Pixels

This algorithm was implemented using a pseudo-code version described in [3][1]. The general idea is that one can decompose the 2D rotation matrix above into a product of three shear matrices. Raster shearing is done on a scan-line basis and is thus quite efficient.

The shear matrix in the x-direction is given by

$$x\ shear = \begin{bmatrix} 1 & \alpha \\ 0 & 1 \end{bmatrix}$$

and the corresponding one for the y-direction by

$$y\ shear = \begin{bmatrix} 1 & 0 \\ \beta & 1 \end{bmatrix}$$

The product of three 2x2 shear matrices gives rise to a 2x2 rotation matrix

$$\begin{bmatrix} 1 & \alpha \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \beta & 1 \end{bmatrix} \begin{bmatrix} 1 & \gamma \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix}$$

[1]It is due to Alan W. Paeth of the University of Waterloo in Ontario, Canada.

from which the following shear product can be derived (using half-angle identity for the tangent as the previous solution is numerically unstable near zero):

$$\begin{bmatrix} 1 & -tan(\frac{\theta}{2}) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ sin(\theta) & 1 \end{bmatrix} \begin{bmatrix} 1 & -tan(\frac{\theta}{2}) \\ 0 & 1 \end{bmatrix}$$

which represents a counterclockwise rotation by θ.
In this algorithm, scan-line shearing is approximated by a blending of adjacent pixels. For angles less than 45 degrees the algorithm operates at its best and no visible shifts in intensity are produces neither are "holes" introduced. "All pixel flux is accounted for".
There are a few things to note/add here as well:

- This implementation, contrary to the previous one, "wraps-around" parts of the image that would normally reside outside the boundaries of the rotated image according to Figure 4.5. This produces a "tiling" effect and none of the rotated image is black.

- This implementation is buggy in that it rotates only a subset of the image correctly. The image is corrupted near the edges. This could be fixed if we used much larger images as intermediate steps between the shears.

- Shear rotation eliminates the aliasing problem encountered in simple rotation.

4.2.3 Operation

The operation of the applet is quite simple. The user enters the angle of rotation (in degrees) in the text area present and chooses the rotation algorithm by ticking the relevant button. Rotation commences as soon as the "return" key is pressed within the text area. The rotation is clock-wise.

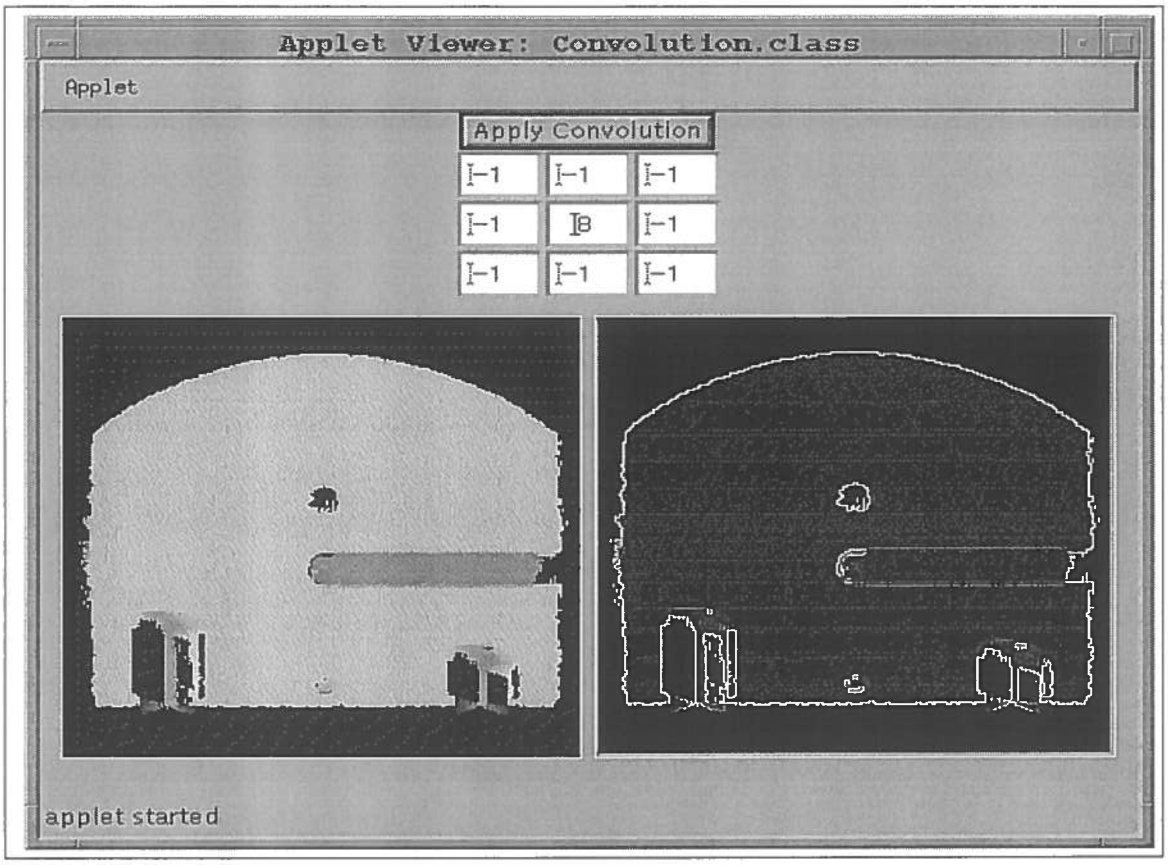
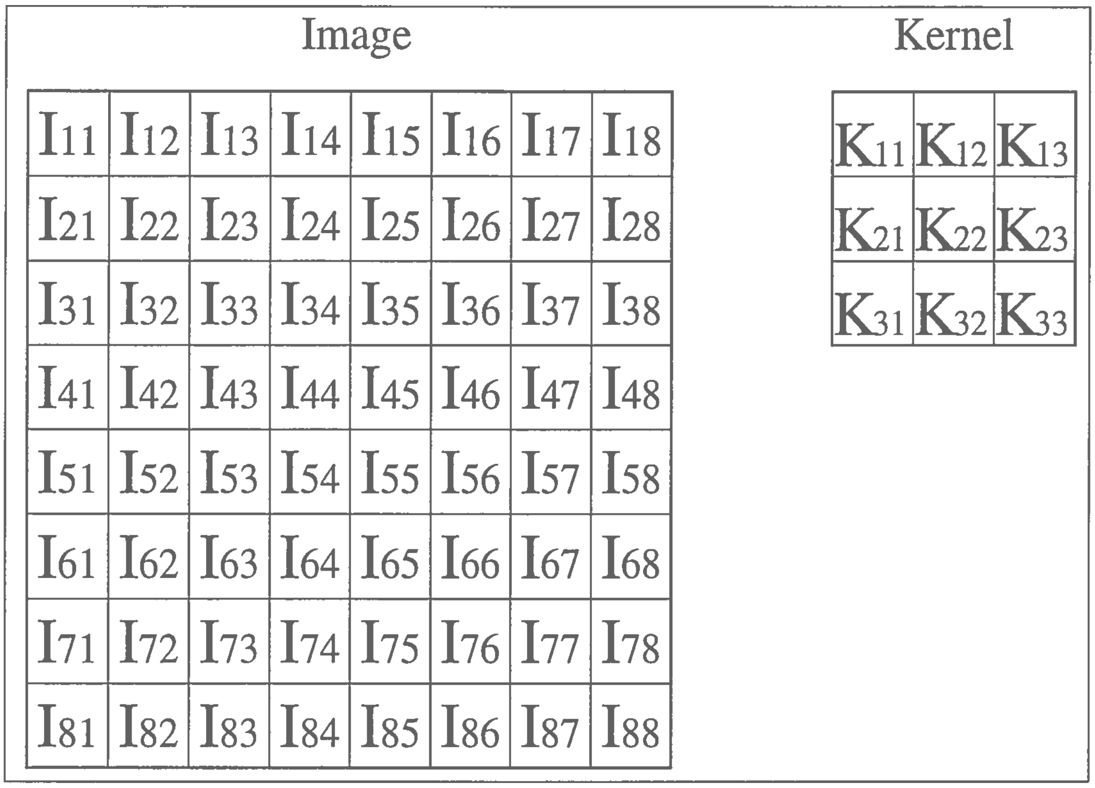


Figure 4.6: The Convolution Applet

4.3 The Convolution Applet

A convolution is a neighbourhood operation that takes place over the whole image. A convolution kernel is essentially a matrix, which may represent a discretely sampled function, which is applied to a number of pixels of the same size and shape as the kernel. If one mentally placed the kernel over each pixel of the source image in turn, then below each kernel element there exists a pixel element. Thus, each output pixel of the destination image is a function of all the elements of the kernel as well as the corresponding pixel elements of the source image.
Convolution is typically used in image processing whenever the output pixels can be described by a simple linear combination of the input pixels.



Image

I_{11}	I_{12}	I_{13}	I_{14}	I_{15}	I_{16}	I_{17}	I_{18}
I_{21}	I_{22}	I_{23}	I_{24}	I_{25}	I_{26}	I_{27}	I_{28}
I_{31}	I_{32}	I_{33}	I_{34}	I_{35}	I_{36}	I_{37}	I_{38}
I_{41}	I_{42}	I_{43}	I_{44}	I_{45}	I_{46}	I_{47}	I_{48}
I_{51}	I_{52}	I_{53}	I_{54}	I_{55}	I_{56}	I_{57}	I_{58}
I_{61}	I_{62}	I_{63}	I_{64}	I_{65}	I_{66}	I_{67}	I_{68}
I_{71}	I_{72}	I_{73}	I_{74}	I_{75}	I_{76}	I_{77}	I_{78}
I_{81}	I_{82}	I_{83}	I_{84}	I_{85}	I_{86}	I_{87}	I_{88}

Kernel

K_{11}	K_{12}	K_{13}
K_{21}	K_{22}	K_{23}
K_{31}	K_{32}	K_{33}

Figure 4.7: How Convolution Works

Mathematically, a convolution is described by the following formula:

$$O(i,j) = \sum_{k=1}^{m} \sum_{l=1}^{n} I(i+k-1, j+l-1)\, K(k,l)$$

where O() and I() are the output and input images respectively and K() the kernel. To make the mathematics more concrete consider the two arrays in Figure 4.7 where I_{ij} represents image elements and K_{ij} kernel elements. Upon completion of the first iteration of the convolution procedure the output element O_{11} takes the value

$$O_{11} = I_{11}K_{11} + I_{12}K_{12} + I_{13}K_{13} + ... + I_{33}K_{33},$$

upon completion of the second iteration, element O_{12} takes the value

$$O_{12} = I_{12}K_{11} + I_{13}K_{12} + I_{14}K_{13} + ... + I_{34}K_{33}$$

and so on. What happens when the kernel reaches the image boundaries and so no complete overlap between kernel and image exists? Three techniques are commonly employed.

1. One can either ignore those pixels in which case the output image will be slightly smaller [2] than the original, or

2. Making the input image larger to accommodate the kernel used and produce an output image the same size as the original input image. In this case one must invent pixel values to enlarge the original image.

3. Return a default value (e.g. 0) at the unprocessed boundary

Of these three the last one has been employed. If the destination image in Figure 4.6 is closely observed the empty pixels on the borderline will be clear. Two of the most common convolution kernels are the Laplacian

$$L = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

used for detecting edges, and the low-pass filter

$$H = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

which has the effect of smoothing the image by removing the high frequencies (see [9]).

4.3.1 Operation

The values are entered in each of the nine text areas of the applet, each one representing the corresponding one of a 3x3 kernel. Real values are allowed so

[2] By the width of the kernel minus one.

that, for instance, low-pass filters can be tried out.
By pressing the "Apply Convolution" button, the applet will commence its execution.
If a resulting value exceeds 255 then it is kept at 255 and negative values are kept to zero.

4.4 The Noise Generating Applet

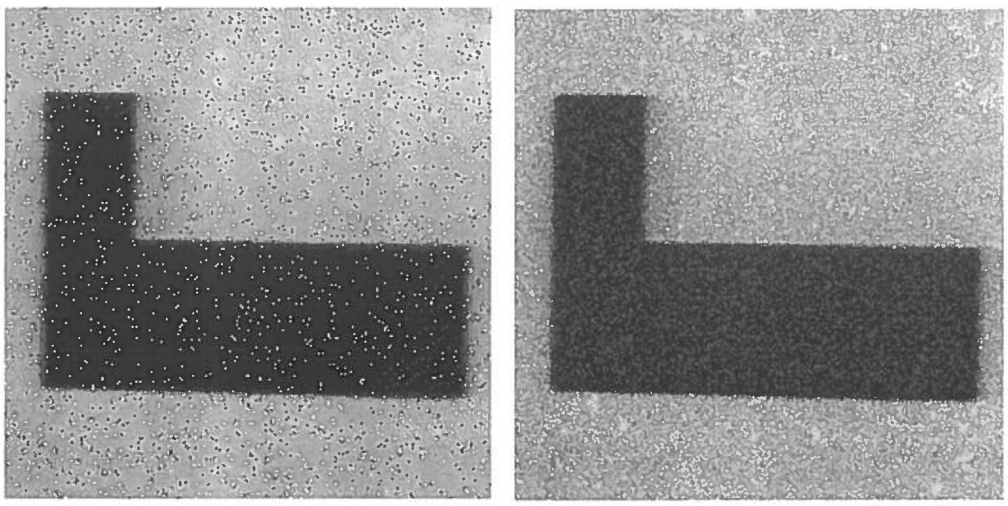


Figure 4.8: Salt-Pepper and Gaussian Noise

The process of capturing an image is not ideal and we cannot expect to get a perfect image out of a natural scene. This discrepancy between real and captured image is called Noise and its causes vary from the sensitivity of the detector to quantisation errors.
The forms of noise most commonly present in images are salt-pepper and Gaussian noise.

Salt-pepper This type of 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.
This type of noise is normally due to errors in data transmission.

Gaussian This type of noise is due to the properties of the detector and is present, to various degrees, in all recorded images. Its name is derived by the fact that it is described by a Gaussian function:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}}$$

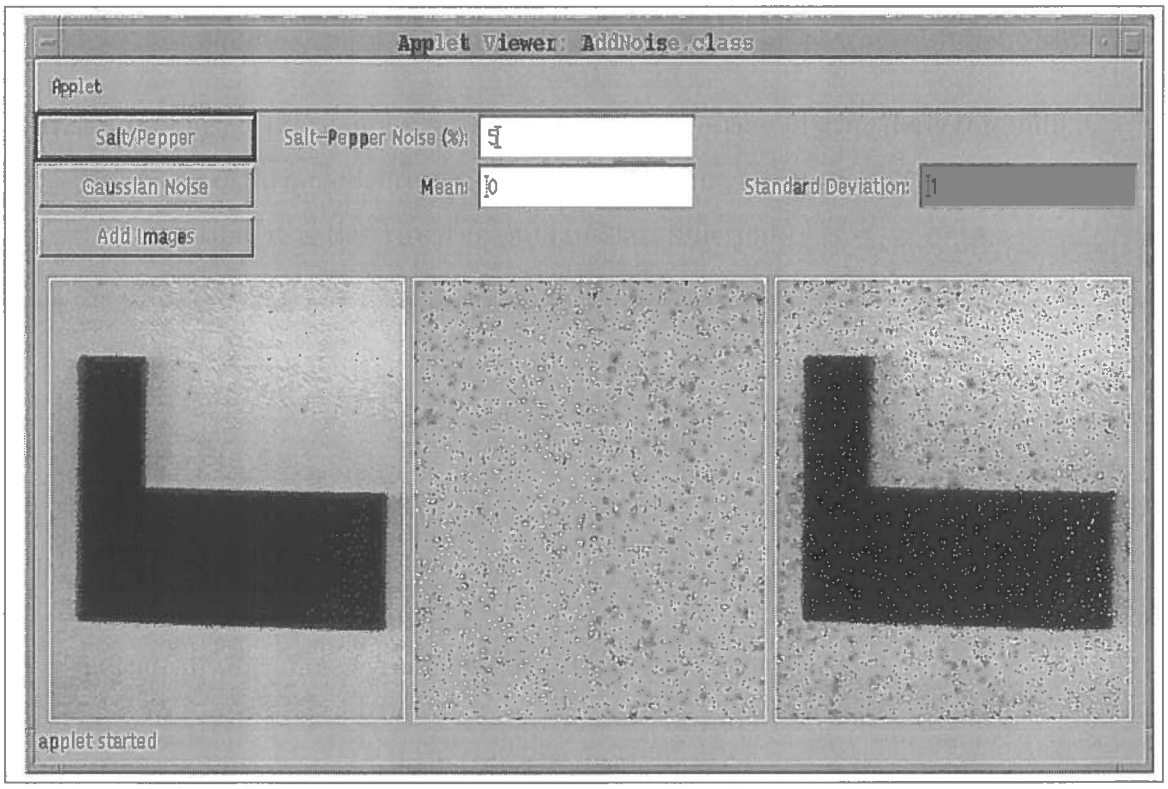
where

$$\mu = \int_{-\infty}^{\infty} x p(x) dx$$

is the mean (usually zero),

$$\sigma^2 = \int_{-\infty}^{\infty} (x-\mu)^2 p(x) dx$$

is the variance and σ the standard deviation.



Salt-Pepper Noise (%): 5
Mean: 0
Standard Deviation: 1

Figure 4.9: The Noise Generating Applet

Both types of noise can be created as separate images and then superimposed on the original image, using image arithmetic, to produce a corresponding noise

image. Figure 4.8 contains two corrupted images.[3]. The left one contains salt-pepper noise with probability five percent whereas the other contains zero-mean Gaussian distributed noise with $\sigma = 2$.
The reason one would desire to corrupt an image in the first place is to test the operation of various filtering techniques which will be discussed further below.

4.4.1 Operation

The applet is displayed in Figure 4.9 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 proceed in generating the type of noise desired by pressing on the corresponding button. After a noise image has been generated (like the middle one in Figure 4.9) it can be added to the original image by pressing the "Add Images" button.

4.4.2 Algorithmic Details

In order to produce salt-pepper noise one requires the use of a random number generator. If the value returned by the generator is greater or less than a certain value dictated by the probability parameter (supplied by the user), then the corresponding pixel is flipped to black or white accordingly.
This process has to be repeated for each element of the image array and even with a 256x256 the time taken is several seconds as calls to random number generators are generally 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. If however many applets are being loaded at the same time on a web page, this delay will be more pronounced

[3]The uncorrupted images may be found in appendix A.

and may become irritating.
In order to overcome this bottleneck the following quick-and-dirty method was employed:

- An array the size of the image filled with random numbers is calculated during the startup phase of the applet, as a separate low priority thread.

- From that same random number generator, a $\mu = 0, \sigma = 1$ Gaussian distributed array is produced using the empirical formula

  $$togauss(r) = 3.47 * r - 1.735$$

  where r is the value produced by Java's random number generator *Random.nextFloat()*. Note that this would not work if the results of this generator were not already approximately Gaussian distributed.
  The fact that the above formula works was experimentally verified by feeding its output to a program (adapted from [16]) which returned the mean and the standard deviation of that data.

- In order to produce Gaussian-distributed noise with varying μ and σ values, the formula

  $$anygauss(g) = \sigma * g + \mu$$

  is employed where g is a Gaussian value picked from the array constructed initially (with $\mu = 0, \sigma = 1$).

Negative Gaussian noise in the noise image has not been normalised so large mean and standard deviation values may have a strange color (due to overspill to the other color channels). Normalisation has, instead, taken place in the addition of Gaussian noise to the original image.

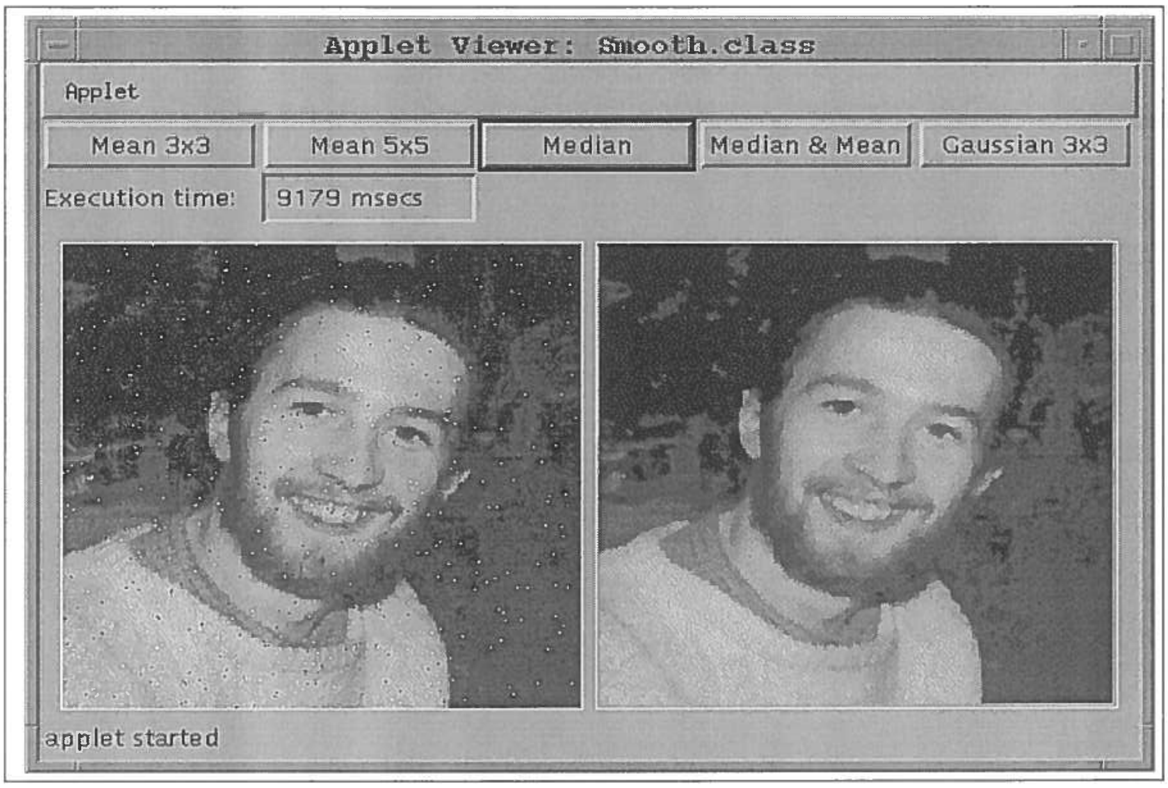


Figure 4.10: The Noise Reducing Applet

4.5 The Noise Reducing Applet

4.5.1 Mean Smoothing

Mean smoothing in image processing can be described by a convolution kernel which for the 3x3 case would look like this:

$$Mean\,(3x3) = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

When this kernel is passed over an image its effect is to average the pixel values in each 3x3 image area.

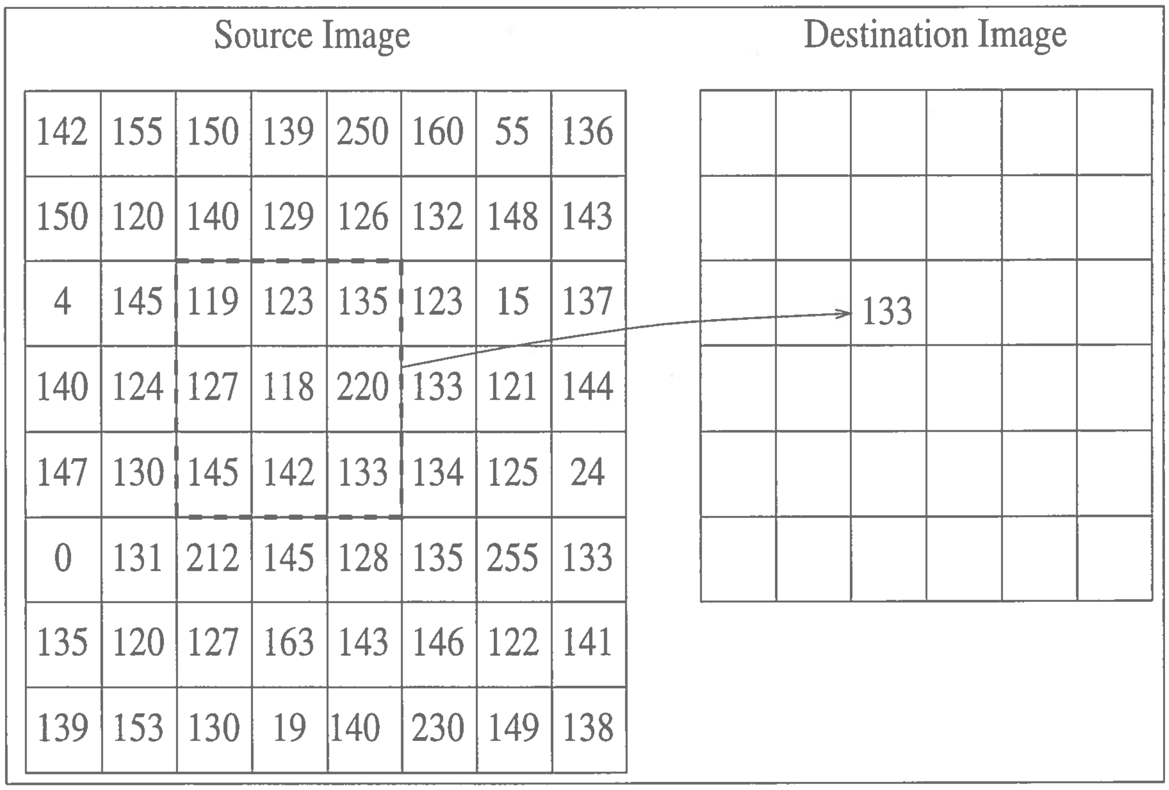
This reduces the amount of intensity variation between neighbouring pixels and thus reduces noise but also visually blurs the image. Larger kernels (e.g. 5x5, 7x7) suppress noise even more but they also tend to blur-out the fine details in

an image. It is a fine balance between removing noise and retaining image detail but in general a 3x3 smoothing operator is sufficient.
The 5x5 mean smoothing kernel which looks like this

$$Mean\ (5x5) = \frac{1}{25}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

has also been implemented for comparative purposes.

4.5.2 Median Smoothing



142	155	150	139	250	160	55	136
150	120	140	129	126	132	148	143
4	145	119	123	135	123	15	137
140	124	127	118	220	133	121	144
147	130	145	142	133	134	125	24
0	131	212	145	128	135	255	133
135	120	127	163	143	146	122	141
139	153	130	19	140	230	149	138

		133			

Figure 4.11: The Median Operator

As opposed to mean smoothing, this filter replaces a set of pixels belonging to the same neighbourhood by the their median value. The median value of an odd-numbered sorted set of numbers is the half-way point of those numbers (or the average of the two in the case of an even set of numbers). Figure 4.11 illustrates this idea on a 3x3 image neighbourhood.
The procedure thus is quite straight forward:

- Retrieve first set of pixels to be processed.
- Sort[4] them and extract the median value.
- Place this value in the corresponding pixel location of the target image.

Median smoothing, although more computationally intensive than mean smoothing, does offer substantial advantages:

- Isolated high frequency noise (e.g. salt-pepper) is effectively removed since it will not be the median value.
- Fine detail is preserved since no averaging is involved.
- All pixel values in the result image are guaranteed to be ones that existed in the source image.

If substantial corruption of the image has occurred, however, one is best off using the mean filter.

4.5.3 Gaussian Smoothing

Gaussian smoothing can be implemented by using discrete convolution kernels. A 3x3 kernel may have the form

$$Gaussian\,(3x3) = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

[4]Initially using quick-sort from [7], but insertion-sort operated much faster for a nine-element array

and its equivalent 5x5 kernel

$$Gaussian\ (5x5) = \frac{1}{115}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

This type of smoothing is quite similar to mean smoothing except that the kernel values are discretely-sampled and Gaussian-distributed (see formulas in Noise Generating Applet). The Gaussian filter has a useful property which the mean filter does not: by choosing the size of the Gaussian one can be certain of the spatial frequencies that are present after filtering. Furthermore, it is biologically plausible and it approaches the optimal edge-detection filter that is used in the Canny edge detector.

4.5.4 Mean-Median Smoothing

One button of interest here is the "Mean & Median" button. The way this operates is as follows:

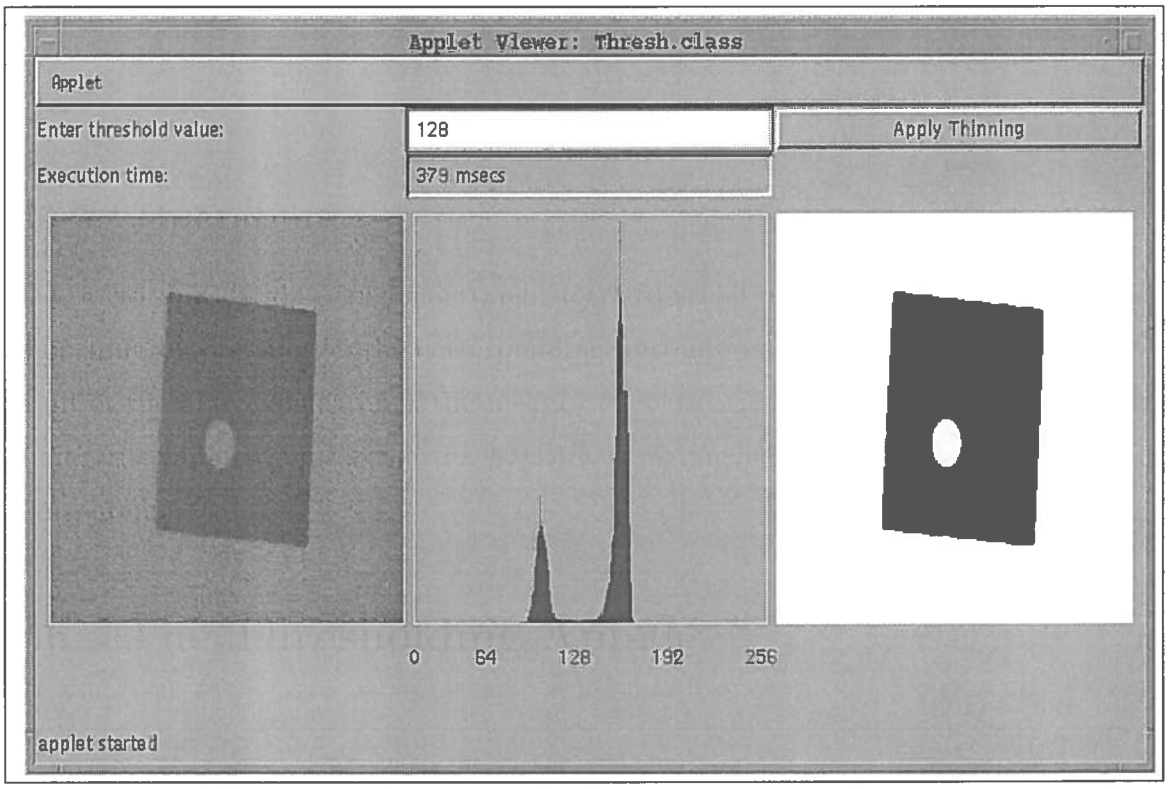
- One pass of each 3x3 neighbourhood is made during which black and white pixels are counted (salt-pepper noise).
- Pixel values excluding salt-pepper noise are summed together.
- The resulting pixel is the average of the values added in the previous step.

This has the effect of removing salt-pepper noise more efficiently than the median operator (for noise greater than approximately 8%) and also performing a mild smoothing effect. If no salt-pepper noise is present then it operates just like the mean filter. The more salt-pepper noise present the more the operation approximates median filtering.

4.5.5 Operation

The operation of this applet is straight-forward. The user chooses the type of smoothing by pressing on the corresponding button.
In all of the above algorithms, the un-processesed corners of the images are left blank (the alpha value of the pixels there is assigned to zero so they become transparent).

4.6 The Thresholding Applet



Enter threshold value: 128
Execution time: 379 msecs

Figure 4.12: The Thresholding Applet

This is a straightforward and quite fast operation 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. Mathematically

$$new(i,j) = \begin{cases} 1\ (or\ 255) & for\ old(i,j) \geq Threshold \\ 0 & for\ old(i,j) < Threshold \end{cases}$$

The output is a binary image and this process is used either as a preliminary segmentation procedure (e.g. separating object from background) or for producing binary images from edge-detection filters for further processing.

4.6.1 The Histogram

The Intensity Histogram is a useful tool in image processing for various reasons. One of its most popular uses is in helping to select a suitable threshold value.
It 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 4.12).
In our case, the histogram shows two clearly separable peaks. This is a "bimodal" histogram, that is one with two clearly separate peaks. The first one represents the object and the second one the background.
By choosing a value between those two peaks (e.g. 128) we can separate object from background nicely.

4.6.2 Operation

As soon as an image is loaded, its histogram is produced and displayed. If the distribution of pixel values is similar to the one in Figure 4.12 then a suitable threshold value can be easily found.
This value can then be entered in the relevant text-field. By pressing "return" inside the text-field, a thresholded image of the original one is placed on the right-hand-side of the applet.
Threshold values less than zero or greater that 255 are not allowed.

4.7 The Thinning Applet

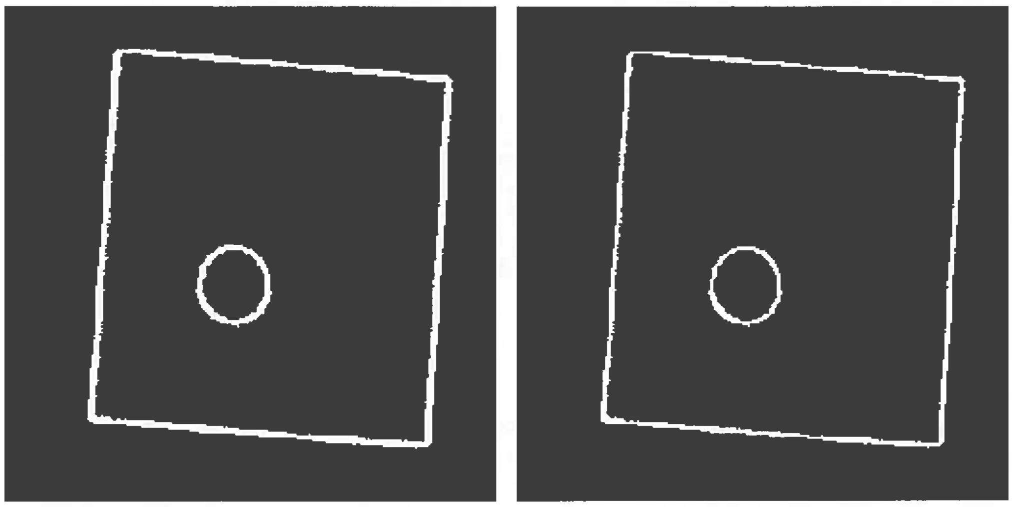


Figure 4.13: Before and After Thinning

Thinning is a morphological operator applied to binary images. It can be used for skeletonisation of images or, more commonly, for producing a one-pixel thick outline of the object in question. The thresholded output from an edge detecting filter (e.g. Sobel) is a binary image containing the outline (edges) of an object. This outline is often several pixels wide and this is not always desirable. One way to produce a single-pixel wide outline of the object without incurring loss to the geometrical properties of the shape is to process the binary image with the structuring elements in Figure 4.14. This processing takes the form of a hit-and-miss transform and it operates as follows:

- Sweep the image with the structuring element.
- 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 resulting image, otherwise put a zero.
- 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.

0	0	0
	1	
1	1	1

	0	0
1	1	0
	1	

1		0
1	1	0
1		0

	1	
1	1	0
	0	0

1	1	1
	1	
0	0	0

	1	
0	1	1
0	0	

0		1
0	1	1
0		1

0	0	
0	1	1
	1	

Figure 4.14: Thinning structuring elements

- Repeat this processes until all structuring elements have passed over the image (each element takes as input the output of the previous one)
- Repeat all of the above until the image doesn't change anymore, i.e. no more patterns 'hit' from the above process and thinning has thus been completed.

This algorithm guarantees that connectivity will be preserved so the overall geometric structure of the object in the image is preserved. Figure 4.13 contains an example of a binary image and its thinned equivalent as produced by the thinning applet.
One obvious disadvantage of this technique is that it takes a long time. In interpreted Java code a straightforward convolution of a 256x256 image with a 3x3 kernel takes approximately five seconds. If a similar process has to take place at least eight times for one iteration (several are required then one might agree that the interactivity of the thinning applet is not stunning.

Chapter 5

Communicating Applets

We have reached a stage where several independent applets have been created. A natural progression is to include them all in an HTML document and display them in a Web page. Furthermore, it would be nice if the output image from each applet could be fed to the next one down the page so that it can be further processed.

The way to achieve this is by giving each applet the ability to communicate with the next one.

First of all, though, we must add the applets to an HTML page. This can be achieved by the following piece of HTML code:

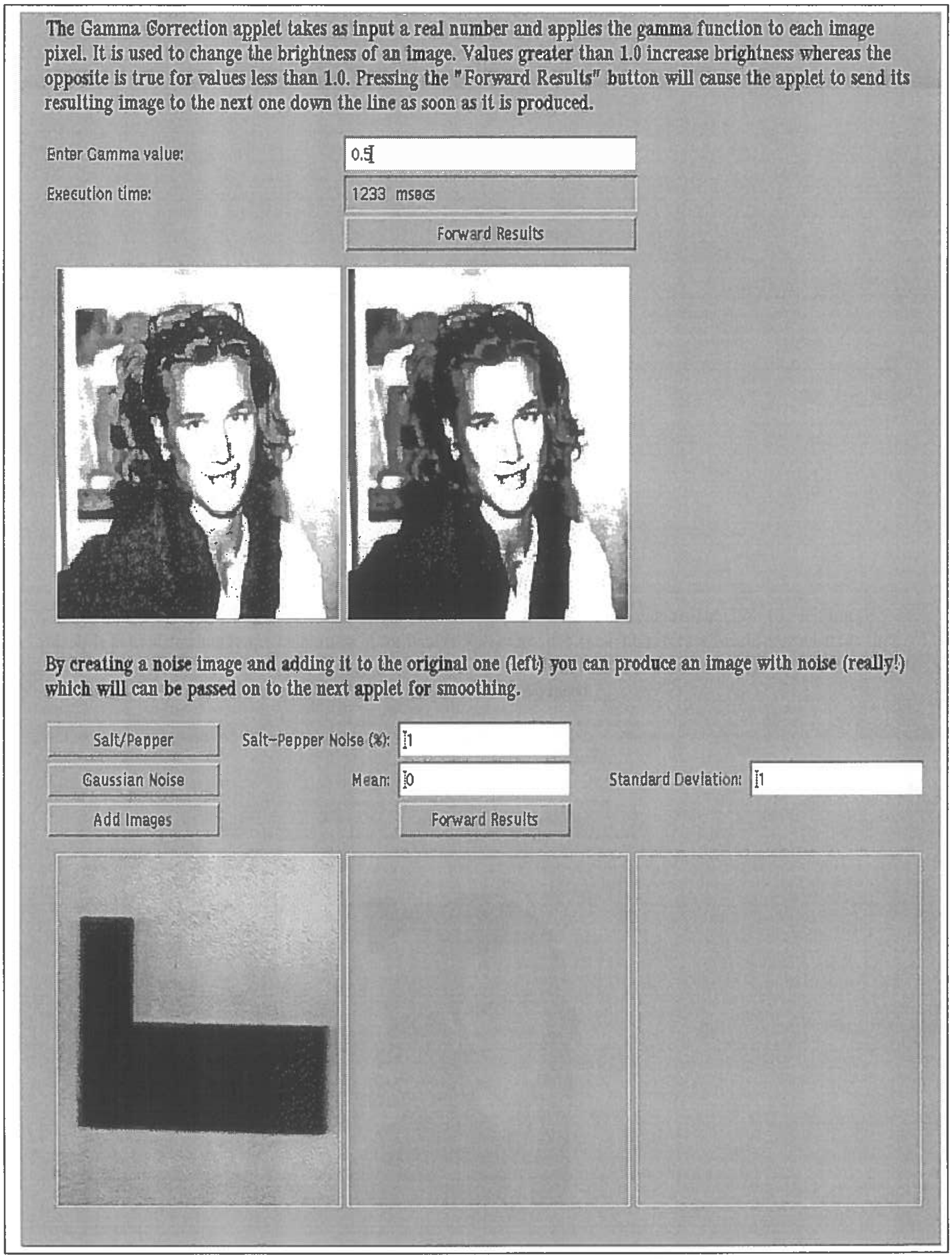
```
<APPLET CODE="AddNoise.class" WIDTH=800 HEIGHT=400>
<PARAM NAME=images  VALUE="images/holesquare2.gif">
</APPLET>
```

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

Next, the procedure which Java employs to grant applets the privilege of communicating with each-other is fairly simple and works as follows:

- The 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">
  ```


The Gamma Correction applet takes as input a real number and applies the gamma function to each image pixel. It is used to change the brightness of an image. Values greater than 1.0 increase brightness whereas the opposite is true for values less than 1.0. Pressing the "Forward Results" button will cause the applet to send its resulting image to the next one down the line as soon as it is produced.

Enter Gamma value: 0.5

Execution time: 1233 msecs

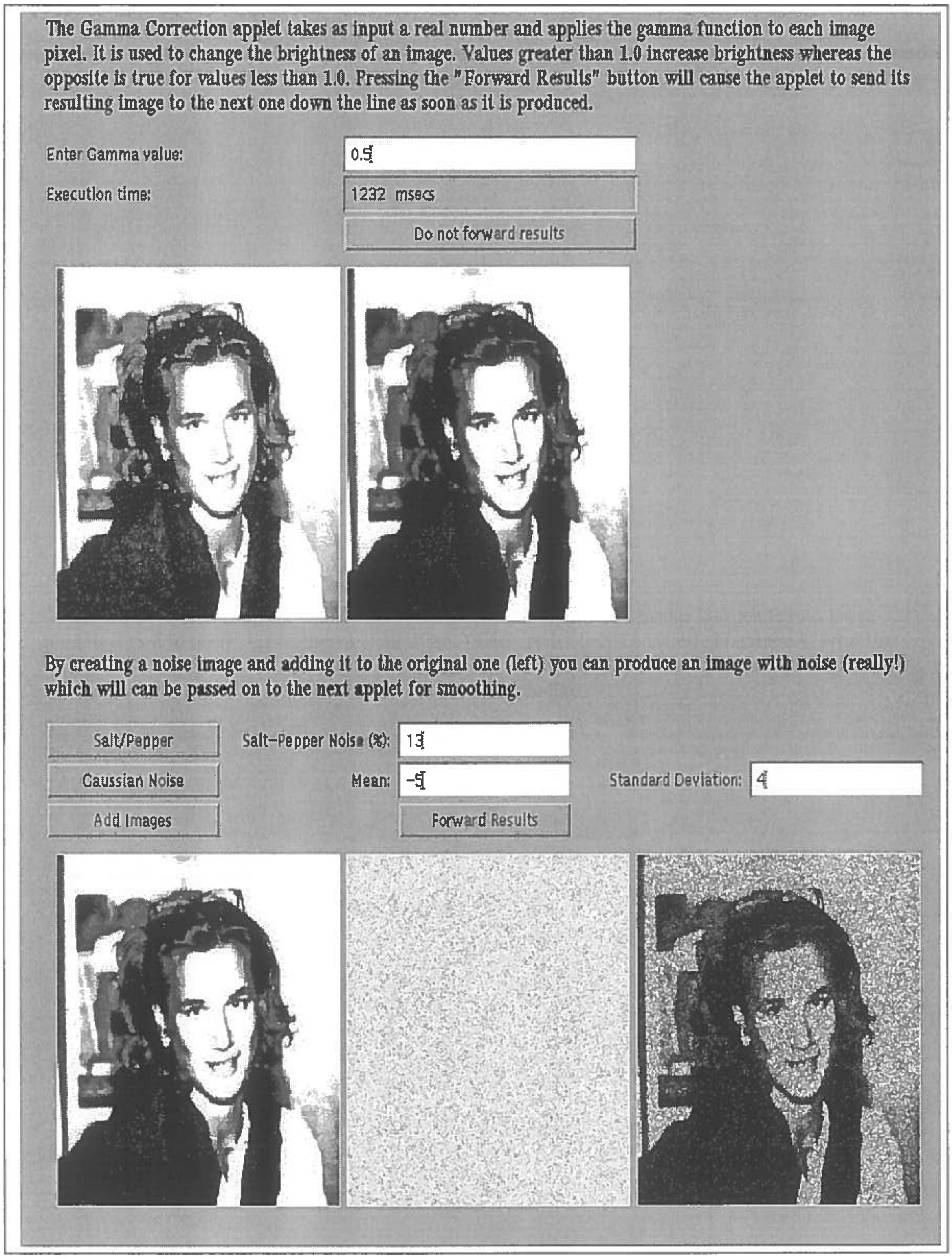
By creating a noise image and adding it to the original one (left) you can produce an image with noise (really!) which will can be passed on to the next applet for smoothing.

Salt-Pepper Noise (%): 1

Mean: 0

Standard Deviation: 1

Figure 5.1: Communicating Applets 1


The Gamma Correction applet takes as input a real number and applies the gamma function to each image pixel. It is used to change the brightness of an image. Values greater than 1.0 increase brightness whereas the opposite is true for values less than 1.0. Pressing the "Forward Results" button will cause the applet to send its resulting image to the next one down the line as soon as it is produced.

Enter Gamma value: 0.5

Execution time: 1232 msecs

By creating a noise image and adding it to the original one (left) you can produce an image with noise (really!) which will can be passed on to the next applet for smoothing.

Salt-Pepper Noise (%): 13

Mean: -5

Standard Deviation: 4

Figure 5.2: Communicating Applets 2

- Then, one must use the *getApplet()* method from the applet context with the name of that applet as a parameter. This will return a reference to that applet. One can use this applet as if it where an object and manipulate it accordingly. For example, if one wishes to obtain the applet named "addnoise" belonging to the AddNoise class one 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()*. This function is allowed to activate after the button "Forward Results" has been pressed (this button only appears if another applet can be found in the same page). The effect of *send_image()*, is to send the whole integer array of the final image produced by this applet to the next applet down the line.
  For example, consider Figures 5.1 and 5.2. The applets pictured are called "gamma" and "addnoise" respectively. In Figure 5.1 the "gamma" applet operates on its own. After the "Forward Results" has been pressed and the gamma operation repeated, the resulting image is passed on to the "addnoise" applet (Figure 5.2).
  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 is the second extra function the applets have been equipped with in order to communicate. Its purpose is simply to set the source image of the applet which defines it to be the image sent by the applet that calls it.
  In Figure 5.2, the source image of the "addnoise" applet is set by a method within this applet called by the *send_src_image()* of the "gamma" applet.

Now each applet can pass its image to the next applet as illustrated by Figure 5.3. To make the communication procedure more general a parameter has been

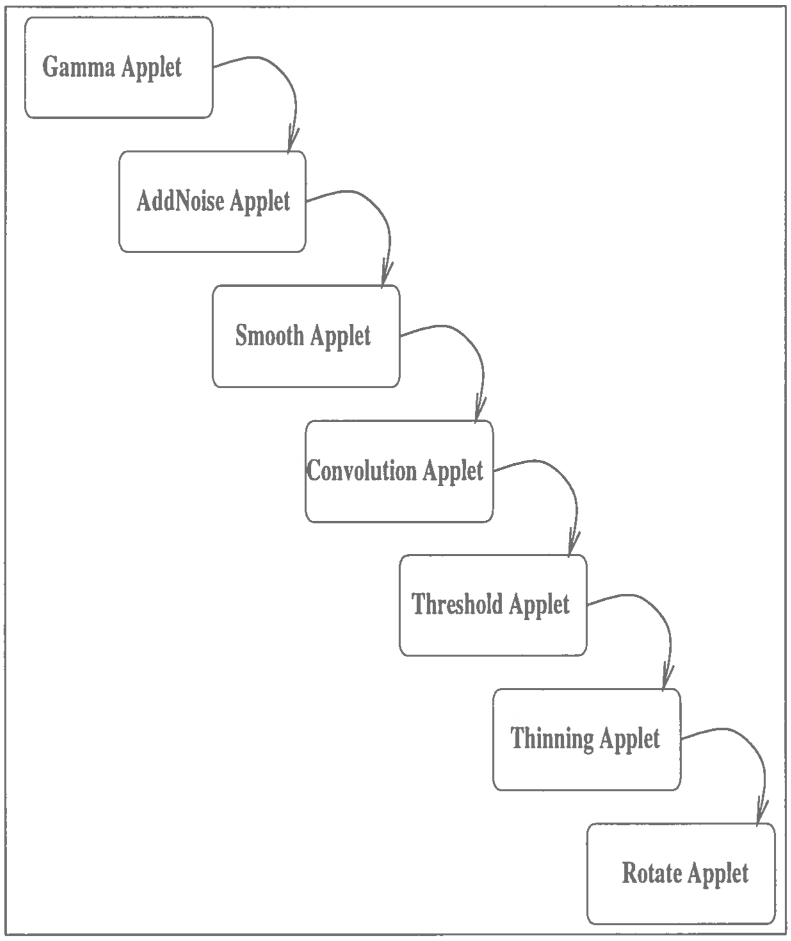


Figure 5.3: The Applets and the Cascade of Images

added which can be set from the relevant HTML document. This parameter is called `receiver_applet` and is used to set the destination of the resulting image of an applet. If this parameter is missing the default is to send the image to the next applet in the manner described above.

Chapter 6

Testing

This chapter aims to make 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. Note that the operations are not strictly equivalent as the underlying algorithms may differ but nevertheless should help provide a feel for the difference in performance.

Java as an interpreted language is pretty slow. This study showed that it is about five to ten times slower than C. Java as a compiled language, on the other hand, it is quite fast.

Below you will find a comparison of Java with C and compiled Java which should put everything into perspective.

6.1 The Comparison

6.1.1 Testing Conditions

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. Visilog has its own built-in counter (also in milliseconds). The applets where being individually run through the *appletviewer* program and all the images used where 256x256

Processor	**Clockspeed**	**SPECint92**	**SPECfp92**
microSparc-II	85MHz	65.3	53.1
microSparc-II	110MHz	78.6	65.3
Pentium 133	133MHz	147.5	109.6
Pentium 100	100MHz	100	81

Table 6.1: Sparc vs. Pentium

pixels large, greyscale.
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[1] compiler for Java.
One should bear in mind that running compiled Java on a different machine than the one C was tested on is liable to create surprising results. A P133 processor is approximately 1.8 times faster than the Sparc 4 processor, by comparing pure processing power in terms of floating point and integer operations (see table 6.1[2]). There are many other factors, however, which cannot be accounted for such as memory speed, caching strategy, etc.

6.1.2 Performance Comparison

Table 6.2 contains the summarised results of this comparative study.

- Rotations

  Two algorithms have been used for rotations. The first one can be thought of as the equivalent of Visilog's 'Nearby pixel' rotation and the second one as 'Four Neighbours' rotation although strictly speaking this is not quite the case. In both of these rotations Java performed adequately by being on average about a second slower than Visilog's algorithms.

[1]This type of compiler is not as efficient as a proper compiler would be. It is , however, the next best alternative.

[2]Figures taken from http://www.maths.lth.se/bengtl/horna/spectable.html

Compiled Java in the case of normal (simple) rotation did not make any difference which is surprising. Later investigation lead the author to believe that Netscape's compiler did not compile the code for simple rotation though the reason for this is not obvious. Microsoft's Internet Explorer, however, does and the performance is three times faster than Visilog's 'Nearby Pixel' rotation (on a Pentium 133).
Compiled shear rotation was indeed very fast, a fact which justifies the predictions of its author.

- Smoothing
  Gaussian (3x3) smoothing is approximately five times slower whereas Mean (3x3), Mean (5x5) and Median smoothing where all about ten times slower in interpreted Java.
  Three Median algorithms where implemented. The first one involved quick-sorting each nine-element array but that proved to be too slow as quick-sort is really not suitable for arrays less than fifty elements long. The second one replaced quick-sort for straight insertion-sort (adapted from [16]). This is the one included in table 6.2.
  A "fast" median finding algorithm adapted from [3] was implemented and was indeed on average half a second faster than the previous one but its performance was not great for large amounts of salt-pepper noise (greater than 8%). I suspect that this algorithm does not compute the true median value but rather an approximation.
  Compiled Java, in almost all cases here, performed faster that compiled C. The striking difference occurred in the case of Gaussian smoothing. This must be due to the algorithm employed by Visilog, since the one implemented for the purposes of this project involves approximately the same computational effort as the algorithm for mean (3x3) smoothing.

- Noise & Arithmetic
  Noise generation was almost instant due to quick-and-dirty programming and image arithmetic was just as fast. In addition of images Java is about

two times slower than C.
Compiled Java performs these operations instantly (down to milliseconds).

- Gamma & Thresholding
  Gamma correction takes a bit less than two seconds. Unfortunately, Visilog does not have a gamma correction transform for comparison.
  Thresholding is, surprisingly, almost as fast Visilog's.
  Compiled Java performs thresholding extremely fast (almost instantly) whereas gamma correction does not seem to have a striking difference in performance than that of interpreted Java. Again, however, Microsoft's compiler showed a two-fold improvement in performance over the (compiled) figure in table 6.2.

- Thinning
  Thinning is another disappointment as the Java program approximately 40 times slower than C. This can be attributed to the extremely naive nature of the algorithm implemented.
  Even compiled Java is five times slower than C which further reinforces the fact that the algorithm employed is by far a sub-optimal one.

- Convolution
  Convolution (3x3) takes, as was the case for Gaussian (3x3), about five times longer in interpreted Java.
  Compiled Java performs twice as fast as C. This suggests that Visilog's algorithm may be more general.

6.2 Correctness Of Algorithms

Although it would have been nice to compare the algorithms (at source code level) produced for this project with the ones used in Visilog, time constraints did not permit such a luxury.
Instead, "correctness" has been tested in terms of the visible output of each

Type of Operation	Java	Visilog (C)	Java with JIT
Rotations			
Normal	1.1-1.3	—	1.2-1.4
Shear	4.0-5.2	—	0.5-0.6
Nearby pixel	—	0.8-1.0	—
Four neighbours	—	3.0-3.2	—
Smoothing			
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 & Mean	5.0-8.0	—	0.3-0.4
Noise Generation			
Salt-Pepper Noise	0.2-0.3	—	instant
Gaussian Noise	0.2-0.3	—	instant
Gamma Correction			
Gamma Correction	1.0-2.0	—	0.7-0.8
Thresholding			
Thresholding	0.3-0.4	0.2-0.3	0.0-0.06
Thinning			
One Iteration	21.0-23.0	0.3-0.5	1.7-1.9
Convolution			
Laplacian (3x3)	4.5-5.5	0.8-1.0	0.4-0.5
Arithmetic			
Addition of images	0.2-0.3	0.1-0.2	instant

Table 6.2: Java vs. C in Seconds

operation.
Below the reader will find a set of paired images. The image on the left is the one produced by the relevant applet and the image on the right the one produced by Visilog except in figure 6.1, where XV[3] was used instead.
Although, in theory at least, no difference between the two images should be detectable, in practice this was not always the case.
In Figures 6.1, 6.2, 6.3, 6.4 and 6.5 you will find images which have been gamma corrected, convolved (Laplacian), mean smoothed (5x5 kernel), median smoothed and thinned.
A brief description of the findings follows. The original, unprocessed images, can be found in appendix A.

- The gamma transform applet produced the same results as the gamma correction option in XV. No obvious flaws were present.

- The Laplacian convolution kernel was applied in the original object of Figure 6.2. The output from the applet is more pronounced than the one from Visilog. The reason for this discrepancy is that Visilog handles differently intensity values which exceed the maximum. In the Java version if a value exceeds 255 then the output is 255. In Visilog a negative value is assigned instead.

- In mean smoothing by a 5x5 kernel the output from Visilog is different than the one from the corresponding applet due to the color-scheme adopted by each implementation (see previous bullet). The output from the equivalent operation in XV is exactly like the one produced by the applet. The reason the one from Visilog is displayed instead, is precisely to illustrate this difference in color handling.

- The original image of Figure 6.4 was corrupted by 25% salt-pepper noise before being passed to the median filters of Visilog and of the Smoothing applet.

[3]©1994 by John Bradley

The median smoothing operation employed by Visilog must bear a very close resemblance to the median-mean operator implemented for this project as the results they produce are identical. The "true" median algorithm (which involved sorting) was outperformed by Visilog's and by the median-mean operator.

- The thinning algorithm employed by Visilog produces the same result (after one iteration) as the one implemented in Java. The only difference is in terms of performance (discussed earlier in this chapter).

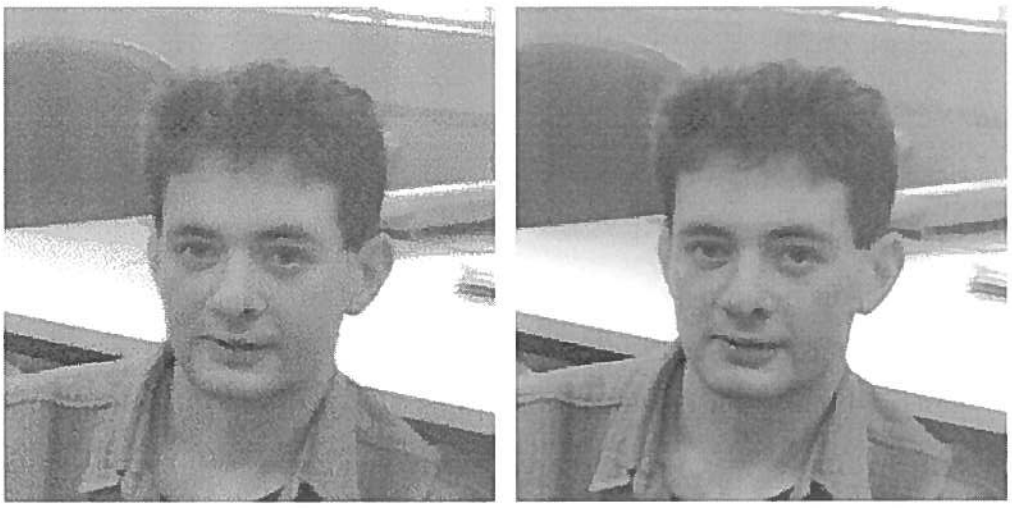


Figure 6.1: Gamma value = 3.3

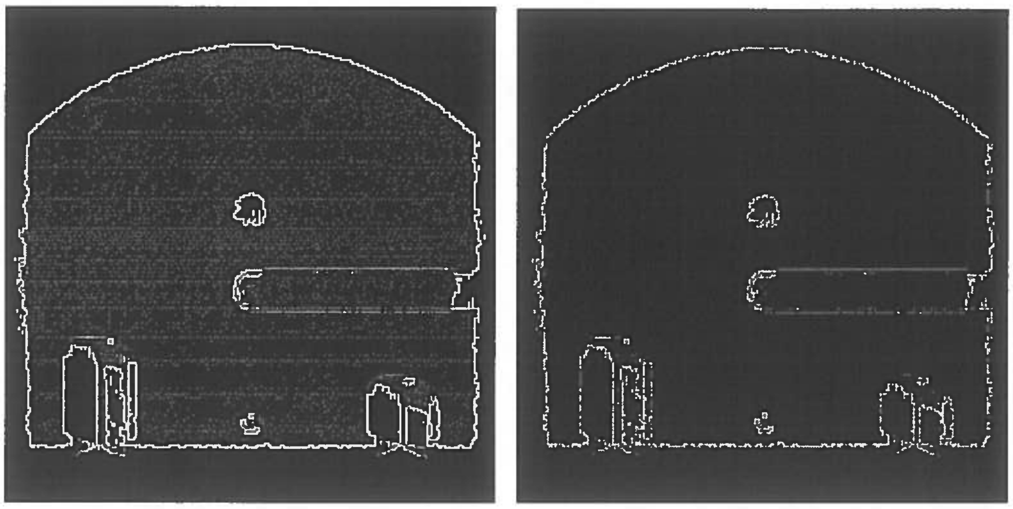


Figure 6.2: Laplacian Operator

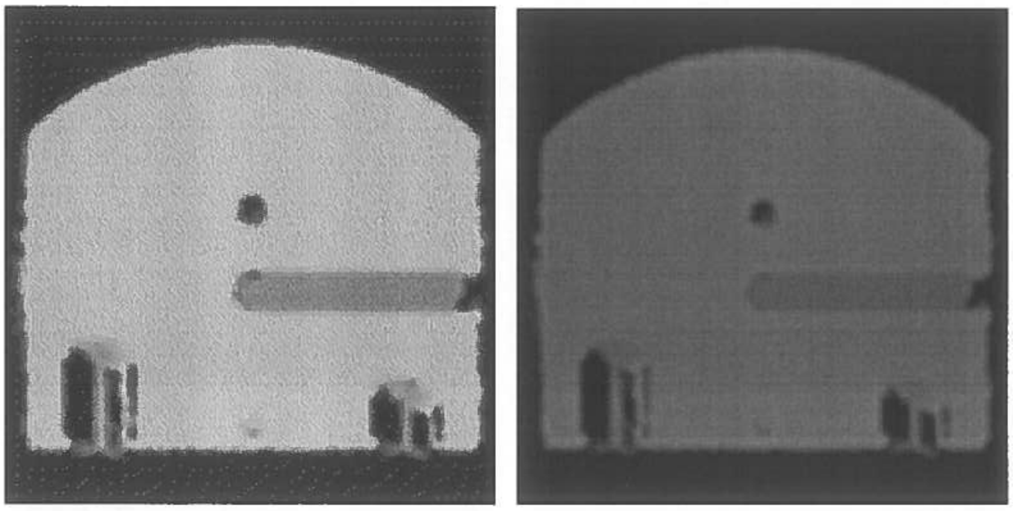


Figure 6.3: Mean Smoothing (5x5)

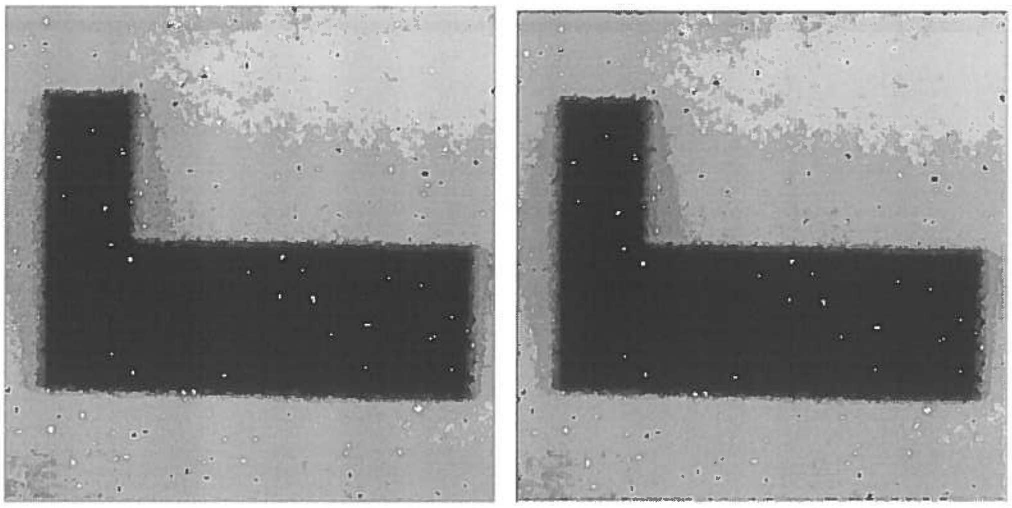


Figure 6.4: Median Smoothing

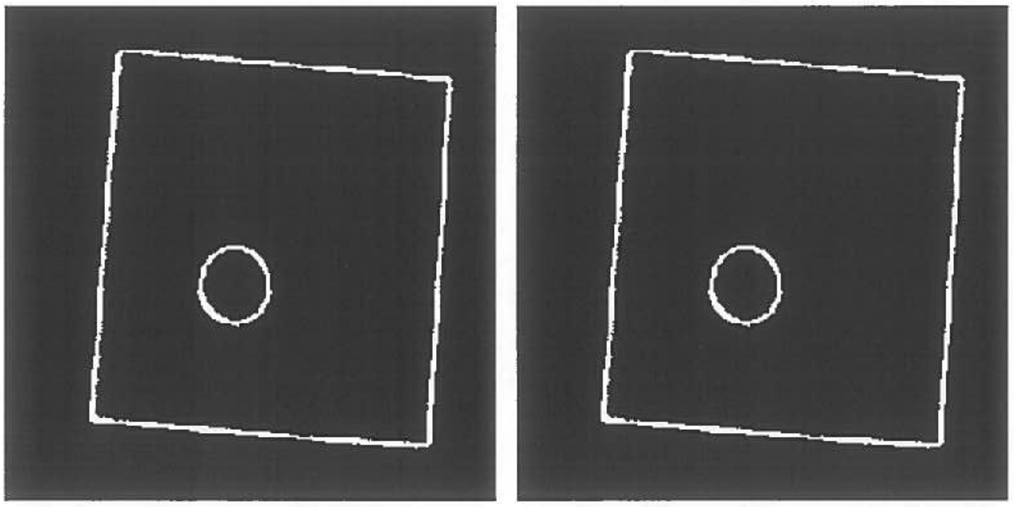


Figure 6.5: Thinning

Chapter 7

Conclusions—Future Work

The remainder of this report contains a brief account of personal opinions concerning Java, this project, and the future.

7.1 Java

The purpose of this project was to investigate the suitability, or otherwise, of Java for image processing. The question to this answer became reasonably apparent during the early stages of the project.
Java is suitable 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 the major reasons arises from the fact that Java comes with a plethora of built-in methods specifically for image manipulation. The facilities which are provided for image loading and pixel retrieval, for instance, are not to be found in other programming languages. Implementing a function to load a gif of jpg image in C would be a tedious and difficult task which may well take over two weeks to complete. 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.
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. According to my opinion Java has a lot going for it and not many against. It may soon be the language most people will be using.

7.2 The Project

This project has been both interesting, practical and fairly straight-forward. I was given the opportunity to experiment with a new tool which has a lot going for it in terms of user-friendliness as well as commercially.
The difficulties encountered where mainly due to the heavy use of threads during the early stages of the development of the applets. It turned out that the web browser the applets where being tested on [1] could easily become overloader which in turn caused it to halt for unreasonably long periods of time. These difficulties where eliminated by restricting the use of threads and by keeping the computation within them down to a minimum.
A general "template" for handling images has been adopted which is a useful preliminary step before the specific pixel manipulation can proceed. Down to algorithmic level there is scope for improvement. The shear rotation algorithm, for example, is not operating properly but that is probably due to my inability to deeply comprehend the paper upon which it is based. Furthermore, the thinning algorithm employed is naive and could be made to operate faster.

7.3 The Future

As far as future work is concerned, the sky is the limit. It would be gratifying to see applets similar to the ones seen earlier included in projects such as HIPR and it would be even more gratifying to find out that students are actually finding them useful.
More features (options) can be built upon the existing applets and new applets

[1]Netscape version 2.02

can be easily created provided the underlying algorithms are available.
If this project has contributed towards something useful I hope it is towards making the life of teachers a bit easier and the life of students a bit more interesting.

Bibliography

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

[2] Helmut Kopka & Patric W. Daly. *A Guide to LaTeX.* Addison-Wesley, 1993.

[3] Edited by Andrew S. Glassner. *Graphics Gems.* Academic Press Professional, 1988.

[4] Edited By Michael P. Ekstrom. *Digital Image Processing Techniques.* Academic Press, Inc., 1984.

[5] David Flanagan. *Java in a Nutshell.* O'Reilly & Associates Inc., 1996.

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

[7] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Second Edition, Prentice Hall, 1988.

[8] Laura Lemay and Charles L. Perkins. *Teach Yourself Java in 21 Days.* Sams.net Publishing, 1996.

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

[10] Andre Marion. *An Introduction to Image Processing.* Chapman and Hall, 1991.

[11] Vaclav Hlavac Milan Sonka and Roger Boyle. *Image Processing, Analysis and Machine Vision.* Chapman & Hall, Cambridge University Press, 1993.

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

[13] Robert Fisher, Simon Perkins, Ashley Walker and Eric Wolfart. *HIPR.* John Wiley & Sons Ltd, 1996.

[14] John C. Russ. *The Image Processing Handbook.* CRC Press Inc, 1995.

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

[16] W.T. Vetterling W.H. Press, S.A. Teukolsky and B.P. Flannery. *Numerical Recipes in C.* Cambridge University Press, 1992.

Appendix A

Images Used

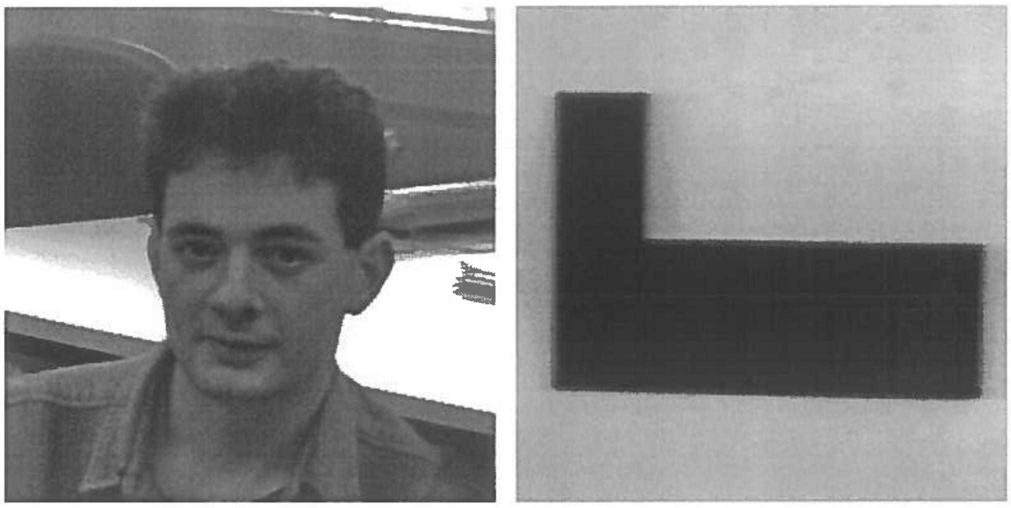


Figure A.1: Simon Perkins and the Famous L-Object

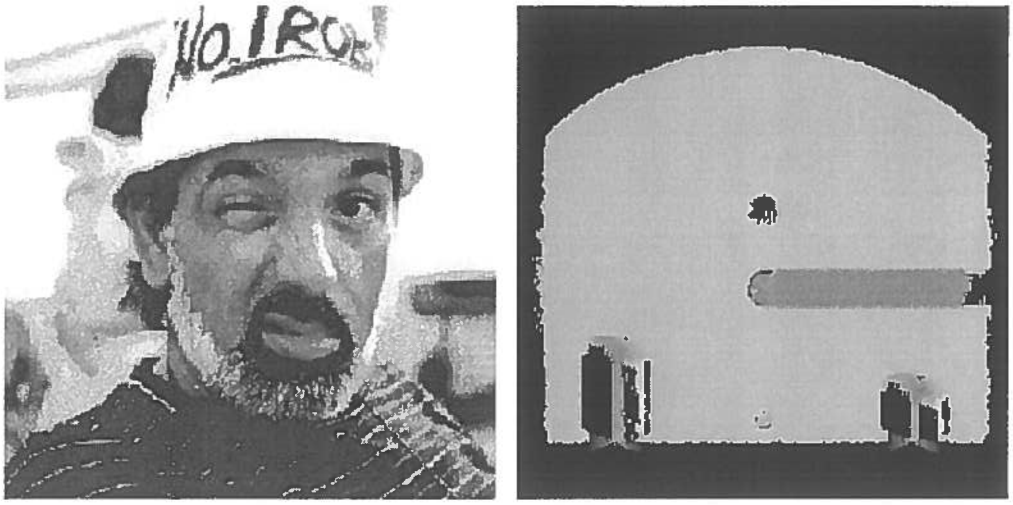


Figure A.2: Papa and the UFO

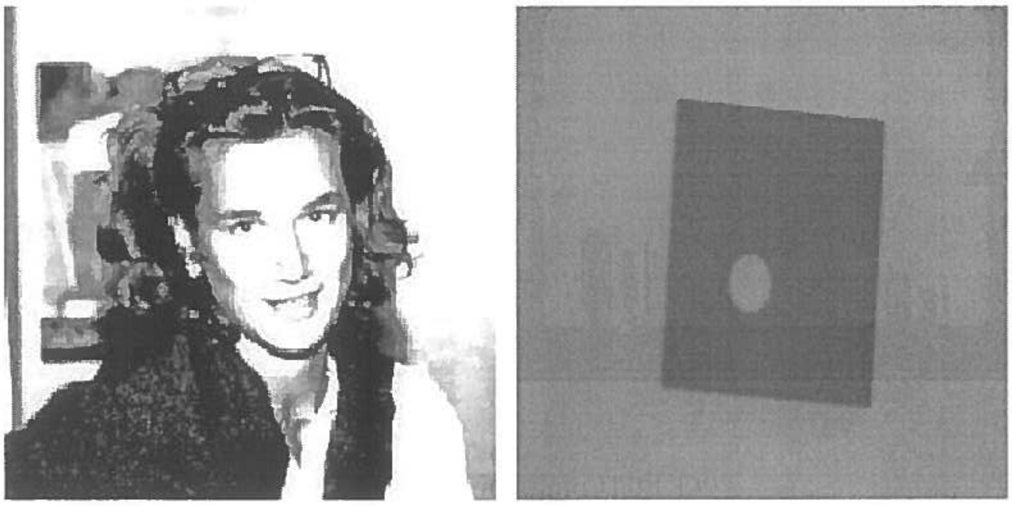


Figure A.3: Olga and the Holy Square

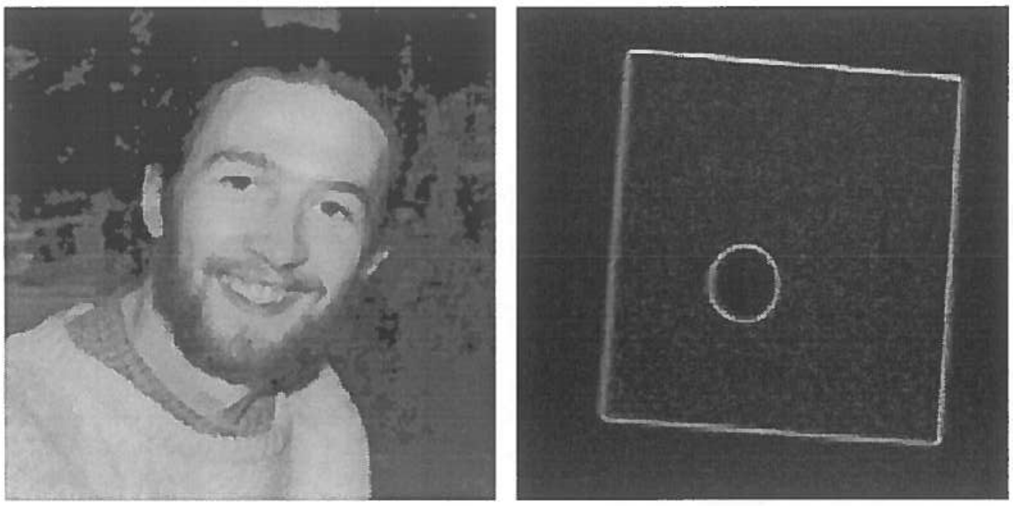


Figure A.4: Andrew and the Dark Holy Square

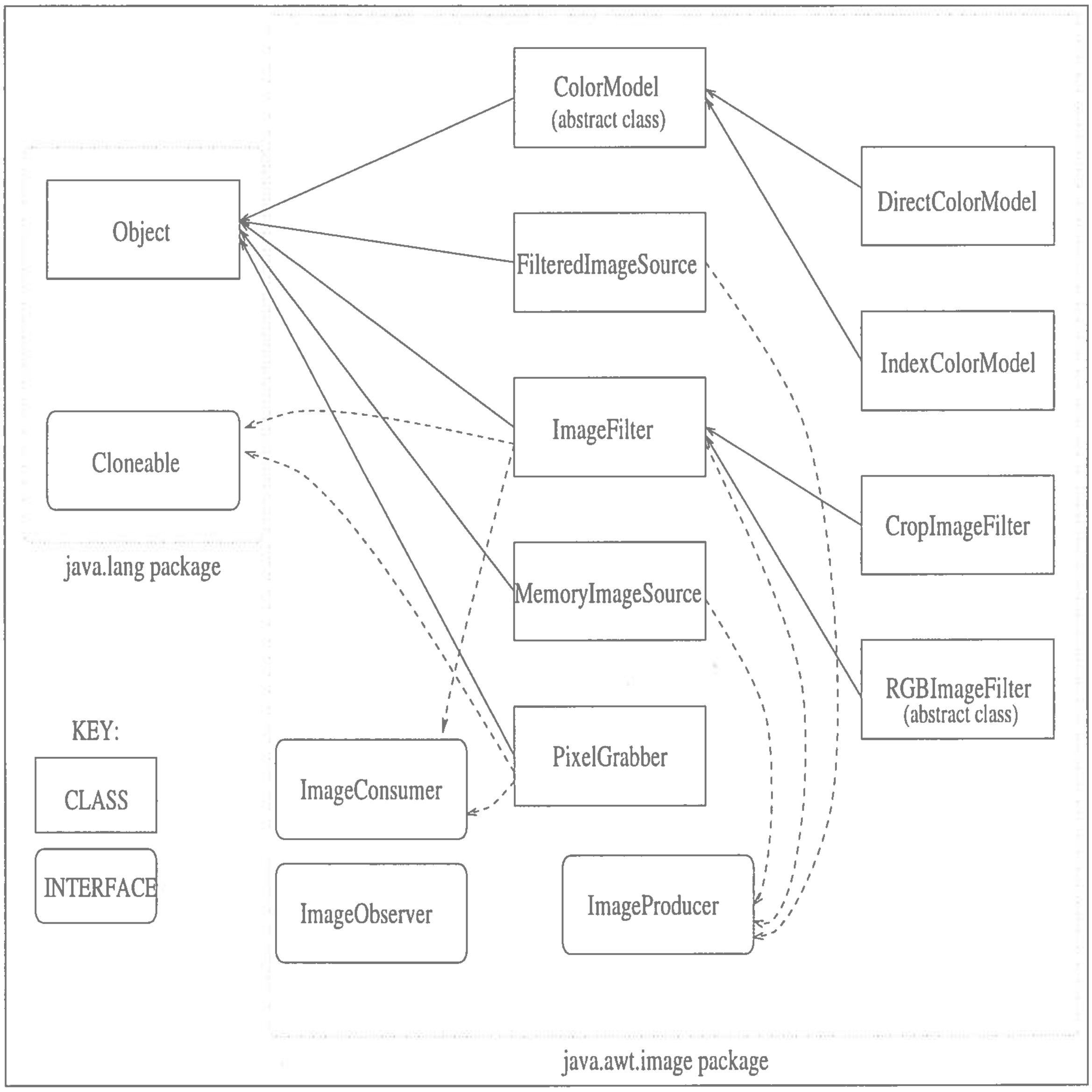


Figure A.5: A diagram of the java.awt.image Package