EIKONA

Digital image processing package

Thessaloniki 1997

Manual, Part I

EIKONA for Windows

Users guide

Version 4.0

The JPEG input/output routines were based in part on the work of the Independent JPEG Group.

The GIF, BMP, TGA input/output routines were based in part on the PBMPLUS toolkit.

The TIFF input/output routines were based in part on the Silicon Graphics LibTIFF library by Sam Leffler.

The scanner interface was based in part on the TWAIN Toolkit, Release 1.6. The TWAIN toolkit is distributed as is. The developer and the distributors of the TWAIN toolkit expressly disclaim all implied, express or statutory warranties including, without limitation, the implied warranties of merchantability, non-infringement of third party rights and fitness for a particular purpose. Neither the developers nor the distributors will be liable for damages, whether direct, indirect, special, incidental, or consequential as a result of the reproduction, modification, distribution or other use of the TWAIN toolkit.

Distributor :

Contents

1

EIKONA for Windows

1.1 Introduction

EIKONA for Windows is a powerful, but yet simple to use, digital image processing software package that runs under Microsoft Windows 95 and implements over 150 image processing routines in the following areas:

- image display, scanning and printing
- image thresholding, clipping
- addition, subtraction and multiplication of images
- and, or, xor bit-level operations between images
- addition/multiplication of an image by a constant
- various image noise generators
- two dimensional filters including adaptive and nonlinear filters
- histogram and cdf histogram computation and equalization
- matrix histogram and cdf matrix histogram computation
- image enhancement and sharpening
- region segmentation
- edge detection
- morphological filters
- image transforms
- color coordinate transformations
- image mosaicing, registration and watermarking

EIKONA user interface is based completely on pull-down menus, dialog boxes and common Windows 95 user interface elements. As a consequence, it is extremely easy to operate even for users not very familiar with image processing. Images are stored on image buffers.All that is needed to apply an image processing function to an image is to specify the source and destination image buffer and the related

parameters. Furthermore, implemented functions are grouped into categories according to the type of operation that they carry out, in order to facilitate the search for a specific task. The user can choose the image region where processing is to be performed. Multiple images can be displayed on the screen at the same time, a valuable feature when comparing the results of different processing functions on the same image. Being a pure 32-bit application, EIKONA fully exploits the Windows 95 flat memory model allowing efficient handling of big images.

EIKONA supports the following file formats : TGA, TIFF, uncompressed BMP, JPEG, GIF and binary (raw) images. Conversion from one image file format to another is also possible.

1.2 Hardware Requirements

EIKONA for Windows can run on any 386/486/Pentium computer under Microsoft Windows 95. However, since many image processing applications are rather time consuming and since images, especially color images, require large amounts of memory, a minimum configuration of a 386 machine with at least 8MByte of RAM is recommended. Also, in order to take advantage of the image display capabilities of Eikona for Windows, a color monitor with a SUPER VGA card capable of displaying at least 32K colors is required.

1.3 Installation

To install EIKONA for Windows insert the setup disk, select **Run** from the **Start** menu and type `a:setup`. You will be prompted for a destination directory on your hard drive. The setup program will copy all the relevant files to this directory and create a new Windows program group on your taskbar.

1.4 Overview

The central concept in EIKONA is that of a *buffer*. EIKONA supports four types of buffers that are used by various image processing functions:

BW buffers are used for storing grayscale images. Essentially they are two-dimensional arrays of unsigned characters. Their format is described in detail in [PIT93].

Color buffers are used for storing color images. They are nothing more than

three BW buffers (one for each color channel) that EIKONA groups together. However the user can view and otherwise act on each channel separately as if it was and ordinary BW buffer. Furthermore three BW buffers can be merged into one color buffer.

Float matrices are used to store the results of some functions implemented in EIKONA (most notable examples being the Fast Fourier Transform and the Discrete Cosine Transform).

Color processing matrices used mainly for color space transformations and other color image processing functions.

EIKONA can allocate as many buffers as required the only limit being the memory capacity of the computer. Buffer dimensions are either specified by the user or determined by EIKONA according to the operation.

EIKONA also allocates 2 one-dimensional float buffers (referred to as vectors) that are used internally in some operations e.g. histogram calculation or matrix histogram calculation. The dimension of these vectors defaults to 256 elements (although it can be user-defined as described in the next section).Their contents can be saved to disk in decimal format in order to be examined.

1.5 Initialization information - The PAR files

The EIKONA.PAR file Certain internal variables of EIKONA can be customized by the user. These variables control the vector size, the maximum window size for image filtering operations and the reference values for color space transformations. During it's initialization phase EIKONA reads the file `EIKONA.PAR` from the directory where it was started.The user can modify this file to define these variables. Typical contents of this file may be the following:

```
WindowXSize=15
WindowYSize=15
VectorSize=256
Xo_reference=1.0
Yo_reference=1.0
Zo_reference=1.0
```

These `EIKONA.PAR` file contents instruct EIKONA the maximal allowable window size in image filtering operations is 15×15, the vector size is 256 points and that the reference values for color space transformations are 1.0. EIKONA is not sensitive to the order of these definitions in the `EIKONA.PAR` file but it requires every one of them to be in a separate line.

The WIN_x.PAR files Various filtering operations require a float or integer coefficient mask. These masks, which are read by EIKONA without user

intervention, are contained in files `WIN_F.PAR` and `WIN_UC.PAR` that must reside in the directory EIKONA. The user must edit these files prior to applying the filtering operation and fill in the filter coefficients. The contents of the `WIN_F.PAR` file could be the following:

```
3
3
0.11111
0.11111
0.11111
0.11111
0.11111
0.11111
0.11111
0.11111
0.11111
```

The first two numbers show the column and row number of the floating point mask (3×3 in this case). Nine floating point numbers follow that correspond to the mask elements written in a row-wise manner. This mask, when used with convolution corresponds to the moving average filter. In addition, this mask is used in User Defined Color Transformation function. The contents of the `WIN_UC.PAR` file could be the following:

```
3
3
0
1
0
1
1
1
0
1
0
```

The first two numbers show the column and row number of the integer mask (3×3 in this case). Nine integer numbers follow that correspond to the mask elements written in a row-wise manner.

1.6 Using EIKONA for Windows

To run EIKONA for Windows double-click the EIKONA icon. The EIKONA main window appears at your screen. The menu that appears on the top side

of the window has six menu items i.e. **File, Black and White, Color, Modules, Windows, Help**. By clicking each of them a pull-down menu appears that contains several options. **File** menu contains file input/output operations as well as other general tasks like file printing, buffer manipulation etc. **Black and White** menu contains all the image processing functions operating on Grayscale images. **Color** menu lists all color image functions provided by EIKONA. The **Modules** menu is used by external modules that extend EIKONA's capabilities. The **Windows** menu provides basic window management functions. Finally **Help** provides help on various image processing routines. In the following each of the six menus will be presented in more detail. It must be noted that EIKONA routines have extensive error handling capabilities. If an error is detected the job is not executed and a relevant error message is displayed. In this case the user has to check again the parameters he fetches to the routine.

1.6.1 File

Open: Load an image file from disk. Supported image formats include Binary (raw), TGA (Color and Black and White, certain types), TIFF, BMP (uncompressed), JPEG and GIF. Raw images must be stored row-wise, the first row being the top row.

Open Matrix: Load a matrix file from disk. The user specifies the matrix dimensions.

Save: Save images and matrices to disk[1]. Supported formats are : Binary (raw), TIFF, TGA and JPEG. Binary color images must be saved in three different files, one for each color component. Thumbnail images can also be saved in JPEG format.

Acquire: Scan an image from a scanner compatible with the TWAIN interface or select a TWAIN source.

Hide ROI: This is a toggle (on/off) option. When it is on the region of interest is *not* displayed on the active window. If the active window has scroll bars then the user should zoom out the window if he would like to see the ROI.

Display Control Window : Toggles the Control window on or off.

Write pixel: The next click of the left mouse button after Write pixel was selected will place the value 255 as pixel value at the pixel pointed by the cursor. This function can be used to put "seeds" in image buffers.

[1]The **Save** option is only available through the EIKONA Save Module.

Print: Print a binary image.

Dump: Write the contents of Region of Interest of a BW buffer to a file in decimal format.

Dump Matrix: Write the contents of the specified float buffer to a file in decimal format.

Dump signal: Write the contents of a one-dimensional buffer (vector) to a file in decimal format.

Dump Matrix histogram: Write the contents of a two-dimensional buffer (vector) to a file in decimal format. The vector[0] buffer contains the X values of matrix histogram and the vector[1] buffer contains the corresponding Y values.

Buffers: General buffer management functions.

Exit: Exit EIKONA.

1.6.2 Black and White

This menu contains all routines for the processing, analysis and display of black and white images. In most cases, the processing routines have one source BW image buffer, one destination BW image buffer and a number of parameters. The user can either specify an existing buffer as the destination buffer or ask for the creation of a new buffer. Processing is performed only within the region of interest (ROI). ROI definition is described in the next chapter. The display is automatically updated to reflect the result of the operation.

This part of the manual does not contain detailed description of the various routines, since they are described in detail in the manual of the EIKONA library.

Basic: Sub-menu that contains a number of basic image operations. These include clearing the contents of an BW buffer, copying the contents of a buffer to another buffer, bit-level operations between images (and, or, xor), adding a constant to an image, multiplying an image by a constant, adding/subtracting two images, transforming an image to matrix and a matrix to image with normalization or truncation, image halftoning etc.

Processing: Sub-menu that includes negation of an image, image sharpening, image zoom/decimation and also additive and multiplicative noise generators (Gaussian, Uniform, Laplacian) that can be used to corrupt an image for simulation purposes.

Analysis: Edge detection algorithms (Sobel, Laplace, etc.), line and point detection functions, various algorithms for edge following and related algorithms like Direct and Inverse Hough transform. Region segmentation algorithms including thresholding and counting. Texture description and shape analysis algorithms, along with thinning and pyramid methods. Finally, image and matrix histogram, Cdf Histogram and Cdf Matrix Histogram are also included in this menu.

Transforms: this sub-menu includes the well-known Fast Fourier Transform (FFT) for one and two dimensions and the Discrete Cosine Transform. Also Power Spectrum Density estimation, image convolution etc.

Filtering: Sub-menu containing some of the most widely used filters in the Digital Image Processing. These include Histogram equalization, moving average, median, minimum, rank filter, L-filter and many more.

Nonlinear filtering: Some more nonlinear operators like morphological operators for grayscale and binary images (opening-closing, erosion-dilation), Signal Adaptive Median filter, hybrid, multistage, weighted, separable, median filters, homomorphic filter, harmonic filter etc.

Merge Channels: Merge three BW buffers in a color buffer.

Display: Display grayscale images. The user specifies the buffer to be displayed. The display window shows the number of the displayed buffer. It can be moved around and it can be closed as any other window.

1.6.3 Color

This menu contains all routines for the processing, analysis and display of color images. In most cases, the processing routines have one color image buffer, one destination color image buffer and a number of parameters. The user can either specify an existing buffer as the destination buffer or ask for the creation of a new buffer. Processing is performed only within the region of interest (ROI). ROI definition is described in the next chapter. The display is automatically updated to reflect the result of the operation.

This part of the manual does not contain a detailed description of the various routines, since they are described in detail in the manual of the EIKONA library.

Basic: Sub-menu containing basic color image operations like bit-level operations (and, or etc.), buffer copy and clear operations, addition and multiplication of the three RGB components of an image with the same constant etc.

Processing: Noise generators for the three components of a color image, color image filters like marginal median and marginal minimum and also erosion-dilation operators are included in this sub-menu.

Analysis: Color image edge, line and point detectors.

Color Representation: This sub-menu contains a number of color representation system transformations. Many of the color coordinate systems that are included, require float color coordinates. Therefore float buffers must be allocated for such transforms.

User Defined Transformation: User Defined Color Transformation. The user specifies the mask of the transformation in the `WIN_F.PAR` file.

Preview, Display: Both options display a color image buffer and have the same effect for output devices that can display more than 256 colors. However the two options are different for 256 color displays. Preview allows the user to view a color image fast without the best color quality. On the other hand, Display takes more time to be executed but it obtains a better image display by constructing a 256-color palette according to the histograms of the three color components.

1.6.4 Modules

This menu is used by additional modules that extend EIKONA's capabilities. Every option provided by each such module appears under this menu. Three modules are currently available :

EIKONA FOR ARTS with options for automatic or manual mosaicing of a set of images and simple or combined registration of two images.

WATERMARKS for casting and detection of digital watermarks on an image.

CRACK RESTORATION for restoring old paintings from cracks.

1.6.5 Windows

This menu provides provides basic function for managing open windows.

1.6.6 Help

This version of EIKONA does not support On-line Help.

2

Users guide

2.1 Basic operations

2.1.1 Opening an image

This section will describe how to load and display grayscale as well as color images. We assume that we are already in the environment of EIKONA for Windows. We also assume that the current graphics display configuration can display at least 256 colors. However, the display mode of 16 million colors that is currently available on most super VGA cards is highly recommended. A moderate alternative is to use display modes with 32 K colors. Check your Windows setup to ensure that you have installed the proper graphics mode.

Let us suppose that you have a grayscale image of dimensions 256×256 named BABOON.JPG in JPEG format on your current directory. This image can be loaded as follows:

- Choose option **File** by mouse.
- Choose option **Open**.
- A common *Open File* dialog box appears on screen that can be used to select the image to be opened (we assume that the user has at least some experience in using these common Windows 95 user interface elements). Double-click on the name BABOON.JPG.

EIKONA will load and display the image automatically. The displayed image is shown in Figure 2.1.1.

The same procedure can be used for loading images in BMP, TARGA, TIFF or GIF format. For these formats, EIKONA automatically determines the dimensions of the image and it's type (grayscale or color), creates a suitable buffer and displays the image without further user intervention.

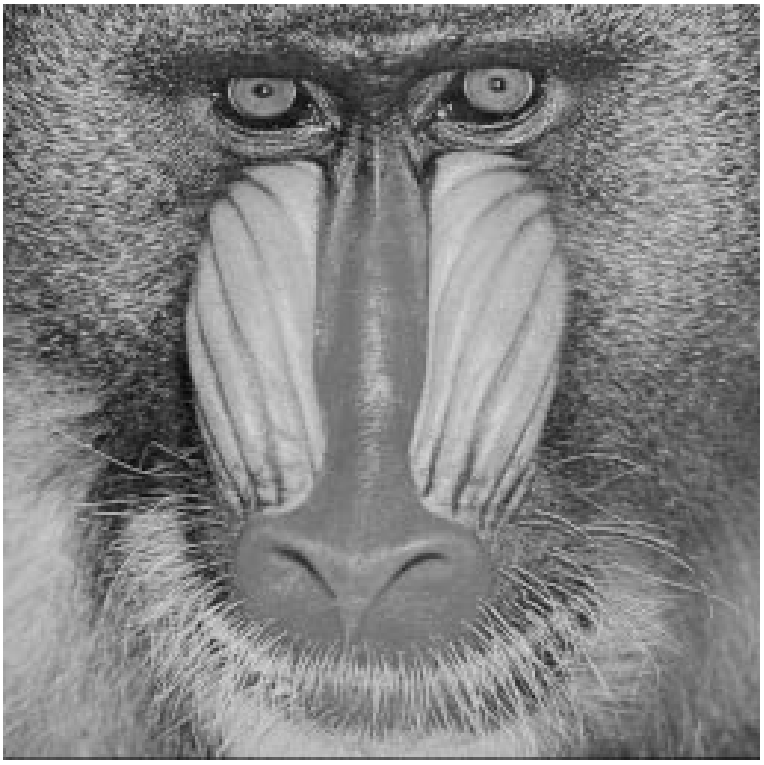

Figure 2.1.1 Grayscale image BABOON.

A slightly different procedure must be used in order to open an image in binary (raw) format. These images are stored in a row-wise manner. Each pixel is stored in one byte, the first byte in the file corresponding to the upper left corner of the image. EIKONA cannot determine the dimensions of such images (it can only make a rough guess) so it displays the *Open Raw* dialog box. The *Width* and *Height* fields of this dialog box display EIKONA's guess about the image's dimensions. The user can either enter the dimensions directly or use the *Guess* button to allow EIKONA to take another guess. A *Swap* button is also provided in case EIKONA has guessed the numbers correctly but in the wrong order. As an example assuming that you have the same image in raw format named BABOON.RAW the procedure to load it is the following:

- Choose **File** from the menu bar.
- Choose **Open**
- Double-click on BABOON.RAW
- The *Open Raw* dialog box appears. For this image the dimensions should be correct (EIKONA always guesses correctly for square images) so click OK.

Windows opened by EIKONA can be moved, minimized or closed as any other window. If the image width / height (or both the width and height) exceeds the screen's width / height, scrollbars appear on the image display window to allow the

user to view parts of the image not currently on display. Multiple windows can be open and the user can switch between them using the CTRL-TAB combination.

2.1.2 Defining the Region Of Interest

The majority of image processing functions implemented by EIKONA are not applied on the whole image but rather on a rectangular part of it called the *Region of Interest*(ROI). Every window opened by EIKONA has an associated ROI which by default covers the entire buffer displayed by the window. The ROI is shown as a rolling dashed rectangle in the currently active window. The user can easily mark another ROI using the following procedure :

- Activate the window on which you want to define the ROI. This can be done either by left-clicking on a point of the window or by pressing CTRL-TAB repeatedly until the desired window becomes active.
- Move the mouse at one corner of the desired ROI, press and hold down the right mouse button.
- Drag the selection rectangle that appears to cover the desired area. If the window has scrollbars moving out of it will cause the scrollbars to follow.
- Release the right mouse button.

The user can also make adjustments to specific edges of the ROI. Moving the mouse over a ROI edge or corner changes the cursor shape appropriately. By pressing and holding down the right mouse button the user can refine his selection (the process is similar to that of resizing windows). Finally, the user can restore the ROI to cover the entire buffer displayed by the active window either by pressing CTRL-A or through the *Control* window as described in the next section.

2.1.3 The Control Window

A number of useful and frequent operations can be instantly accessed through the *Control* window. This window is displayed in the upper-right corner of the screen when EIKONA starts-up. Figure 2.1.2 shows the control window.

Going from top to bottom we can see :

The ROI information section where the coordinates and dimensions of the ROI rectangle for the currently active window are displayed.

The 'Select Whole Image' button The user can click on this button to make ROI cover the entire buffer displayed by the active window.

The cursor position section where the coordinates of the cursor are displayed.

The color space selection buttons the color space in which the color values will be displayed.

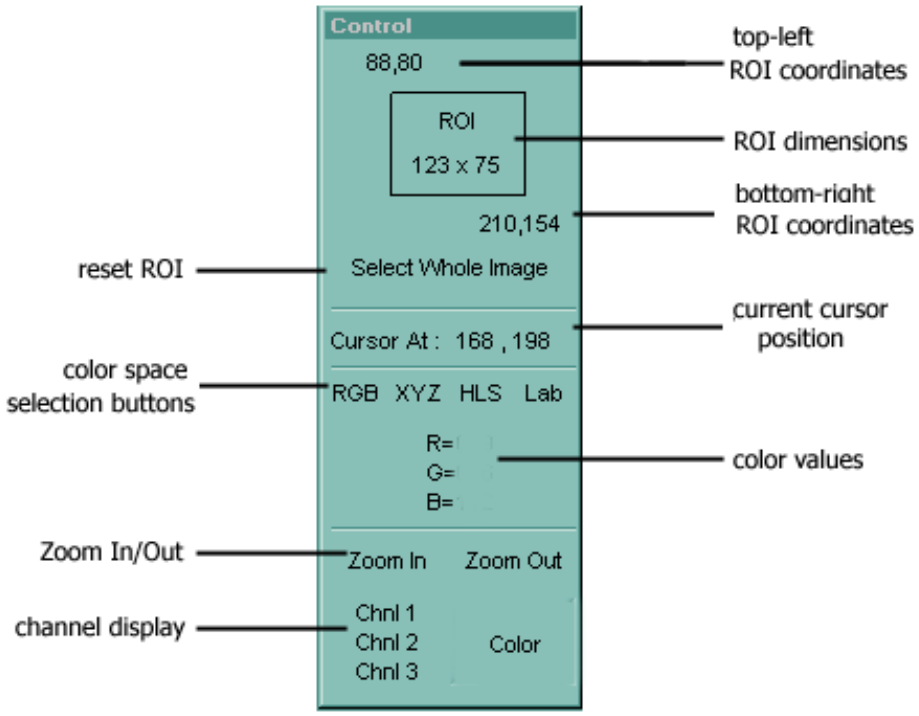

Figure 2.1.2 The Control window

The color values section where the color values for the pixel at the current cursor position are shown.

The Zoom In/Out buttons that control the zooming factor for the active window.

The channel display buttons that control the channel displayed by the active window. These buttons are enabled only when the active window displays a color image.

2.1.4 Buffer management

Buffer management is done in EIKONA through the option **File, Buffers** which brings up the *Buffers* dialog box depicted in Figure 2.1.3. The *Buffer Type* combobox is used to select the type of buffers that are displayed in *Buffer List*. The thumbnail at the upper right corner of the dialog box displays the buffer currently selected in the *Buffer List* listbox. The dimensions and memory used by this buffer are displayed just below the thumbnail. The user can click *New Buffer* to create a new buffer of the type currently selected in *Buffer Type*. Clicking on *Delete Buffer* the currently selected buffer is destroyed (the user is asked to confirm the deletion if the currently selected buffer is part of a multichannel buffer).

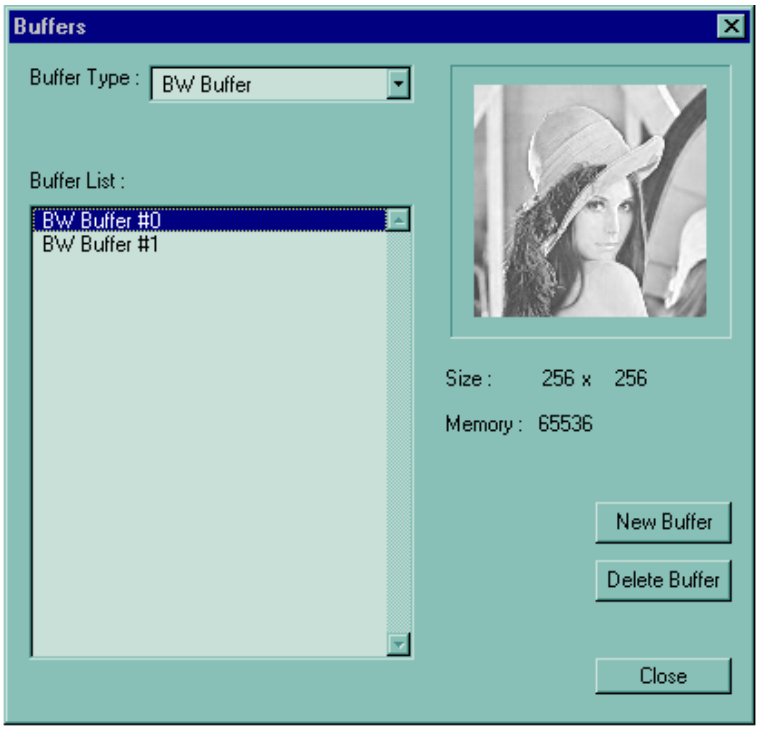

Figure 2.1.3 The buffer control dialog.

2.2 Examples of grayscale image processing and analysis

Now you are ready to perform some image processing operations. A basic knowledge of digital image processing operations is recommended before proceeding to the rest of the manual. However, this is not a must, especially for simple image processing operations used in producing video effects. For more advanced processing and analysis, it is recommended to refer to the book I. Pitas, *Digital image processing algorithms*, Prentice Hall, 1993. Basic digital image processing literature can be found in the section REFERENCES of this manual. If you have any question on the commands and their operation refer to the second part of this manual by command name. In the rest of this chapter a sequence of menu item selections that have to be performed by user in order to perform a certain task will be given as a sequence of commands in bold, separated by commas. For example the sequence **Black and white, Basic, Copy** means that the user must select **Black and white** from the main menu, then click on **Basic** from the pull-down menu that will show up and finally select **Copy**. The first example is given in more detail and serves as a guide to using EIKONA. Now, you are ready to explore the many exciting facets of image processing.

2.2.1 Basic image processing

Let us suppose that that the image file LENNA256.DAT of size 256×256 has already been loaded as described in the previous section. We assume that no other image buffers are currently allocated.

To brighten LENNA, we can add a suitable constant to the grayscale level of every point. This can be done by the sequence **Black and white, Basic, Addc**. The dialog box that appears is shown in Figure 2.2.1. This dialog box is very

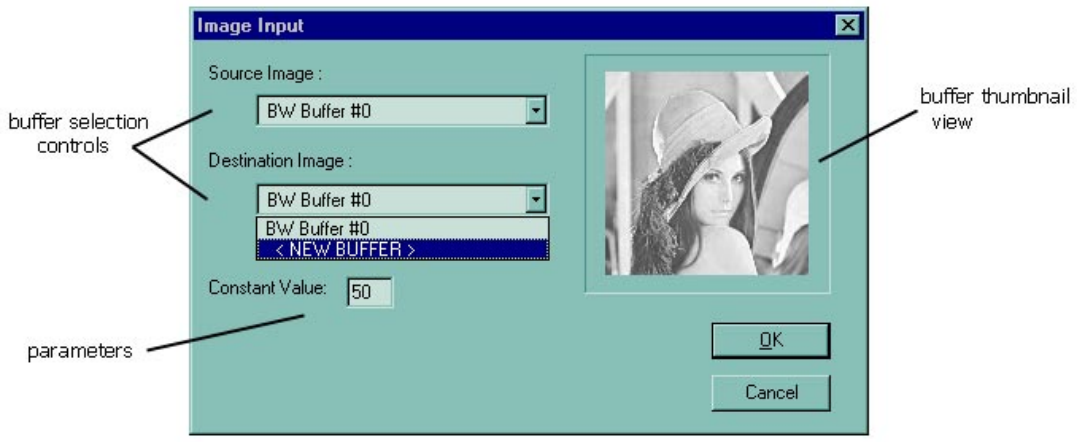

Figure 2.2.1 The Add Constant dialog box.

similar to most dialog boxes used in EIKONA for user input, so we'll describe it in some detail. As can be seen from Figure 2.2.1 it can be divided in three areas.

- The **buffer selection controls** are used to specify the source and destination buffers for the operation. EIKONA takes great care to fill these combo-boxes only with values valid for the requested operation. The source buffer control suggests the buffer displayed by the active window. The destination buffer control is filled only with buffers that have suitable dimensions to hold the function's result. For most functions (as in this example) there is also an option to create a new buffer for the result.
- The **buffer thumbnail view** allows the user to see the contents of the buffer he selects from the combo-boxes. This is a valuable feature especially when many buffers are allocated but not currently displayed.
- The **parameter edit fields** where the user can type in the values for various parameters the routine may require.

There are also the usual *OK* and *Cancel* buttons that allow the user to complete or cancel the operation.

Returning to our example, by choosing the buffer were LENNA256.DAT is loaded as the source buffer, requesting a new buffer for output and entering 50 for the constant to be added we get the new brightened image of LENNA. Both images are shown in Figure 2.2.2.

Figure 2.2.2 LENNA and its brightened image.

Let us suppose that we want to compute the negative of LENNA and store it on the same buffer we stored the brightened LENNA. This can be done by choosing the functions **Black and white, Processing, Neg** and selecting LENNA and it's brightened image as the source and destination buffers. At the end of the operation the window displaying the brightened LENNA is repainted and shows the negative of image LENNA. The negative image is shown in Figure 2.2.3.

Figure 2.2.3 LENNA and its negative image.

If we want to threshold an image we can follow the sequence **Black and white, Analysis, Region Segmentation, Threshold**. As an example, if image LENNA is

currently loaded in BW buffer 0, we can choose BW buffer 0 and <New Buffer> from the source and destination controls, respectively, and 150 as the threshold level. The original and the thresholded version of LENNA are shown in Figure 2.2.4.

Figure 2.2.4 LENNA and its thresholded image.

Let us suppose that images LENNA and BABOON are loaded in BW buffers 0, 1 respectively.If you want to mix the two images of buffers 0, 1 (BABOON and LENNA) and store the result on a new buffer, you choose the option **Black and white, Basic, Mix**. You choose buffers 0,1 as source buffers , <New Buffer> as destination buffer and use 0.5, 0.5 as mixing percentages of the two images. The mix of the two images is depicted in Figure 2.2.5.

2.2.2 Image transforms

The Transforms option of the **Black and White** menu contains various implementations of algorithms concerning transforms of images. The two dimensional FFT is a typical example which we will examine. We assume that the file LENNA256.DAT is already loaded in BW buffer 0 and that no other buffers are allocated. We choose **Black and White,Transforms, 2-D FFT** and a message box appears informing as that at least two float buffers are required for this operation. After allocating two float buffers of dimensions 256×256 through the menu **File, Buffers** and repeating the above selection a dialog box appears where we can choose BW buffer 0 as the source and the just allocated float buffers for the real and imaginary parts of the Fourier Transform. After the conclusion of the operation we can test the obtained results through the option **Black and White, Transforms, Inverse 2-D FFT** and see if the resulting image resembles our original one. We could also select **Black and white, Basic, Matrix to Image** to have either the real or the imaginary

Figure 2.2.5 Result of mixing of LENNA with BABOON.

part of the DFT, converted to image format, in order for us to view it. As an example, assuming that the imaginary part of the transform was stored in Float buffer 1 than selecting **Black and white, Basic, Matrix to Image** and selecting Float buffer 1 and <New Buffer> as the source and destination buffers, we can convert the imaginary part of the FFT of LENNA to image format as shown in Figure 2.2.6. We must note, though, that by converting a float buffer to an image buffer, effectively the float buffer is clipped between the values of 0 and 255. Thus, the image might not prove to be an accurate representation of the matrix, in this case.

The Periodogram, AR PSD and Blackman Tukey PSD are used for Power Spectral Density estimation of images. AR PSD estimation can be easily implemented in EIKONA. Assuming that LENNA is loaded in BW buffer 0 we can get an AR PSD estimate, by selecting **Black and white, Transforms, AR PSD** and entering 0 as the source image buffer and <New Buffer> as the destination buffer. If we enter 3, 3, and 3 as the AR model left x, right x and y coefficient window sizes, then the result will resemble the one of Figure 2.2.7.

If the grayscale image LENNA256.DAT is loaded in image buffer 0, we can get an estimate of the Power Spectral Density by selecting the option **Black and White, Transforms, Periodogram** and entering 0 and <New Buffer> as the source and destination image, respectively. The periodogram of LENNA is depicted in Figure 2.2.8. The similarity between the result of this spectral estimation method and of the one of the AR PSD estimate (shown in Figure 2.2.7) is obvious.

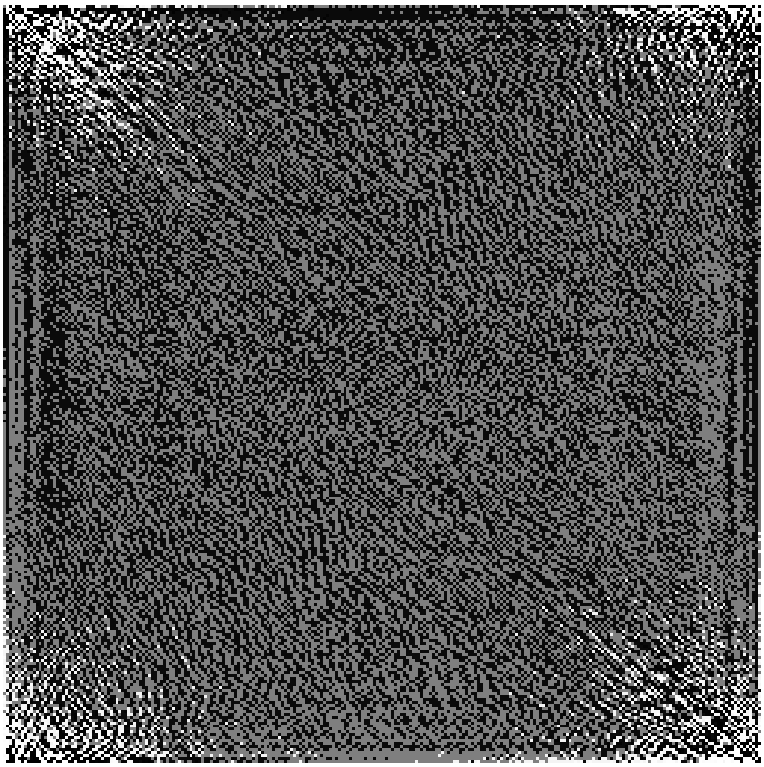

Figure 2.2.6 Imaginary part of the 2-D FFT of LENNA.

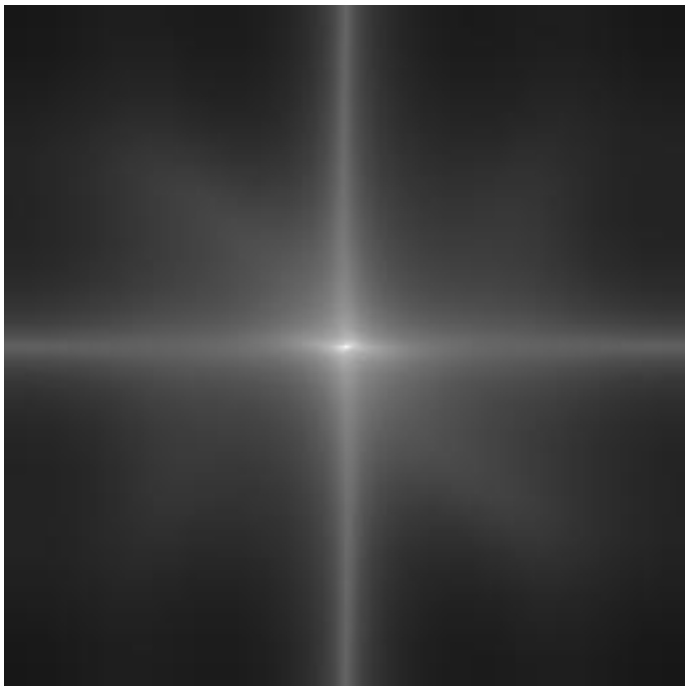

Figure 2.2.7 AR PSD estimate of LENNA.

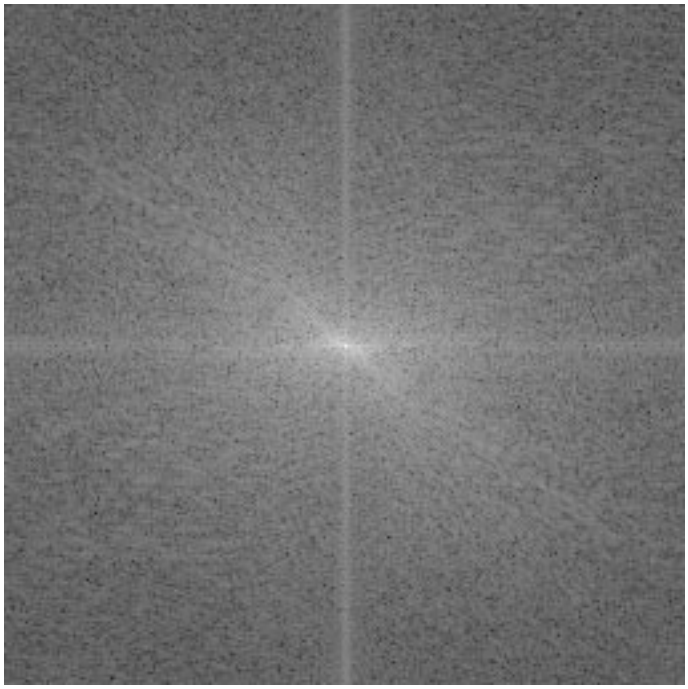

Figure 2.2.8 Periodogram of LENNA.

2.2.3 Digital image filtering and enhancement

EIKONA can be used as a tool to filter out noise from a digital image. Let's take for example the case of an image which is corrupted by impulsive ("salt and pepper") noise [PIT90]. We will assume that the image LENNA is loaded in image buffer 0. You can create a corrupted version of this image by following the sequence **Black and white, Processing, Impulsive**. In the window that appears, you can enter 0 and <New Buffer> as the source and the destination image, respectively. A commonly used set of values for the noise probability and the minimum and maximum spike values, are 0.1, 0 and 255, respectively. The result can be seen on Figure 2.2.9.

Figure 2.2.9 LENNA and its corrupted version.

Now, you can easily filter out the noise from the corrupted version of LENNA, by utilizing a median filter [PIT90]. This can be done, easily, by selecting **Black and white, Filtering, Median**. If the corrupted version of LENNA was stored in BW buffer 1 than you should enter 1 and <New Buffer> as the source and destination buffers, respectively. The filtering window size is also required. A 3×3 window is usually utilized, so you can fill-in 3, 3 in the remaining window fields. The result is shown in Figure 2.2.10.

As you can see, the filtered image closely resembles the original one. You could experiment with the other noise models and filtering operators that EIKONA supports. For example, using the corrupted image of Figure 2.2.9 as the source image, what would you expect for the filtered image to look like, when you utilize a minimum or maximum filter (**Black and white, Filtering** and **Mini** or **Maxi**, respectively)?

Figure 2.2.10 Corrupted and filtered images of LENNA.

Simulation of photographic film noise is also possible by using the noise generators provided by EIKONA. The effect shown in Figure 2.2.11 can be created by using the option **Black and white, Processing, Mult Uni** which is a noise generator that creates multiplicative noise of uniform distribution. We use 0, <New Buffer> as source and destination buffers and we use noise range 0.3.

Figure 2.2.11 Simulation of photographic film noise.

In many cases we are interested in improving the visual quality of an image. For example, the contrast of an image can be improved, by performing histogram equalization [PIT90]. In Figure 2.2.12 the histogram for image LENNA, is shown. It can be computed by **Black and white, Analysis, Histogram, Histogram**. If

the image of LENNA is in image buffer 0, then choose BW buffer 0 in the window that appears.

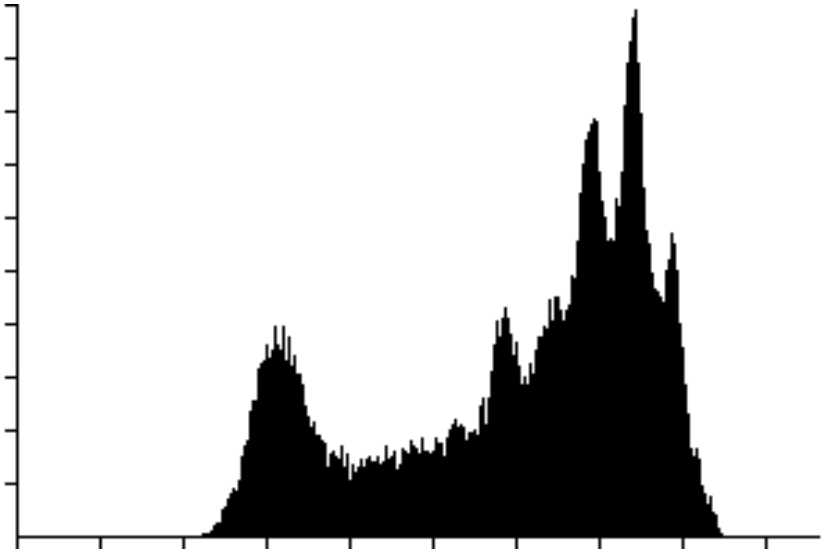

Figure 2.2.12 Histogram of LENNA.

Now, we will perform histogram equalization on LENNA, storing the resulting image in a new buffer. This can be done by the sequence **Black and white, Filtering, Histeq**. Enter BW buffer 0, <New Buffer> as the source and destination images, respectively. The resulting histogram equalized image is shown in Figure 2.2.13, while its histogram is given in Figure 2.2.14.

Figure 2.2.13 Original and histogram-equalized images of LENNA.

We can zoom an image using a variety of interpolation methods through the option **Black and White, Processing, Zoom**. Figure 2.2.15 shows LENNA

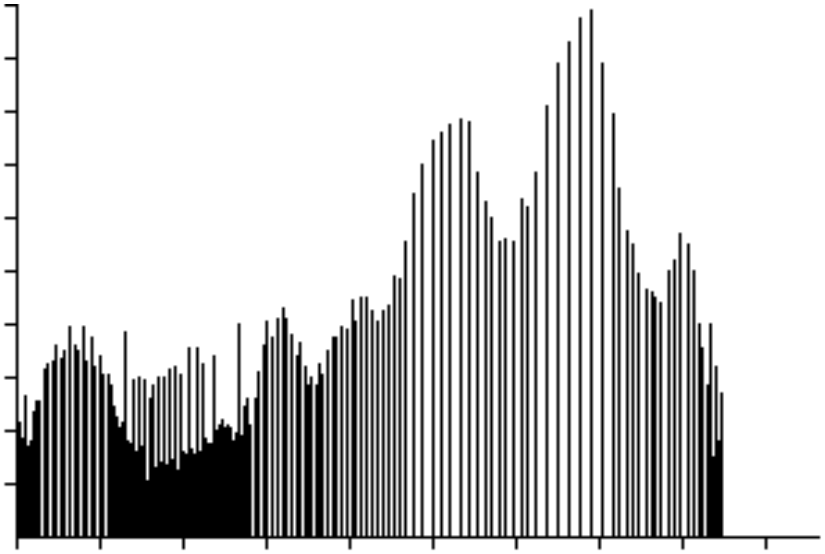

Figure 2.2.14 Histogram of histogram-equalized LENNA.

zoomed by a factor of 2 using zero-order interpolation.

Figure 2.2.15 LENNA and its zoom version.

EIKONA can also be used for image sharpening. Assuming that image BABOON is loaded in BW buffer 0 we can sharpen the left half of it as follows: First we make a copy of the image in a new buffer using **Black and White, Basic, Copy**. Using the mouse we define a ROI that covers the left half of the original image. Then, we select **Black and White, Processing, Sharp** and enter the original image and it's copy as the source and destination buffers respectively. The result of image sharpening on the half part of the original image is impressive, as can be seen from Figure 2.2.16.

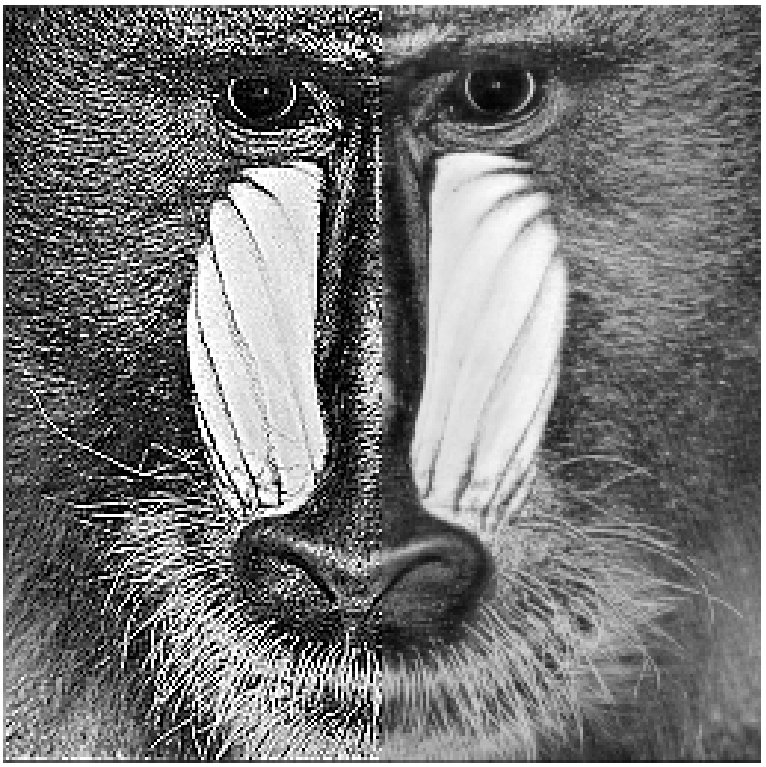

Figure 2.2.16 BABOON with its left side sharpened.

2.2.4 Nonlinear digital image filtering

EIKONA supports a multitude of nonlinear filtering operations. You can perform many median-based filtering by selecting **Black and white, Non-linear filtering, Median Filters**. We can apply a separable median filter to the image of LENNA corrupted by impulsive noise by selecting **Black and white, Non-linear filtering, Median Filters, Separable Median**. Enter the corrupted image as the source image and <New Buffer> as the destination buffer. In the window size fields enter 3, 3. The resulting image is depicted in Figure 2.2.17.

Another non-linear filter is the local adaptive filter, which is based in local statistics. Let us suppose that we have an image of LENNA corrupted by additive uniform noise (this can be done by selecting **Black and white, Processing, Add Uni** and entering the original image as a source, <New Buffer> for destination and 30 for the noise range). Now, the local adaptive filter can be utilized to filter out the noise. This can be done by selecting **Black and White, Non-linear filtering, Local adapt. filter**. Select the corrupted image and <New Buffer> as the source and destination respectively. Since the noise range used for the creation of the noisy image was 30, enter 75 for the noise variance. The corrupted and filtered images are shown in Figure 2.2.18.

Figure 2.2.17 LENNA corrupted by impulsive noise (left) and filtered by a separable median filter (right).

Figure 2.2.18 LENNA corrupted by additive uniform noise and filtered by a local adaptive filter.

Many morphological operations are also supported. These include both binary and grayscale morphological operators. Assuming that the thresholded binary image of LENNA, shown in Figure 2.2.4, is loaded in buffer 1, we can perform a binary opening operation, by performing a binary erosion followed by a binary dilation operation. The first step can be performed by selecting **Black and white, Non-linear filtering, Morphology, Binary Erode** and entering the thresholded image as the source and <New Buffer> for destination. You can use a cross type structuring element, so enter 2 in the structuring element box. The opened version of the image of Figure 2.2.4 can be obtained by performing a binary dilation operation on the eroded image. Select **Black and white, Non-linear filtering, Morphology, Binary Dilate** and enter the eroded image as the source and <New Buffer> as the destination. Both the eroded and the opened images are shown in Figure 2.2.19.

Another well known morphological operation is the skeletonization [PIT93]. It is used to extract the skeleton of an image. EIKONA supports this operation in binary images. Let us suppose that the binary image RECT.BIN is loaded in BW buffer 0. Select **Black and white, Non-linear filtering, Morphology, Skeleton**. Enter BW buffer and <New Buffer> as source and destination image and 2 as the parameter. The original image and its skeleton are shown in Figure 2.2.20.

Figure 2.2.19 Eroded and opened images from thresholded LENNA.

Small details of a binary image can be extracted by the means of the top hat transformation [PIT93]. Once more, we will use the thresholded image of LENNA of Figure 2.2.4, as a demonstration of the transformation. Select **Black and white, Non-linear filtering, Morphology, Top Hat** enter the thresholded image and <New Buffer> as source and destination respectively and 3, 3 for the window's dimensions The result is depicted in Figure 2.2.21. It is obvious, that this morphological transformation yields many small details of the source binary image.

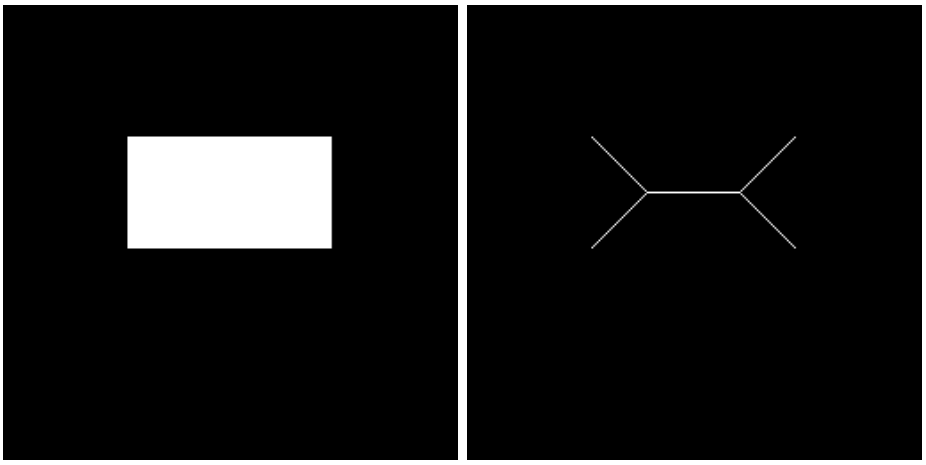

Figure 2.2.20 The binary image RECT and its skeleton.

2.2.5 Edge detection

There are many ways to perform edge detection in an image, using EIKONA. We will show most of these operations, since edge detection is an area of image analysis for which a lot of interest exists.

The compass operation is a directional edge detector. It can detect edges having a specific direction (slope). Although this seems limiting at first, we must take into account that this detector gives us an efficient tool to find edges, we are looking for, with specific slopes. As an example, if LENNA is loaded in BW buffer 0, we can find lines that are almost vertical, by selecting **Black and white, Analysis, Edge detection, Compass** and entering BW buffer 0, <New Buffer> , 90 as the values of the source and destination images and edge direction, respectively. The resulting image is given in Figure 2.2.22.

We can perform edge detection on the left half of LENNA as follows: We copy LENNA in a new buffer through the option **Black and White, Basic, Copy** and define ROI on the original image to cover the left half of the image. We, then, select **Black and White, Analysis, Edge Detection, Sobel** and enter the original LENNA as source and the copy as destination. The result is shown in Figure 2.2.23.

Figure 2.2.21 Top hat transformation of LENNA.

Figure 2.2.22 Result of the compass edge detector on LENNA.

Figure 2.2.23 Result of the sobel edge detector on LENNA.

Remaining in the field of Edge Detection, we should give an example of Line Detection. Select option **Black and White, Analysis, Edge Detection, Line Detect**. Assuming that image BW buffer 0 contains the image LENNA256.DAT, choose BW buffer 0 as source and <New Buffer> for the output buffer. You should also select a line direction from the allowable range, which is:0, 45, 90 and 135 degrees. You can check the results of these actions on your monitor. If you enter 0, 1 and 45 as the source and destination images and line direction, then the image of Figure 2.2.24 will appear on screen.

Figure 2.2.24 Result of the line detect operation on LENNA.

Concluding the Edge Detection menu we give a typical example of how to perform Edge Following. Before selecting the option we should first make visible the image buffer we are interested in, using the option **Black and White, Display**. Then we choose the option **Black and White, Analysis, Edge Detection, Edge Following** and after we insert the proper buffer numbers and threshold values (see [PIT93]), a non-modal window appears instructing us to select the start pixel by left clicking the desirable point on the image buffer. We should note that if some windows overlap we can easily move them in order to make properly visible the image buffer of interest.

2.2.6 Region segmentation and texture analysis

We will now explore some applications of EIKONA in the areas of region segmentation and texture analysis.

If the thresholded image of LENNA, of Figure 2.2.4, is loaded in image buffer 0, we can have EIKONA report the number of distinct binary sets that make up the whole image. This can be easily done by selecting **Black and white, Analysis, Region Segmentation, Count items**. After selecting the thresholded image as the source an information window will pop-up, informing us that this binary image of LENNA consists of 82 sets.

EIKONA can also be used to segment a grayscale image to a number of uniform regions. For example, we can segment the image of LENNA to 4 regions by selecting **Black and White, Analysis, Region Segmentation, Segment** and entering 4 for the number of regions fields. Figure 2.2.25 depicts the segmented image of LENNA, where each gray scale represents one of each regions.

Figure 2.2.25 LENNA and the segmented resulting image.

We already saw an example of the **Threshold** option, so we move to a description of **Region Grow**. Choose **Black and White, Analysis, Region Segmentation, Region Grow**. Select the appropriate input and output image buffers and give the Threshold value which will distinguish one region from another. A non-modal window will appear letting you select the "seeds" (up to 256) before clicking the OK button. Figure 2.2.26 shows the result of applying **Region Grow** on LENNA with a threshold of 20 and the seeds picked at random all over the image.

The next menu to be described concerns Texture Analysis algorithms. The Gray Level Differences Histogram of an image can be computed in the following way. Choose option **Black and White, Analysis, Texture, Gray Level Dif. Histogram**. Assuming that image buffer 0 contains the image LENNA, we give BW Buffer 0 as the source, <New Buffer> for output and enter 1, 1 as the displacement coordinates. Like we saw before, in the Histogram menu, the output is a window containing the visual information of the Gray Level Differences Histogram. It is given in Figure 2.2.27.

Figure 2.2.26 Result of a region grow operation on LENNA.

Some valuable information about this histogram can be calculated through the option **Black and White, Analysis, Texture, Gray Lev. Dif. Hist. Parameters**. Check this information on your monitor in conjunction to the previous example.

One other interesting feature of the Texture Menu is the Angular-Radial option. By choosing **Black and White, Analysis, Texture, Angular-Radial** we can obtain visual information about the angular and radial distribution of the power spectrum of an image in a format similar to the one used to display histograms. For example, the radial and the angular distribution of the power spectrum of LENNA is shown in Figures 2.2.28-2.2.29, respectively.

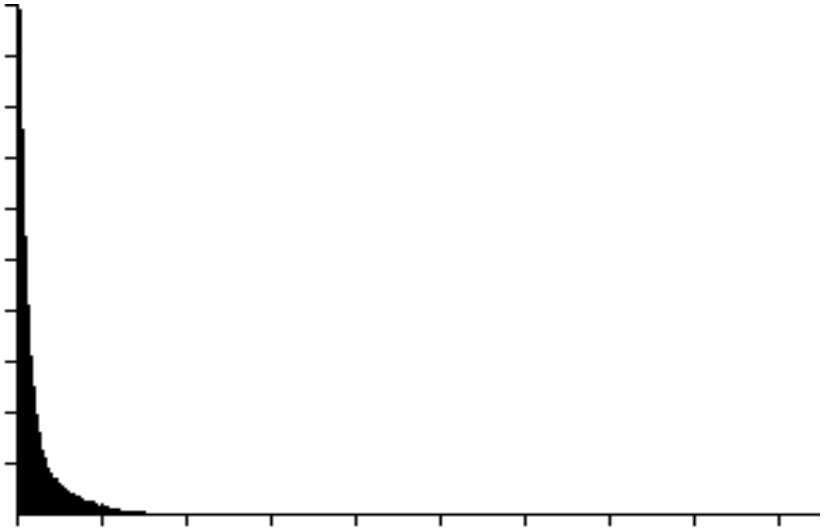

Figure 2.2.27 Gray level dif. histogram of LENNA with displacement values 1,1.

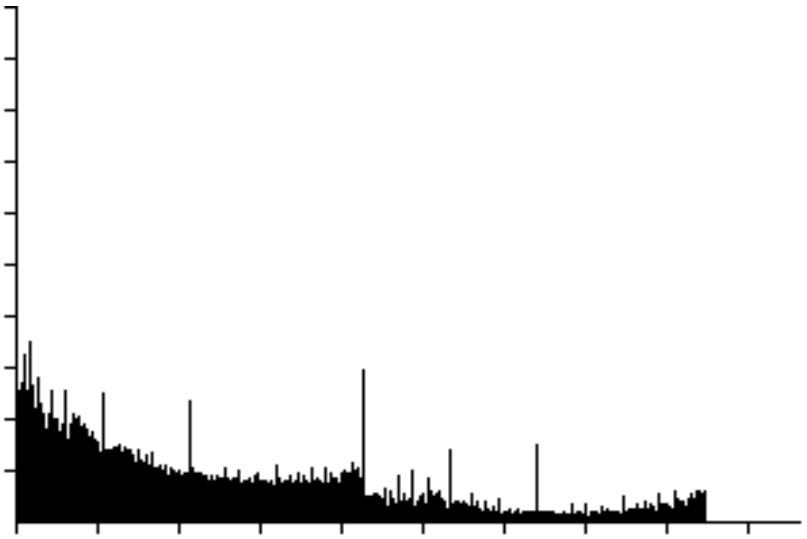

Figure 2.2.28 Radial distribution of PSD of LENNA.

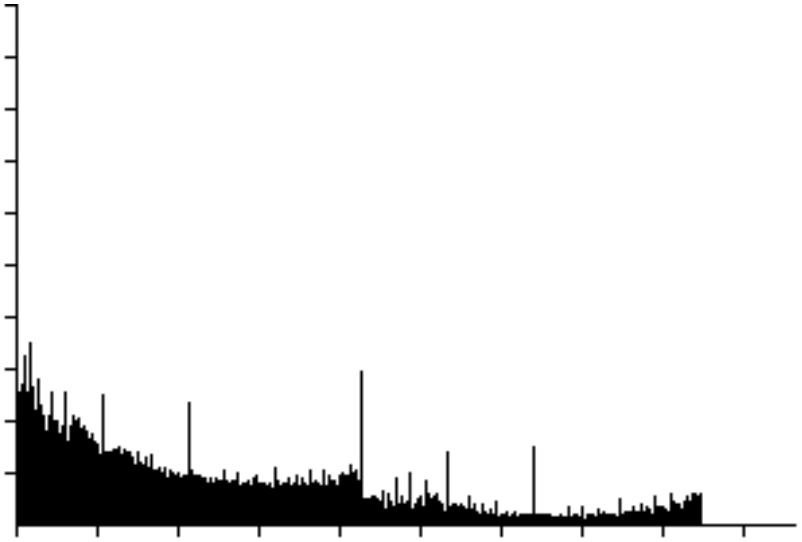

Figure 2.2.29 Angular distribution of PSD of LENNA.

2.2.7 Shape description

The Shape Analysis menu gives valuable information about objects of grayscale and binary images. We should remind you that any grayscale image can be transformed to a binary one by using the **Black and White, Analysis, Region Segmentation, Threshold** option and giving the desired number to act as a threshold between 0 and 1.

We can obtain the characteristics of a binary image by selecting **Black and White, Analysis, Shape Analysis, Find Char**. Let us suppose that the image RECT.BIN is loaded in BW buffer 0. By selecting the operation mentioned above, and entering BW buffer 0 as the source buffer and 0 as the threshold value, EIKONA will respond with a pop-up window, which gives information regarding the the characteristics of the first object to meet in a row wise manner from the upper left corner (i.e. the rectangle). The reader must have understood by now that in order to choose another object, he should adjust the ROI appropriately. For our second example the above mentioned ROI adjustments around an object of interest also hold. Choose the ROI to be the area surrounding the rectangle. If you choose the option **Black and White, Analysis, Shape Analysis, Chain Coding** and select BW buffer for source, <New Buffer> for the output and a moderate number for the maximum number of chain nodes (in this case 2000), you can check the results on your monitor. It is obvious that Chain Coding can be used as another Edge Detection algorithm for binary images. Figure 2.2.30 depicts the result of chain coding.

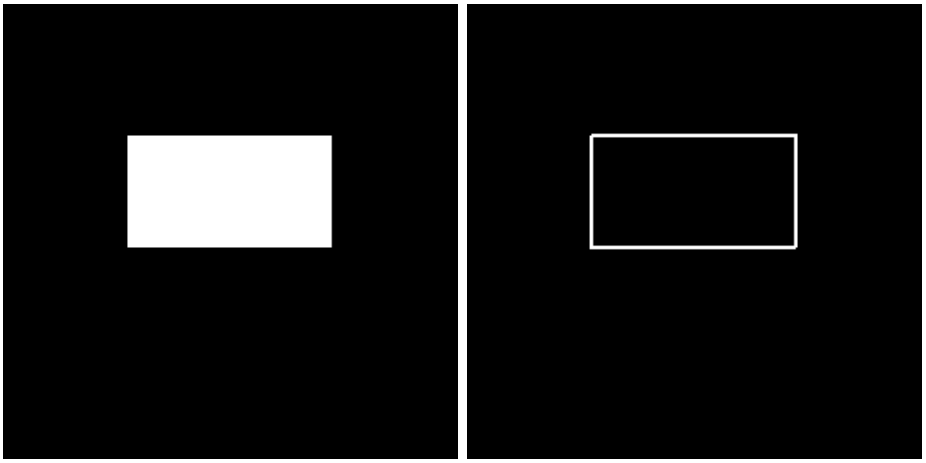

Figure 2.2.30 Original and chain coded image of RECT.

Figure 2.2.31 Thresholded LENNA and the result of one-pass thinning.

Concluding this section we will describe the Pyramid and Thinning options. If we choose **Black and White, Analysis, Thinning** we come up with a submenu containing two options: **One Pass Thinning** and **Two Pass Thinning**. We should note that these are the only options having as output image buffer the input image buffer. As a result, we must be very careful when using these two options, because our original image can be destroyed. If we perform one-pass thining on the thresholded image of LENNA of Figure 2.2.4, then we'll get the binary image of Figure 2.2.31.

By choosing **Black and White, Analysis, Pyramid** and giving the appropriate input and output image buffers, we come up with a very impressive result, as you can verify from Figure 2.2.32 which shows the pyramid of LENNA. It is obvious that adjusting correctly the ROI around images and copying or saving, can give us results similar to those obtained by using the **Black and White, Processing, Decim** option.

Figure 2.2.32 Pyramid of thresholded LENNA.

In this section some examples of grayscale and binary image analysis routines have been given. EIKONA contains several powerful routines for image analysis (e.g. mathematical morphology) that are a very useful tool for the interested and/or advanced user. Information on how to use these routines and commands are found in the library reference manual.

2.3 Examples of color image processing and analysis

In this section we will examine some of the operations on color images, that EIKONA supports. With some notable exceptions, these operations replicate the ones used in the section of grayscale image processing and analysis. For this reason, only a small subset of the operations will be covered here.

2.3.1 Basic image processing

We can easily copy a color image to another by selecting **Color, Basic, Copy**. We must enter the source and the destination image buffers in the box that appears.

We can also add a constant value to every one of the red, green and blue color components that compose the color image. This can be done by selecting **Color, Basic, Addc**. In the box that appears enter the source and destination color image buffer, as well as the constant to be added.

2.3.2 Color transforms

EIKONA implements a large number of color transformations. The CMY transformation essentially produces the "negative" of a color image. As an example, assuming that the image BABOON is loaded in BW buffer 0, select **Color, Color repr., From RGB to, CMY** and enter BW buffer 0, <New Buffer> for source and destination buffers respectively. The resulting image is depicted in Figure 2.3.1. The negative image can be transformed to positive again by choosing the option **Color, Color repr., To RGB from, CMY** and selecting the CMY image as the source buffer.

2.3.3 Digital image filtering and enhancement

Let us suppose that the image file BABOON.TGA is stored on color buffer 0. The image size is 256×256 pixels. We can enlarge image buffer 0 by selecting **Color, Basic, Zoom**, choosing color buffer 0, <New Buffer> as the source and destination buffers respectively and entering the desired zoom factor and interpolation order. Figure 2.3.2 shows BABOON enlarged by a factor of 2 using a zero-order interpolation. Decimation is also available through the **Color, Basic, Decim** option. The user specifies the source and destination buffers and the decimation factor.

Simulation of photographic film noise can be performed by using the noise generators provided by EIKONA. The effect shown in Figure 2.2.12 can be created by using the option **Color, Processing, Mult uni** which is a noise generator that creates multiplicative noise of uniform distribution. Figure 2.3.3 depicts the result

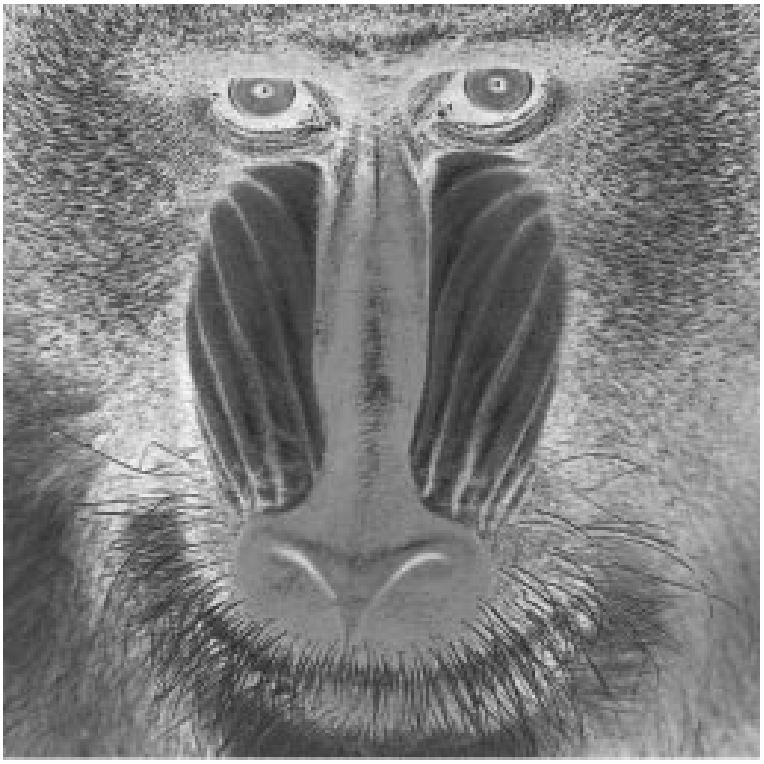

Figure 2.3.1 CMY color model image of BABOON.

of this operation on image BABOON for a noise range of 1.

Histogram equalization can be performed as follows. Let us suppose that the image BABOON.TGA is stored on color buffer 0. The histogram of its R,G,B components can be calculated by choosing option **Black and white, Analysis, Histogram** and by selecting *Color buffer 0 - Channel 1* , *Color buffer 0 - Channel 2* and *Color buffer 0 - Channel 3* respectively. The image histogram is used implicitly to perform image enhancement by histogram equalization. Select **Color, Processing, Histeq**. Enter the buffer holding the BABOON image as source and <New Buffer> for destination. The resulting histogram-equalized image is given in Figure 2.3.4.

Figure 2.3.2 Zoomed image of BABOON.

2.3.4 Edge detection

You can perform edge detection on the left half of image BABOON as follows. Copy the image to a new buffer through the **Color, Basic, Copy**. Define a ROI on the original image that covers the left half of the image. Select **Color, Analysis, Sobel** and enter the original image as the source and the copy as the destination. The result is shown in Figure 2.3.5.

2.3.5 Region segmentation and texture analysis

Region segmentation can be performed by using the option **Color, Analysis, Segment** and specifying the source and destination buffers and the number of regions in which we want each channel of the image the image to be segmented. Figure 2.3.6 shows image BABOON segmented into 4 regions.

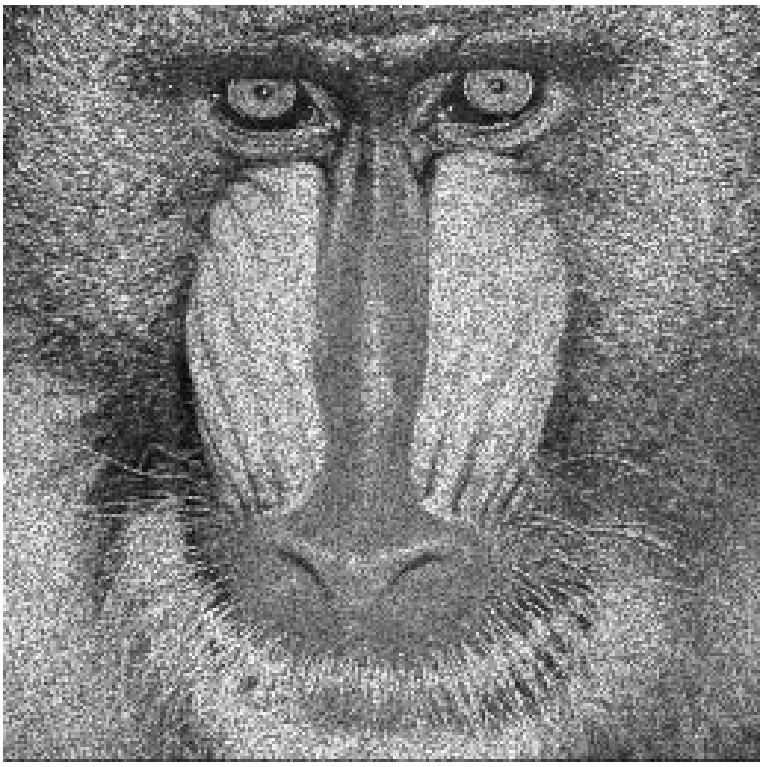

Figure 2.3.3 Simulation of photographic film noise on color image BABOON.

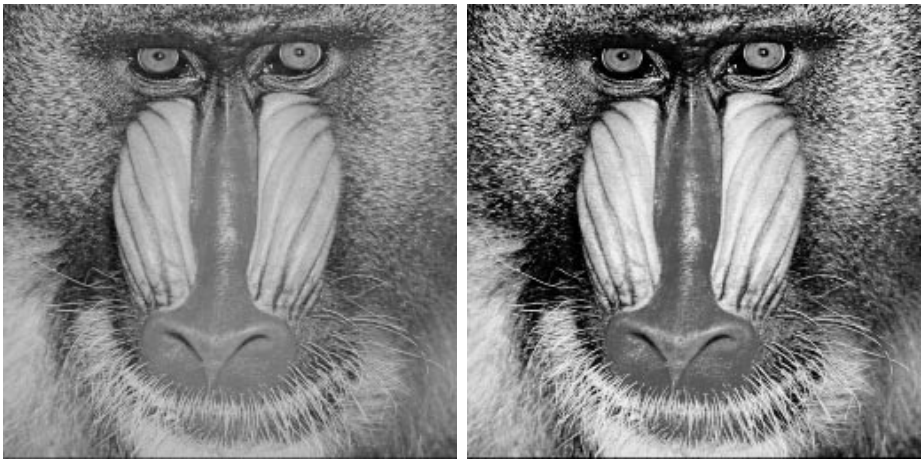

Figure 2.3.4 Original and histogram-equalized images of BABOON.

Figure 2.3.5 Sobel edge detection on image BABOON.

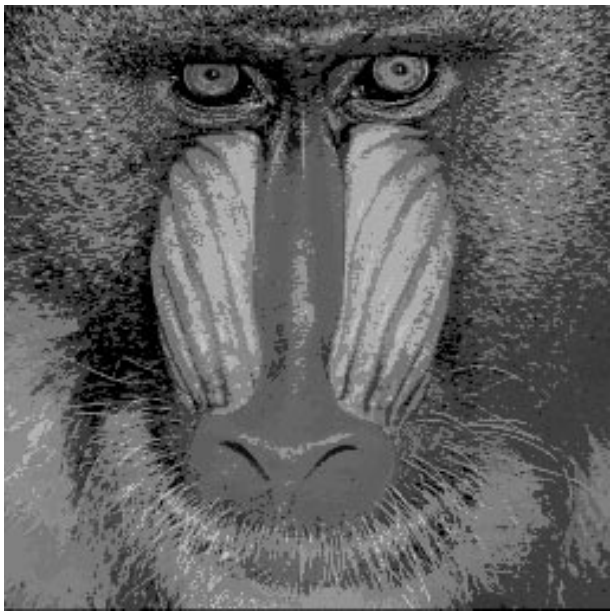

Figure 2.3.6 Segmentation of image BABOON.

3

Writing DLLs for EIKONA

3.1 Introduction

3.1.1 DLL basics

In the Microsoft Windows operating system dynamic-link libraries (DLL) are modules that may contain code or data. The functions making up a DLL's code can be of two kinds :

internal functions which can only be called by other functions defined in the same DLL.

exported functions which can be called by other modules .

A DLL can be loaded at run-time by other programs which can then call the DLL's exported functions. Thus, DLLs provide a way to modularize applications so that basic application functionality can be easily updated or extended (through the DLL's exported code).

3.1.2 EIKONA DLLs

EIKONA makes full use of the benefits provided by DLLs. More specifically, EIKONA can dynamically load any DLL that conforms to a well-defined *interface*. This interface defines a set of rules that a DLL should follow in order to be loaded and used by EIKONA. The rules specify mainly a set of functions that a DLL should export for EIKONA and a number of functions and services that EIKONA provides to attaching DLLs. Among others, the DLL can

- Create a private menu under EIKONA's **Modules** menu and install a function to handle commands from this menu.

- Bypass default EIKONA message processing (if this is required) and provide it's own message handler ...
- ...or just enjoy the readily available EIKONA GUI while concentrating on image processing issues.

3.1.3 The Example DLL

The following sections describe in detail the steps required to build a DLL for EIKONA. The technical details are exemplified by use of the Example DLL. This is a fully functional DLL that can be build and used by EIKONA. It adds one menu under the **Modules** menu that provides two options : **Grayscale Open** for performing an opening operation on a grayscale image and **Region Grow** for segmentation of an image given some seed points on it. Both these functions are already provided by the main EIKONA executable but they serve well for demonstration of the EIKONA–DLL interface. The **Grayscale Open** is a typical operation requiring a simple dialog box for input of a source and a destination buffer. The **Region Grow** option is more complicated as it needs window message handling to allow input of the seeds using the mouse. Code fragments from the Example DLL will be given in the relevant sections. The complete listing is found in the last section of this chapter.

3.2 DLLs for EIKONA

In this section the DLL's part of the EIKONA–DLL interface will be described. In particular each of the five functions that every DLL should export will be examined in detail. Types, structures and functions for this functionality are defined in the header file `DLLDEFS.H` which should be included for every DLL.

3.2.1 Initialization

A DLL for EIKONA (in fact, every DLL) has the chance to perform initialization actions through the DllMain[1] function. This function is the first one called by EIKONA (more precisely by Windows itself) during the DLL loading process.

[1]It should be noted that DllMain is not required in a strict sense. In fact, EIKONA will load and use correctly a DLL without a DllMain function.

Furthermore, the name 'DllMain' is compiler dependent. The runtime library of Visual C++ v4.2 uses this name ; if you use other tools check your documentation for the DLL main entry point.

DllMain

A DLL defined callback function that will be called when the DLL is loaded by EIKONA and when EIKONA closes.

```
BOOL WINAPI
    DllMain (
        HANDLE hInstance,
        ULONG reason_for_call,
        LPVOID lpReserved
        )
```

Parameters

hInstance

holds the DLL's instance. This is particularly useful for DLLs that have resources (dialog boxes, bitmaps, icons) since it is required by the relevant WIN32 functions to access them.

reason_for_call

For Eikona DLLs this can take the following values:

- DLL_PROCESS_ATTACH
  Meaning that the DLL is being loaded. This is the right time for initialization.
- DLL_PROCESS_DETACH
  Meaning that EIKONA is preparing to close. If there is any clean-up to be done for this DLL it should be done now.

lpReserved

This is reserved for system use and should not be used by DLL code.

Return Value

When *reason_for_call* is DLL_PROCESS_ATTACH the DllMain function should return TRUE if the DLL should actually be loaded by EIKONA and FALSE otherwise. For other values of *reason_for_call* DllMain should return TRUE.

The following code fragment shows the Sample DLL's DllMain function and the relevant declarations.

```
HINSTANCE ExampleInstance;
BOOL WINAPI
    DllMain ( HANDLE hInstance, ULONG reason, LPVOID lpReserved )
{
    HMODULE EikonaModule;
    switch ( reason )
    {
    case DLL_PROCESS_ATTACH :
        // Save the DLL's instance in a global variable. We'll need it
```

```
        // later to access our resources.
            ExampleInstance = hInstance;
        //
        // Perform any required initialization here.
        // Return TRUE if everything was OK.
        //
        return TRUE;
    default :
        return TRUE;
    }
}
```

3.2.2 Installing the DLL's menu

The services provided by a DLL are accessed through the **Modules** menu of EIKONA. Clearly, the DLL should create an appropriate entry under the **Modules** menu and respond to choices from this menu accordingly. This is done in close cooperation with EIKONA through the EikonaDLLReturnMenu and EikonaDLLReturnCmdHandler function which every DLL should provide.

EikonaDLLReturnMenu

```
_declspec(dllexport)
    char **EikonaDLLReturnMenu ( void )
```

Return Value

The function should return a pointer to an array of strings containing the names of the commands that should be inserted in the DLL's menu. The format of this array for a menu of N items is shown in the following table :

Element	Points to
1	*name1*
2	*name2*
⋮	
N	*name*N
$N+1$	NULL
$N+2$	*MenuName*

name1, *name2*, ..., *nameN* should be NULL-terminated strings containing command names for the DLL's menu. The *MenuName* should contain the name of the pop-up menu that will be installed by EIKONA for this DLL under the **Modules** menu. The following special characters are recognized in these strings:

- '&' causes the next character in the string to appear underlined in the menu. This character will be the item's keyboard shortcut.
- '-' This is recognized only if it is the first character in a command name. It causes EIKONA to insert a horizontal separator just before this item in the pop-up menu.

Having installed it's menu the DLL should next inform EIKONA about the function that will handle these menu commands. This is done with the EikonaDLL-ReturnCmdHandler function.

EikonaDLLReturnCmdHandler

DLL defined callback function to return the address of the DLL's menu command handler.

```
_declspec(dllexport)
    CMDHANDLERPROC EikonaDLLReturnCmdHandler ( void )
```

Return Value

The return value is of type CMDHANDLERPROC which is a pointer to a function that returns nothing and takes one argument of type WPARAM. This is the function that will be called by EIKONA whenever the user chooses an item from the DLL's menu.

CommandHandler

DLL-defined callback function to handle menu commands for the DLL.

```
void
    CommandHandler (
        WPARAM command
        )
```

Parameters

command

Specifies the command for which handling is required.

Remarks

The name CommandHandler is only a place holder. The function that will be called by EIKONA when a menu item from the DLL's menu is chosen is the one returned by EikonaDLLReturnCmdHandler. For a menu of N items *command* may take values in the range $0 \ldots N$-1.

The following excerpt from the Example DLL shows the menu handling code and declarations.

```
// Menu table for the EXAMPLE DLL.
char *commands[] = { "Grayscale &Open","-&RegionGrow",NULL,"E&xample"};

// Define some constants for menu commands. Note the correspondence with 'commands' above.
enum command_ids { COMMAND_OPEN=0, COMMAND_REGION_GROW };

// Return the table of menu command names. This is the 'commands' array defined above
_declspec(dllexport)
    char **EikonaDLLReturnMenu( void )
{
    return commands;
}

// This is the menu command handler function.
void
    ExampleMenuHandler(WPARAM command)
{
  //
  // Variable declarations.
  //
    switch(command)
    {
        case COMMAND_OPEN :
            //
            // Code to handle the 'Open' command.
            //
        case COMMAND_REGION_GROW :
            //
            // Code to handle the 'Region Grow' command.
            //
    }
}

// Return the command handler function. This is the 'ExampleMenuHandler'
// function defined above.
_declspec(dllexport)
    CMDHANDLERPROC EikonaDLLReturnCmdHandler(void)
{
    return ExampleMenuHandler;
}
```

3.2.3 Installing a message handler

EIKONA allows an attaching DLL to respond to standard window messages thus taking control of windows displayed by it. The DLL may install a special procedure to which window messages will be forwarded. The DLL may then process the message or pass it back to EIKONA if it decides that it should not handle it. In it's full generality this model allows the DLL to take complete control of windows. This, however, requires considerable effort, knowledge of the WIN32 API and EIKONA's internal workings (it should be noted here, that even this part of the chapter is admittedly more demanding). Installation of the message handler is done with the EikonaDLLReturnMsgHandler function described next.

EikonaDLLReturnMsgHandler

DLL defined callback function to install a message handler procedure.

```
_declspec(dllexport)
    MSGHANDLERPROC EikonaDLLReturnMsgHandler (
        DLLMSGRETURNPROC message_return_proc
        )
```

Parameters

message_return_proc

The address of the function the DLL should use to pass message return values to EIKONA.

Return Value

The function should return the address of the message handler function for this DLL or NULL if the DLL will not handle window messages. *message_return_proc* should be saved in a global variable since it will be needed in the message handler.The message handler should be a function having the same argument list as a standard window procedure but of void return type.

The function passed by EIKONA as the argument to the EikonaDLLReturnMsgHandler function should be used by the DLL's handler to notify EIKONA about the result of the message processing. It is described next.

message_return_proc

Function to return message processing return values.

```
void
    message_return_proc (
        BOOL handled,
        LRESULT retvalue
        )
```

Parameters

handled

Should be TRUE if the message was handled by the DLL or FALSE if not.

retvalue

If *handled* is TRUE this is the value EIKONA will return to Windows as the message's return value. Otherwise, it is ignored.

Remarks

The message handler should use this function for *every* message it receives whether handled or not.

The following code fragment shows the Example DLL's relevant code.

```
// Define a pointer for the message return procedure;
DLLMSGRETURNPROC DllMsgReturn;

// This is the message handler function.
void
    ExampleMsgHandler(HWND hwnd, UINT message, WPARAM wparam, LPARAM lparam)
{

    //
    // Various declarations and code to decide whether
    // we should consider this message or not.
    //
    switch ( message )
    {
        case WM_MOUSEMOVE :
            //
            // Code to handle this message.
            //          // Inform EIKONA about the result.
            (*DllMsgReturn)(TRUE,0);
            break;
        case WM_RBUTTONDOWN :
            //
            // Code to handle this message.
            //          // Inform EIKONA about the result.
            (*DllMsgReturn)(TRUE,0);
            break;
        default :
            (*DllMsgReturn)(FALSE, 0);
    }
    return;
}

// Function to return the address of the message handler procedure.
_declspec( dllexport)
    MSGHANDLERPROC EikonaDLLReturnMessageHandler( DLLMSGRETURNPROC f )
{
    //Save the address passed to us by EIKONA and return the address of our
```

```
    //message handler.
    DllMsgReturn = f;
    return ExampleMsgHandler;
}
```

3.3 Functions exported by EIKONA

In this section the functions exported by EIKONA for use by DLLs are described in detail.

3.3.1 Using the exported functions

The functions exported by EIKONA can not be used as regular functions by the DLL. Rather, the DLL accesses them indirectly through a pointer. The pointer is returned by the WIN32 GetProcAddress function. The procedure required to use a function exported by EIKONA is outlined here:

- Define a pointer of the correct type for the function you want to use. Function pointer types are defined in the relevant header files. Suppose, for example that you want to use the library's sobel function (defined in `EIKONA.H`) and the Get1BufferDialog function (defined in the `BUFFERS.h` file). You should define two pointers as shown below

  ```
  SOBEL_PROC sobel;
  GET1BUFFERDIALOG_PROC Get1BufferDialog;
  ```

  Type names are constructed from the name of the function as it appears in the documentation by converting all characters to uppercase and adding the _PROC suffix.
- Get the function's address in the pointer. This is done using the WIN32 GetProcAddress function :

  fpointer = (*typename*)GetProcAddress(*EikonaInstance*,*function_ordinal*);

  fpointer is your function pointer. *typename* should be the appropriate type as explained above (the cast is nessecary to avoid compiler warnings). *EikonaInstance* is the instance handle of the EIKONA application (which can be easily retrieved with the call GetModuleHandle("eikona.exe")). *function_ordinal* is the function's export ordinal. Constants with suggestive names for function

ordinals are defined in the file `EXPFUNC.H` which should be included by every DLL. They can be constructed from the function's name as it appears in the documentation by converting all characters to uppercase. Continuing our example and assuming that the variable *EikInst* holds EIKONA's instance handle, you should initialize the two pointers as shown below:

```
sobel = (SOBEL_PROC)GetProcAddress(EikInst,SOBEL));
Get1BufferDialog =(GET1BUFFERDIALOG_PROC)
                        GetProcAddress(EikInst, GET1BUFFERDIALOG);
```

- To call the function just use the normal C syntax for function calls through pointers like this

  ret = (**fpointer*)(*argument list*);

3.3.2 EIKONA library functions

EIKONA, naturally enough, exports all the image processing routines provided by the EIKONA Image Processing library. They are accessed using the mechanism described above. The required types are defined in the `EIKONA.H` file. To allow the programmer to get standard library error messages EIKONA also exports the following function.

error_handler

Display a message box with a message appropriate for the specified library error code.

```
void
    error_handler (
        int errcode
        )
```

Parameters

errcode

The error code for which to print a message. This should be a value returned by a library function.

Remarks

The function also sets the mouse cursor to the standard arrow shape so you don't

have to reset it explicitly every time an error occurs.

3.3.3 Buffer management functions

EIKONA provides a variety of functions for buffer management. Included are functions for creating and deleting buffers, counting the number of buffers and getting user's buffers selection through dialog boxes. There are also some macros for accessing common fields of internal buffer structures (the user is encouraged to use these macros rather than directly accessing the structures).The required types and constants are defined in the `BUFFERS.H` file which should be included by every DLL that uses this functionality. We begin by describing the accessor macros

BUFFER_TYPE

Get's the specified buffer's type.

```
unsigned
    BUFFER_TYPE (
        BUFFER buffer
        )
```

Parameters

buffer
: A buffer.

Return Value

Returns *buffer*'s type.

BUFFER_WIDTH , BUFFER_HEIGHT

Get a buffer's width and height respectively.

```
int
    BUFFER_WIDTH (
        BUFFER buffer
        )

int
    BUFFER_HEIGHT (
        BUFFER buffer
        )
```

Parameters

buffer

A buffer.

Return Value

Return *buffer*'s width and height respectively.

UC_BUFFER_CHANNEL

Get an image buffer's channel.

```
image
    UC_BUFFER_CHANNEL (
        BUFFER buffer,
        int n
        )
```

Parameters

buffer

A buffer who's type is either GRAYSCALE_IMAGE or COLOR_IMAGE.

n

The number of channel required. It should be 1, 2 or 3.

Return Value

Returns an image which is *buffer*'s *n*'th channel.

Remarks

If *buffers*'s type is COLOR_IMAGE channels 1, 2 and 3 are its R,G and B components respectively. If *buffer*'s type is GRAYSCALE_IMAGE then channel 0 holds it's grayscale level and the other channels are guaranteed to be NULL. For other buffer types the result is undefined.

FLOAT_BUFFER_CHANNEL

Get a float buffer's channel.

```
matrix
    FLOAT_BUFFER_CHANNEL (
        BUFFER buffer,
        int n
        )
```

Parameters

buffer

Specifies the buffer whose channel is required. It' type should be either COLOR_PROCESSING_MATRIX or SINGLE_CHANNEL_MATRIX.

n

The channel required. It should be 1, 2, or 3.

Return Value

Returns an image which is *buffer*'s *n*'th channel.

Remarks

For SINGLE_CHANNEL_MATRIX buffers the float matrix is stored in channel 0. For image buffers the result is undefined.

CreateBuffer

Creates a buffer of the specified type and dimensions.

```
BUFFER
    CreateBuffer (
        int type,
        int width,
        int height
        )
```

Parameters

type

Can be any of the following

- GRAYSCALE_IMAGE
  Create a buffer for a grayscale image.
- COLOR_IMAGE
  Create a buffer suitable for a color image (having three grayscale channels)
- SINGLE_CHANNEL_MATRIX
  Create a float matrix.
- COLOR_PROCESSING_MATRIX
  Create a float buffer suitable for color space transformations (having three float matrices).

width

Width of the new buffer.

height

Height of the new buffer.

Return Value

Returns a BUFFER if succesfull or NULL if an error occurred.

DeleteBuffer

Deletes the specified buffer.

```
void
    DeleteBuffer (
        BUFFER buffer
        )
```

Parameters

buffer
: The buffer to be deleted.

CreateBufferEx

Creates a buffer allowing more control over the creation parameters.

```
BUFFER
    CreateBufferEx (
        int type,
        int width,
        int height,
        unsigned flags
        )
```

Parameters

type
: See CreateBuffer above.

width
: Width of the new buffer.

height
: Height of the new buffer.

flags
: Can be any combination of

- CBE_BUFFER_IN_LIST
  The new buffer will be linked to the list of buffers maintained by EIKONA (the most important consequence of this being that the buffer will be visible to the user through the buffer selection dialog boxes). If this flag is not specified the new buffer will not appear in EIKONA dialog boxes.
- CBE_ALLOCATE_CHANNELS
  Space will not be allocated for the channels required for this buffer.

Return Value

Returns the newly allocated buffer or NULL if an error occurred.

Remarks

Buffers allocated without the CBE_ALLOCATE_CHANNELS flag do not have their channel fields initialized. The programmer should fill in these fields before using the buffer (using the xx_BUFFER_CHANNEL macros).

DeleteBufferEx

Low-level buffer deletion routine. See **Remarks** below.

```
void
    DeleteBufferEx (
        BUFFER buffer
        unsigned flags
        )
```

Parameters

buffer

The buffer to be deleted.

flags

Can be any combination of

- DBE_FREE_IN_LIST
  Marks *buffer* as deleted in the buffer list maintained by EIKONA but *does not* actually delete it. Useful if you want to hide this buffer from user-interface elements.
- DBE_FREE_CHANNELS
  Frees the space used by *buffer*'s channels.
- DBE_FREE_BUFFER_STRUCT
  Free the space occupied by internal housekeeping structures. Buffers allocated by CreateBufferEx without the CBE_BUFFER_IN_LIST flag should be deleted with this flag specified.

Remarks

This is a rather low-level function. The user is strongly encouraged not to use this function except only for deleting a buffer created by CreateBufferEx without the CBE_BUFFER_IN_LIST flag.

EnoughBuffers

Checks if there are enough buffers of the specified type.

```
BOOL
    EnoughBuffers (
        unsigned type,
        unsigned nbuffers
        )
```

Parameters

type
: The type of buffers for which availability is to be tested.

nbuffers
: The minimum number of buffers of type *type* required.

Return Value

Returns TRUE if there are at least *nbuffers* of type *type* or FALSE if there are not. In the later case it displays an appropriate message box.

GetNumberOfBuffers

Get the number of buffers of the specified type.

```
unsigned
    GetNumberOfBuffers (
        unsigned type
        )
```

Parameters

type
: Specifies the type of buffers to count.

Return Value

Returns the number of buffers of type *type* that where found in EIKONA's global list of buffers.

PrepareLibrary

The PrepareLibrary function prepares the EIKONA image processing library for use of a specified buffer by correctly setting up the library's internal state.

```
void
    PrepareLibrary (
        BUFFER buffer
        )
```

Parameters

buffer
: A buffer.

Remarks

The function sets MMAX and NMAX (or FMMAX and FNMAX depending on the *buffer*'s type) to the *buffer*'s width and height respectively. It should be called before any library function is used.

The following two functions are designed to ease the process of getting user input while retaining the familiar EIKONA user interface (combo-boxes for selecting buffers). They are rather low-level functions since they should be used in a dialog procedure written by the DLL programmer.

FillInDialogControl

Fill a dialog control (combo- or listbox) with available buffers of the specified type.

```
void
   FillInDialogControl (
      HWND dialog,
      int control_id,
      int control_type,
      int buffer_type,
      int width,
      int height,
      unsigned flags
      )
```

Parameters

dialog

Handle of the dialog box that contains the control to be filled.

control_id

The resource identifier of the control to be filled.

control_type

Specifies the type of the control. Can be one of the following :

- COMBOBOX
  Specifies that the control is a combobox. It should be created with the CBS_DROPDOWNLIST style.
- LISTBOX
  Specifies that the control is a listbox.

buffer_type

Specifies the type of buffers that will be used to fill the control.

width,

height
Specify the dimensions of the buffers that will be used to fill the control depending on the value of *flags*.

flags
Can be a combination of the following :

- MATCH_SIZE_EXACTLY
  Only buffers of size exactly *width*×*height* will be entered in the control.
- AT_LEAST_THAT_SIZE
  Buffers whose width and height are larger than *width* and *height* respectively will be entered in the control.
- SUGGEST_ACTIVE_BUFFER
  If this flag is specified the the buffer displayed by the currently active window (if any) will be initially selected in the control.
- NO_NEW
  By default the control will display the <New Buffer> option. If this flag is specified the <New Buffer> option will not appear in the control.

Remarks

The MATCH_SIZE_EXACTLY and AT_LEAST_THAT_SIZE flags cannot be combined. If *buffer_type* is GRAYSCALE_IMAGE a separate entry for each channel of a COLOR_IMAGE buffer will be created in the control.

GetInfoFromDialogControl

Get the currently selected buffer and channel from a dialog control previously filled with FillInDialogControl.

```
void
   GetInfoFromDialogControl (
      HWND dialog,
      int control_id,
      int control_type,
      int buffer_type,
      int width,
      int height,
      unsigned flags,
      BUFFER * pbuffer,
      void ** pchannel
      )
```

Parameters

dialog
Handle of the dialog box that contains the control to read.

control_id
Resource identifier of the control.

control_type
Specifies the type of the control to be read. Can be either COMBOBOX or LISTBOX.

buffer_type
Type of buffers in the control.

width, height
Dimensions of the buffers in the control. See **Remarks** below.

flags
Flags used to fill the control.

pbuffer
Points to a BUFFER that on return will receive the buffer selected in the control.

pchannel
Address of a generic pointer where the specific channel selected will be written.

Remarks

This function should be used in conjunction with the previous one. In particular, it should be called for a control previously filled with FillInDialogControl with the *buffer_type*, *width*, *height* and *flags* arguments having the same values in the two calls. If this is not the case, the results in *pbuffer* and *pchannel* are *not* correct. If *buffer_type*=GRAYSCALE_IMAGE and the current selection specifies a COLOR_IMAGE buffer's channel than the color buffer is returned in *pbuffer* and the particular channel in *pchannel*. If the NO_NEW flag is not specified, and the current selection is the <New Buffer> option then **pbuffer*=NULL on return.

The following functions are provided to allow the programmer to get user input without having to delve deep in Windows programming details. They allow the complete operation of dialog boxes for selecting 1 or 2 buffers with 1 or 2 parameters. The collection is by no means exhaustive and the programmer should be aware that full control is possible only by designing his own dialog boxes and providing custom dialog box procedures. Since they are very similar only two of them (Get1BufferDialog and Get1Buffer1ParDialog) will be described in detail – only the prototypes will be given for the rest.

Get1BufferDialog

Create, display and operate a dialog box for selecting a buffer.

```
BOOL
    Get1BufferDialog (
        char * title,
        unsigned type,
        char * caption,
        unsigned flags,
        BUFFER * pbuffer,
        void ** pchannel
        )
```

Parameters

title

Points to a null-terminated string to be used as the dialog's title.

type

Specifies the type of the buffers that should appear on the buffer selection controls of the dialog.

caption

Points to a null-terminated string to be used as a description for the buffer selection control.

flags

Can be any combination of the following :

- SUGGEST_ACTIVE_BUFFER
  If this flag is specified the buffer displayed by the currently active window will be initially selected in the buffer selection control.
- NO_NEW
  By default the buffer selection control includes the <New Buffer> option. If this flag is specified this option is not included in the control.

pbuffer

Address of a BUFFER where the selected buffer will be written on return.

pchannel

Address of a pointer where the specific channel of the selected buffer will be written on return. See **Remarks** below.

Return Value

Returns TRUE if user clicked the *OK* button or FALSE if the user canceled the operation.

Remarks

If *type* is GRAYSCALE_IMAGE a separate entry is created in the control for each channel of a COLOR_IMAGE buffer and the user can select one of the channels. In this case the routine writes the COLOR_IMAGE buffer's address in *pbuffer* and the channel selected in *pchannel*. If the NO_NEW flag is not specified in *flags* and the

user actually selects the <New Buffer> option then * *pbuffer* = NULL on return.

Get1Buffer1ParDialog

Create, display and operate a dialog box for selecting a buffer and entering a parameter value.

```
BOOL
    Get1Buffer1ParDialog (
        char * title,
        unsigned type,
        char * buffer_caption,
        unsigned flags
        BUFFER * pbuffer,
        void ** pchannel,
        char * parameter_caption,
        char * parameter_value
        )
```

Parameters

title
: Points to a null-terminated string to be used as the dialog's title.

type
: Specifies the type of buffers that will appear in the buffer selection control.

buffer_caption
: Points to a null-terminated string to be used as a description for the buffer selection control.

flags
: See Get1BufferDialog above.

pbuffer
: Address of a BUFFER where the selected buffer will be written on return.

pchannel
: Address of a pointer where the specific channel of the selected buffer will be written on return. See **Remarks** for Get1BufferDialog above.

parameter_caption
: Points to a null-terminated string that will be used as a description for the parameter's edit field.

parameter_value
: Points to an array of characters of size at least 5 that will receive the text the user entered in the parameters edit field.

Return Value

Returns TRUE if user clicked the *OK* button or FALSE if user canceled the operation.

Get1Buffer2ParDialog

Create display and operate a dialog box for selecting a buffer and entering two parameter values.

```
BOOL
    Get1Buffer2ParDialog (
        char * title,
        unsigned buffer_type,
        char * buffer_caption,
        unsigned flags,
        BUFFER * pbuffer,
        void ** pchannel,
        char * parameter1_caption,
        char * parameter1_value
        char * parameter2_caption
        char * parameter2_value
        )
```

Get2Buffers1ParDialog

Create, display and operate a dialog box for selecting two buffers and one parameter value.

```
BOOL
    Get2Buffers1ParDialog (
        char * title,
        unsigned buffer1_type,
        char * buffer1_caption,
        unsigned buffer1_flags,
        BUFFER * pbuffer1,
        void ** pchannel1,
        unsigned buffer2_type,
        char * buffer2_caption,
        unsigned buffer2_flags,
        BUFFER * pbuffer2,
        void ** pchannel2,
        char * parameter_caption,
        char * parameter_value
        )
```

Get2Buffers2ParDialog

Create, display and operate a dialog box for selecting two buffers and two parameter values.

```
BOOL
    Get2Buffers2ParDialog (
        char * title,
        unsigned buffer1_type,
        char * buffer1_caption,
        unsigned buffer1_flags,
        BUFFER * pbuffer1,
        void ** pchannel1,
        unsigned buffer2_type,
        char * buffer2_caption,
        unsigned buffer2_flags,
        BUFFER * pbuffer2,
        void ** pchannel2,
        char * parameter1_caption,
        char * parameter1_value,
        char * parameter2_caption,
        char * parameter2_value
        )
```

3.3.4 Thumbnail functions

Almost every dialog box displayed by EIKONA provides a thumbnail view of the buffer currently selected in the active combo-box. To help the DLL writer preserve this look in his custom DLLs EIKONA exports the relevant functions. The required definitions can be found in the `THMBNAIL.H` file which should be included if you use these features.

CreateDialogThumbnail

Prepare a specified static dialog control to be used for buffer thumbnails.

```
void
    CreateDialogThumbnail (
        HWND dialog,
        int thumbnail_control_id
        )
```

Parameters

dialog

Handle of the dialog box containing the thumbnail.

thumbnail_control_id

Specifies the resource identifier of the control to be used for the thumbnail.

Remarks

The control specified by a static control created with the SS_CENTERIMAGE, SS_BITMAP and SS_REALSIZEIMAGE style bits. This function should be used in the dialog procedure's WM_INITDIALOG handler.

DestroyDialogThumbnail

Destroys the thumbnail created by the most recent call to the CreateDialogThumbnail function.

```
void
    DestroyDialogThumbnail ( void )
```

Remarks

Obviously, calls to CreateDialogThumbnail and DestroyDialogThumbnail should be paired. An attempt to create a new thumbnail while there is already one will not be rejected but will result in a memory leak.

DrawThumbnail

Displays the contents of the specified buffer in a thumbnail control.

```
void
    DrawThumbnail (
        HWND dialog,
        int thumbnail_control_id,
        BUFFER buffer,
        void * channel
        )
```

Parameters

dialog

Handle of the dialog box that contains the thumbnail.

thumbnail_control_id

Resource identifier of the static control on which the thumbnail will be displayed.

buffer
The buffer to view in the thumbnail.

channel
The channel to view.

Remarks

The thumbnail identified by *dialog* and *thumbnail_control_id* should be previously initialized with the CreateDialogThumbnail function. If *buffer* is a COLOR_IMAGE buffer and *channel*=NULL the thumbnail will be drawn in color. If *channel* is one of the COLOR_IMAGE buffer's channels then the thumbnail created will be a grayscale one showing the specified channel. If *buffer*=NULL the text "`New Buffer`" will be printed in the thumbnail.

3.3.5 Magnifier functions

Whenever EIKONA requires mouse input from user (as in the **Region Grow** menu option) it displays a modal dialog box that shows a magnified view of the image region around the cursor. The relevant functions are also exported to allow the programmer to use this user-friendly feature in his DLL. The nessecary definitions can be found in the `MAGNIFIER.H` file.

CreateDialogMagnifier

Prepare a specified static dialog control for showing a magnified view of an image.

```
BOOL
  CreateDialogMagnifier (
    HWND dialog,
    int control_id
    )
```

Parameters

dialog
Specifies the dialog box which contains the control identified by *control_id*.

control_id
The resource identifier of the static control to be used for the magnification.

Return Value

Returns TRUE if succesfull or FALSE if the magnifier could not be created.

DestroyDialogMagnifier

Destroys the magnifier previously created with the CreateDialogMagnifier function freeing up all associated resources.

```
void
    DestroyDialogMagnifier ( void )
```

MagnifyInDialog

Displays a magnified (×3) view of the image under the cursor.

```
void
    MagnifyInDialog (
        HWND window,
        int xpos,
        int ypos,
        BOOL flag
        )
```

Parameters

window
: The window handle of the window that displays the image to magnify.

xpos, ypos
: The coordinates of the center of the square that will be magnified.

flag
: If TRUE a green cross will be drawn in the magnifier ; otherwise a large red × sign will be drawn (denoting an invalid image region).

Remarks

The routine will create a magnified view (by a factor of 3) of a 40×40 pixels square centered around the point specified by *xpos* and *ypos* on the dialog control specified with an earlier call to CreateDialogMagnifier.

3.3.6 Window Management functions.

This section describes functions used for displaying a buffer and getting information about a displayed buffer. Relevant definitions can be found in the `WINMAN.H` file.

RefreshBuffer

Updates the window displaying the specified buffer or creates a new window for it if the buffer is not currently displayed.

```
HWND
    RefreshBuffer (
        BUFFER buffer
        )
```

Parameters

buffer

The buffer to be refreshed. It should be either a COLOR_IMAGE or a GRAYSCALE_IMAGE buffer.

Return Value

Returns the handle of the window displaying *buffer* or NULL if it could not create such a window.

Remarks

This function should be called after every operation with *buffer* being the destination buffer of the operation so that changes it's contents are drawn.

DisplayBufferEx

Display the specified buffer. The user can specify the title for the window that will be created and some features the window should support.

```
HWND
    DisplayBufferEx (
        BUFFER buffer,
        unsigned flags,
        char * title
        )
```

Parameters

buffer

The buffer to display. It should be either a GRAYSCALE_IMAGE or a COLOR_IMAGE buffer.

flags

Specifies a number of capabilities for the window to be created. Can be any combination of :

- HAS_ROI
  ROI operations will be available for this window. In particular, the ROI rectangle will be drawn whenever the window is active and the usual ROI marking procedure will be applicable. If this flag is not specified ROI features will not be available for this window.
- HAS_ZOOM
  The *Zoom In / Zoom Out* operations will be available for this window. If this flag is not specified, the relevant buttons on EIKONA's control window will be disabled for this window.
- HAS_CLOSE_BUTTON
  The *Close* button and window menu option will be enabled for this window. If this flag is not specified these options will be disabled.

title

A null-terminated string that will be the new window's title.

Return Value

Returns the handle to the new window or NULL if the window could not be created.

Remarks

Windows created by this function without the HAS_CLOSE_BUTTON flag are intended for displaying special buffers for user input. You should explicitly close such windows when they are no longer needed since the user will not be able to close them.

GetROI

Get the ROI coordinates

```
void
   GetROI (
      BUFFER buffer,
      int * top,
      int * left,
      int * bottom,
      int * right
      )
```

Parameters

buffer

The buffer who's ROI coordinates are required.

top, left, bottom, right
: Pointers to integers where the upper-left and bottom-right ROI corner coordinates will be written on return.

Remarks

If *buffer* is not currently displayed or *buffer* is a float buffer the ROI is assumed to cover the entire buffer.

Apart from these high level functions, there are some low-level macros for accessing internal structures. EIKONA stores information about every window displayed in a structure allocated by Windows. This structure contains useful information about the window such as the current scrollbar positions, the current zoomfactor for the window and the buffer the window displays. A pointer to this structure can be obtained from the window handle of the window using the call

```
WINFO window =(WINFO) GetWindowLong ( hwnd, 0 )
```

where WINFO is the correct type for such a pointer (defined in the file `WINMAN.H`) and *hwnd* is the window handle. The programmer can then use the following macros (note that they all take an argument of type WINFO).

WINDOW_BUFFER

Get the buffer displayed by the specified window.

```
BUFFER
   WINDOW_HANDLE (
      WINFO window
      )
```

WINDOW_SCROLLPOS_H, WINDOW_SCROLLPOS_V

Return the current scrollbar positions (horizontal and vertical).

```
int
   WINDOW_SCROLLPOS_H (
      WINFO window
      )

int
   WINDOW_SCROLLPOS_V (
      WINFO window
      )
```

WINDOW_ZOOMFACTOR

Return the current zoomfactor for the window.

```
int
    WINDOW_ZOOMFACTOR (
        WINFO window
        )
```

3.4 Example DLL source code

```
#include <windows.h>

#include "dll definitions.h"
#include "buffers.h"
#include "eikona.h"

#include "exported library functions.h"
#include "window management definitions.h"
#include "magnifier.h"
#include "resource.h"


HINSTANCE ExampleInstance;

// TRUE while user marks RegionGrow seeds.
BOOL InRegionGrow;

// Count left button clicks in source buffer's ROI.
int ClickCount;

// Threshold to use in RegionGrow.
int threshold;

// Handle of window displaying RegionGrow's source buffer.
HWND RegionGrowWindow;

// BUFFERs and image channels used for user input.
BUFFER buffer1, buffer2, tmp;
image channel1, channel2;


int roitop, roileft,roibottom, roiright;
// Pointer to a structure holding information about EIKONA's windows.
EIKONAINFO EikonaInfo;
```

```
// Pointers to functions we will use.
DLLMSGRETURN_PROC DllMsgReturn;
GET2BUFFERSDIALOG_PROC Get2BuffersDialog;
GET2BUFFERS1PARDIALOG_PROC Get2Buffers1ParDialog;
CREATEBUFFER_PROC CreateBuffer;
DELETEBUFFER_PROC DeleteBuffer;
PREPARELIBRARY_PROC PrepareLibrary;
GETROI_PROC GetROI;
GETEIKONAINFO_PROC GetEikonaInfo;
CREATEDIALOGMAGNIFIER_PROC CreateDialogMagnifier;
DESTROYDIALOGMAGNIFIER_PROC DestroyDialogMagnifier;
MAGNIFYINDIALOG_PROC MagnifyInDialog;
ENOUGHBUFFERS_PROC EnoughBuffers;
REFRESHBUFFER_PROC RefreshBuffer;
WIN_BUILD_IMAGE_PROC win_build_image;
WIN_FREEUC2_PROC win_freeuc2;
ERROR_HANDLER_PROC error_handler;
IMERODE_PROC imerode;
IMDILATE_PROC imdilate;
REGION_GROW_PROC region_grow;
CLEAR_PROC clear;

// Dialog procedure for the 'PixelInput' dialog used for selecting
    // RegionGrow seeds.
BOOL CALLBACK
    PixelInputProc ( HWND hdlg, UINT message, WPARAM wparam, LPARAM lparam )
{
    int errcode;

    switch (message)
    {
        case WM_COMMAND:
            switch (LOWORD(wparam))
            {
                case IDOK:

            // Turn off the 'InRegionGrow' flag and close the
            // 'Pixel Input' dialog.
                    InRegionGrow = FALSE;
                    (*DestroyDialogMagnifier)();
                    DestroyWindow ( hdlg );

            // Complete the operation.
                    SetCursor ( LoadCursor(NULL,IDC_WAIT));
                    errcode = (*region_grow)( channel1, channel2,
                            0,0,roibottom,roiright,ClickCount,threshold);
                    if (errcode )
                    {
                        if ( ! buffer2 )
                            (*DeleteBuffer)(tmp);
                        (*error_handler)(errcode);
```

```
                }
                else
                    (*RefreshBuffer)(tmp);
                return (TRUE);

            case IDCANCEL:
                lnRegionGrow = FALSE;
                (*DestroyDialogMagnifier)();
                DestroyWindow ( hdlg );
                if ( ! buffer2 )
                    (*DeleteBuffer)(tmp);

        }
        break;
    case WM_INITDIALOG :
        lnRegionGrow = TRUE;
        (*CreateDialogMagnifier)( hdlg, IDW_MAGNIFIER);
        return TRUE;

    }
    return FALSE;
}

// Message handler for the EXAMPLE DLL.
void
    ExampleMsgHandler ( HWND hwnd, UINT message, WPARAM wparam, LPARAM lparam )
{
    WINFO window;
    int xpos, ypos;
    int zoomfactor;

// We'll only handle messages when selecting RegionGrow seeds.
// If this is NOT the case, let EIKONA handle the message.
    if ( ! lnRegionGrow )
    {
        (*DllMsgReturn)(FALSE,0);
        return;
    }

// We only handle messages for the window displaying RegionGrow's source
// buffer. If this message is for another window, let EIKONA handle it.
    if ( hwnd != RegionGrowWindow )
    {
        (*DllMsgReturn)(FALSE,0);
        return;
    }
    window = (WINFO) GetWindowLong ( hwnd, 0);
    zoomfactor = WINDOW_ZOOMFACTOR( window );

    switch ( message )
        {
            case WM_MOUSEMOVE :
```

```
            SetCursor ( LoadCursor(NULL,IDC_CROSS));
            xpos = (WINDOW_SCROLLPOS_H( window )+LOWORD(lparam))*zoomfactor;
            ypos = (WINDOW_SCROLLPOS_V( window )+HIWORD(lparam))*zoomfactor;
            (*MagnifyInDialog)( window, LOWORD(lparam), HIWORD(lparam),
                    xpos<=roiright && xpos>roileft &&
                    ypos<=roibottom && ypos>roitop);

            (*DIIMsgReturn)(TRUE,0);
            break;
        case WM_LBUTTONDOWN :
            xpos = (WINDOW_SCROLLPOS_H( window )+LOWORD(lparam))*zoomfactor;
            ypos = (WINDOW_SCROLLPOS_V( window )+HIWORD(lparam))*zoomfactor;
            if ( xpos <= roiright && xpos > roileft &&
                ypos <= roibottom && ypos > roitop )
                channel2[ypos][xpos] = ClickCount ++;
            (*DIIMsgReturn)(TRUE,0);
            return;


        default:

            (*DIIMsgReturn)(FALSE,0);
            break;
    }
    return;
}


// Menu table for the EXAMPLE DLL.
char *commands[] = { "Grayscale &Open","-&RegionGrow",NULL,"E&xample"};

// Command identifiers for the EXAMPLE DLL.
enum command_ids { COMMAND_OPEN=0,COMMAND_REGION_GROW};

void
    ExampleCommandHandler ( WPARAM command )
{
    int ret;
    image intermediate, str_el;
    int errcode;
    char t[5];

    switch ( command )
    {
        case COMMAND_OPEN :
            if ( ! (*EnoughBuffers)(GRAYSCALE_IMAGE, 1 ) )
                break;

            ret = (*Get2BuffersDialog)( "Example DLL : Grayscale Open", GRAYSCALE_IMAGE,"Source Image",
                    NO_NEW — SUGGEST_ACTIVE_BUFFER, &buffer1, &channel1,
                    GRAYSCALE_IMAGE, "Destination Image",0,&buffer2, &channel2);
```

```
            if ( ret )
            {
                int width = BUFFER_WIDTH(buffer1);
                int height = BUFFER_HEIGHT(buffer1);

                SetCursor(LoadCursor( NULL, IDC_WAIT));
                if ( ! buffer2 )
                {
                    tmp = (*CreateBuffer)( GRAYSCALE_IMAGE, width, height );
                    if (!tmp)
                        break;
                    channel2 = UC_BUFFER_CHANNEL(tmp,0);
                }
                else
                    tmp = buffer2;

                if ( width > BUFFER_WIDTH(tmp) ||
                    height > BUFFER_HEIGHT(tmp) )
                {
                    (*error_handler)(21);
                    break;
                }
                str_el = (*win_build_image)(3,3);
                if ( ! str_el )
                {
                    (*error_handler)(9);
                    break;
                }
                str_el[0][1] = str_el[1][0] = str_el[1][1] = str_el [1][2] = str_el[2][1] = 1;
                intermediate = (*win_build_image)( height, width );
                if ( ! intermediate )
                {
                    (*error_handler)(9);
                    break;
                }
                (*GetROI)(buffer1, &roitop, &roileft, &roibottom,&roiright);
                (*PrepareLibrary)(tmp);
                errcode = (*imerode)(channel1,intermediate,
                            str_el, 3, 3, roitop, roileft, roibottom+1, roiright+1);
                if ( errcode )
                    goto ERROR_EXIT;
                errcode = (*imdilate)(intermediate, channel2,
                            str_el, 3, 3, roitop, roileft, roibottom+1, roiright+1);
                if (errcode )
                    goto ERROR_EXIT;
                (*win_freeuc2)(intermediate);
                (*win_freeuc2)(str_el);
                (*RefreshBuffer)(tmp);
                SetCursor(LoadCursor(NULL,IDC_ARROW));
            }
            break;
ERROR_EXIT :
```

```
    (*error_handler)(errcode);
    break;

case COMMAND_REGION_GROW :
    if ( ! (*EnoughBuffers) (GRAYSCALE_IMAGE, 1) )
        break;

    ret = (*Get2Buffers1ParDialog) ( "Example RegionGrow",
            GRAYSCALE_IMAGE, "Source Image", NO_NEW | SUGGEST_ACTIVE_BUFFER,
            &buffer1, &channel1,
            GRAYSCALE_IMAGE, "Destination Image", 0, &buffer2, &channel2,
            "Threshold : ", t);
    if ( ret )
    {
        int width = BUFFER_WIDTH(buffer1);
        int height = BUFFER_HEIGHT(buffer1);

        if ( ! buffer2 )
        {
            tmp = (*CreateBuffer)(GRAYSCALE_IMAGE,width,height);
            if ( ! tmp )
                break;
            channel2 = UC_BUFFER_CHANNEL( tmp, 0);
        }
        else
            tmp = buffer2;

        if ( width != BUFFER_WIDTH(tmp) ||
            height != BUFFER_HEIGHT(tmp) )
        {
            (*error_handler)(21);
            break;
        }
        (*PrepareLibrary)(buffer1);
        (*GetROI)(buffer1, &roitop, &roileft, &roibottom, &roiright);
        errcode=clear(channel2, 0,roitop,roileft,roibottom,roiright);
        if ( errcode )
        {
            (*error_handler)(errcode);
            if ( ! buffer2 )
                (*DeleteBuffer)(tmp);
            break;
        }
        threshold = atoi ( t );
        RegionGrowWindow = (*RefreshBuffer)(buffer1);
        ClickCount = 1;
        CreateDialog ( ExampleInstance, MAKEINTRESOURCE(IDD_PIXELINPUT),
                        EikonaInfo->EikonaFrame, PixelInputProc);
        return;
    }
```

```
            break;
    }
}


_declspec(dllexport)
    char **EikonaDLLReturnMenu( void )
{
    return commands;
}


// Function to return the menu command handler of the DLL.
_declspec(dllexport)
    CMDHANDLERPROC EikonaDLLReturnCmdHandler(void)
{
    return ExampleCommandHandler;
}


_declspec( dllexport)
    MSGHANDLERPROC EikonaDLLReturnMessageHandler( DLLMSGRETURN_PROC f )
{
    DllMsgReturn = f;
    return ExampleMsgHandler;
}


BOOL WINAPI
    DllMain ( HANDLE hInstance, ULONG reason, LPVOID lpReserved )
{
    HMODULE EikonaModule;
    switch ( reason )
    {
        case DLL_PROCESS_ATTACH :

        // Get EIKONA's module handle. We need this to get
        // pointers to EIKONA exported functions.
            EikonaModule = GetModuleHandle ("eikona.exe");
            if ( ! EikonaModule )
                return FALSE;

        // Store our instance handle to a global variable so that
        // we can later acess our resources.
                ExampleInstance = hInstance;

        // Get pointers to functions we will use.
                Get2BuffersDialog = (GET2BUFFERSDIALOG_PROC)
                        GetProcAddress ( EikonaModule, GET2BUFFERSDIALOG);
                Get2Buffers1ParDialog = ( GET2BUFFERS1PARDIALOG_PROC)
                        GetProcAddress (EikonaModule, GET2BUFFERS1PARDIALOG);
                CreateBuffer = (CREATEBUFFER_PROC)
```

```
                GetProcAddress ( EikonaModule, CREATEBUFFER );
            DeleteBuffer = (DELETEBUFFER_PROC)
                GetProcAddress( EikonaModule, DELETEBUFFER );
            RefreshBuffer = (REFRESHBUFFER_PROC)
                GetProcAddress ( EikonaModule, REFRESHBUFFER );
            PrepareLibrary = (PREPARELIBRARY_PROC)
                GetProcAddress ( EikonaModule, PREPARELIBRARY );
            error_handler = (ERROR_HANDLER_PROC)
                GetProcAddress ( EikonaModule, ERROR_HANDLER );
            EnoughBuffers = ( ENOUGHBUFFERS_PROC )
                GetProcAddress ( EikonaModule, ENOUGHBUFFERS );
            GetROI = ( GETROI_PROC)
                GetProcAddress ( EikonaModule, GETROI);
            region_grow = ( REGION_GROW_PROC)
                GetProcAddress ( EikonaModule, REGION_GROW );
            imerode = (IMERODE_PROC)
                GetProcAddress ( EikonaModule, IMERODE);
            imdilate = (IMDILATE_PROC)
                GetProcAddress (EikonaModule, IMDILATE);
            clear = (CLEAR_PROC)
                GetProcAddress ( EikonaModule, CLEAR);
            win_build_image = (WIN_BUILD_IMAGE_PROC)
                GetProcAddress ( EikonaModule, WIN_BUILD_IMAGE);
            win_freeuc2 = (WIN_FREEUC2_PROC )
                GetProcAddress ( EikonaModule, WIN_FREEUC2);
            GetEikonaInfo = (GETEIKONAINFO_PROC)
                GetProcAddress ( EikonaModule, GETEIKONAINFO);
            CreateDialogMagnifier = ( CREATEDIALOGMAGNIFIER_PROC)
                GetProcAddress ( EikonaModule, CREATEDIALOGMAGNIFIER);
            DestroyDialogMagnifier = ( DESTROYDIALOGMAGNIFIER_PROC)
                GetProcAddress ( EikonaModule, DESTROYDIALOGMAGNIFIER);
            MagnifyInDialog = (MAGNIFYINDIALOG_PROC)
                GetProcAddress ( EikonaModule, MAGNIFYINDIALOG);

        // Get information about EIKONA windows.
            EikonaInfo = (*GetEikonaInfo)();
          default :
            return TRUE;
    }
}
```

References

[PIT93] I.Pitas *Digital image processing algorithms*, Prentice Hall, 1993.

[PIT93] I.Pitas, editor, *Parallel algorithms for digital image processing, computer vision and neural networks*, J. Wiley, 1993.

[PIT90] I.Pitas, A.N.Venetsanopoulos, *Nonlinear digital filters: Principles and applications*, Kluwer Academic, 1990.

[AND77] H.C.Andrews, B.R.Hunt, *Digital image restoration*, Prentice Hall, 1977.

[ANG90] E.Angel, *Computer graphics*, Addison-Wesley, 1990.

[BAL82] D.H.Ballard, C.M.Brown, *Computer vision*, Prentice Hall, 1982.

[FOL90] J.D.Foley, A.van Dam, S.K.Feiner, J.F.Hughes, *Computers graphics: Principles and practice*, Addison-Wesley, 1990.

[GON87] R.C.Gonzalez, P.Wintz, *Digital image processing*, Addison-Wesley, 1987.

[HAR87] S.Harrington, *Computer graphics: A programming approach*, McGraw-Hill, 1987.

[JAI89] A.K.Jain, *Fundamentals of digital image processing*, Prentice Hall, 1989.

[LEV85] M.D.Levine, *Vision in man and machine*, McGraw-Hill, 1985.

[LIN91] C.A.Lindley, *Practical image processing in C*, Wiley, 1991.

[MIC87] *Microsoft C: Runtime library reference*, Microsoft Press, 1987.

[NIB86] W.Niblack, *Digital image processing*, Prentice Hall, 1986.

[PRA91] W.K.Pratt, *Digital image processing*, Wiley, 1991.

[PRE88] W.H.Press, B.P.Flannery, S.A.Teukolsky, W.T.Vetterling, *Numerical recipes in C*, Cambridge University Press, 1988.

[RIM90] S.Rimmer, *Bit-mapped graphics*, Windcrest, 1990.

[ROS82] A.Rosenfeld, A.C.Kak, *Digital picture processing*, Academic Press, 1982.

[SCH89] R.J.Schalkof, *Digital image processing and computer vision*, Wiley, 1989.

[SER82] J.Serra, *Image analysis and mathematical morphology*, Academic Press, 1982.

[WIL87] R.Wilto, *Programmer's guide to PC and PS/2 video systems*, Microsoft Press, 1987.

[WYZ67] G.W.Wyzecki, W.S.Stiles, *Color science*, Wiley, 1967.

Index