INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

**REGION SEPARATION AND CONTOUR TRACING TECHNIQUES FOR COLORED
RASTER IMAGES**

by

RAVI H. KRISHNAMURTHY, B. E.

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Computer Science

Department of Computer Science

Carleton University

Ottawa, Ontario

February 1998

©copyright

1998, Ravi H. Krishnamurthy

**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

**395 Wellington Street
Ottawa ON K1A 0N4
Canada**

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

**395, rue Wellington
Ottawa ON K1A 0N4
Canada**

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-32388-9

The undersigned recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis

Region Separation and Contour Tracing Techniques For Colored Raster Images

submitted by **Ravi H. Krishnamurthy, B. E.**
in partial fulfillment of the requirements for
the degree of Master of Computer Science

Director, Department of Computer Science

Thesis Supervisor

CARLETON UNIVERSITY

FEBRUARY 1998

ABSTRACT

The recognition of objects embedded in a raster image has always posed a strong challenge to computer scientists. This is quite simply because of the complexity of the information present in such images. In order to recognize all of the objects in a given image, it is necessary to execute a series of steps to both establish the feature space from the training samples and to classify the test samples once the system is built. The specific series of steps used often depends on the application domain and the underlying philosophy of the approach used. In this thesis we propose a new way of developing image modules, namely by breaking up the given image into different objects using region separation techniques, detecting the edges of such objects that we identify and then tracing the object edges for contour generation.

Color separation is one of the most powerful region separation technique that can be used to identify objects in an image, based upon the adjacent gray scale value differences. While objects with scattered data cannot be identified with this method, in an application with a defined scope to establish only closed objects, as is the case with our work, this is very powerful for “fragmenting” the image data. Edge detection templates, such as the Sobel’s which we discuss in detail, play a key role in determining the edges of the objects. Even though edge detection can be used as a substitute for region separation techniques to separate objects embedded in an image by marking the edges as the boundaries, we have opted to proceed using a “region separation” method because it yields superior results. However, once the regions are clearly segregated with a region separation technique, the edge detection templates can be used to develop algorithms that aid in further processing the image data. Indeed, we use Sobel’s edge detection template to develop a new algorithm for generating the contour.

The combination of the methods and the algorithms that we discuss in our work can be used as preprocessing techniques for a variety of image processing applications. Thus, one could

use color separation and contour generation together to extract the shape of objects in order to build a pattern recognition system. In addition, all the methods which we discuss in our work can be applied to various specific image file formats, because we introduce our own internal representation of a generic file format, and provide a simple interface to transform the data from the various formats to the one we have designed.

ACKNOWLEDGMENTS

First and foremost, I would like to thank Dr. John Oommen, my supervisor, for his inspiration, ideas and enthusiasm. He spent many hours patiently discussing the thesis with me and his thoughtful insight and advice proved to be extremely valuable. I would also like to thank Dr. Oommen for the extra time and effort he put in to help me finish my thesis on time.

I owe thanks to my parents and my brother who all stood behind my efforts to enter into a reputed graduate school, and helped me in the best possible way to cherish my dream of completing my Masters degree. They provided a great deal of moral support and encouragement during my two years at Carleton. I really appreciate this and without this my thesis would not have crystallized. This thesis is dedicated to them.

I would like to extend my thanks to the administrative and support staff of the School of Computer Science at Carleton. In particular, I am grateful to Marlene Wilson who helped me with all the various administrative requirements. Finally, thanks to Igor Razumeiko, my pal at work, for his comments and useful suggestions.

TABLE OF CONTENTS

Chapter 1 Introduction.....	1
1.1 Overview.....	1
1.2 Our Work.....	2
1.3 Scope of Our Work.....	3
1.4 Organization	4
Chapter 2 Data Formats And The Generalized Contour Tracing Problem	7
2.1 Introduction	7
2.2 Goal of The Solution	7
2.3 Data Library and Formats.....	10
2.3.1 Graphics File Formats	11
2.3.2 Bitmap Files.....	13
2.4 Conclusion	25
Chapter 3 Interface Objects	27
3.1 Introduction	27
3.2 Microsoft Windows Bitmap	28
3.3 TIFF	28
3.4 Windows Bitmap and Color Palette	31
3.4.1 File Details.....	31
3.4.2 Color Palette	34
3.5 Generic Image Format	37
3.6 Reading Paletted Images.....	40

3.7 Results	48
3.8 Conclusion	48
Chapter 4 Color Separation	50
4.1 Introduction	50
4.2 Finding Basic Shapes.....	51
4.2.1 Edge Detection	51
4.2.2 Segmentation	52
4.3 Region Operations	53
4.3.1 Method 1	53
4.3.2 Method 2.....	54
4.4 Color Separation Technique	57
4.5 Algorithm Color Separation	58
4.6 Results	63
4.7 Conclusion	66
Chapter 5 Edge Detection And Tracing	67
5.1 Introduction	67
5.2 Edge Detection	68
5.3 Templates for edge detection.....	68
5.3.1 Convolution Principle.....	69
5.3.2 Simple Edge Detection	70
5.3.3 Sobel Edge Detection	72
5.3.4 The Edge Detection Algorithm.....	75
5.3.5 Results	79
5.3.6 Shortcomings of first-order edge detection	82
5.4 Tracing.....	84

5.4.1 Contour Tracing Algorithm	85
5.4.2 Results	97
5.5 Conclusion	98
Chapter 6 Design and Implementation Considerations	99
6.1 Introduction	99
6.2 Design considerations with GenericImage	100
6.3 Application Framework	105
6.4 User-Interface Components	107
6.4.1 Application User-Interface	107
6.4.2 Application Menu	108
6.4.3 Toolbar	110
6.4.4 Status Bar	111
6.4.5 Client Window	111
6.4.6 Dialogs	111
6.5 Conclusion	112
Chapter 7 Conclusion	114
7.1 Summary	114
7.2 Future Work	116
7.2.1 Automated color separation technique	116
7.2.2 Different region splitting methods	117
7.2.3 Elimination of sub-regions within the basic shape	117

LIST OF TABLES

TABLE 4.1: LENGTH OF CRACK BETWEEN REGIONS.....	56
TABLE 4.2: FULL LENGTH OF THE BOUNDARY OF EACH OF THE REGIONS.....	56
TABLE 5.1: IMAGE DATA AND THE ASSOCIATED GRADIENT DIRECTION.....	72

LIST OF FIGURES

FIGURE 1.1: UNIFORMLY AND NON-UNIFORMLY COLORED IMAGES.....	3
FIGURE 2.1: THE IMAGE OF A FLOWER	8
FIGURE 2.2: COLOR SEPARATED PETALS OF THE FLOWER OF FIGURE 2.1	9
FIGURE 2.3: COLOR SEPARATED MIDRIB OF THE FLOWER OF FIGURE 2.1	9
FIGURE 2.4: EXHIBIT 2.....	10
FIGURE 2.5: BITMAP DATA	19
FIGURE 2.6: CONTIGUOUS SCAN LINES.....	21
FIGURE 2.7: STRIPS OF SCAN LINES.....	22
FIGURE 2.8: TILES OF SCAN LINES.....	23
FIGURE 2.9: ORGANIZATION OF PIXEL DATA INTO COLOR PLANES.....	24
FIGURE 3.1: TIFF FILE ORGANIZATION.....	29
FIGURE 3.2: TIFF FILE DATA STRUCTURE LINKS.....	30
FIGURE 3.3: PALETTE AND COLOR MAPPING	35
FIGURE 3.4: INTERACTION BETWEEN DIFFERENT FILE FORMATS WITH OUR APPLICATION.....	37
FIGURE 4.1: COLOR SEPARATION OF COMPONENTS OF EXHIBIT 1 AT THRESHOLD = 20	64
FIGURE 4.2: COLOR SEPARATION OF COMPONENTS OF EXHIBIT 1 AT THRESHOLD = 40	64
FIGURE 4.3: COLOR SEPARATION OF COMPONENTS OF EXHIBIT 1 AT THRESHOLD = 80	64
FIGURE 4.4: COLOR SEPARATION OF EXHIBIT 2 AT THRESHOLD = 20.....	65
FIGURE 4.5: COLOR SEPARATION OF EXHIBIT 2 AT THRESHOLD = 40	65
FIGURE 4.6: COLOR SEPARATION OF EXHIBIT 2 AT THRESHOLD = 80	65
FIGURE 5.1: COLOR SEPARATED COMPONENTS OF EXHIBIT 1	79
FIGURE 5.2: COLOR SEPARATED COMPONENT OF EXHIBIT 2	79
FIGURE 5.3: EDGE DETECTED COMPONENTS OF EXHIBIT 1 AT THRESHOLD = 5	80
FIGURE 5.4: EDGE DETECTED COMPONENTS OF EXHIBIT 1 AT THRESHOLD = 127	80

FIGURE 5.5: EDGE DETECTED COMPONENTS OF EXHIBIT 1 AT THRESHOLD = 250	80
FIGURE 5.6: EDGE DETECTED COMPONENT OF EXHIBIT 2 AT THRESHOLD = 5	81
FIGURE 5.7: EDGE DETECTED COMPONENT OF EXHIBIT 2 AT THRESHOLD = 127	81
FIGURE 5.9: EDGE DETECTED COMPONENT OF EXHIBIT 2 AT THRESHOLD = 250	81
FIGURE 5.9: TRACED COMPONENTS OF EXHIBIT 1.....	97
FIGURE 5.10: TRACED COMPONENTS OF EXHIBIT 2.....	97
FIGURE 6.1: APPLICATION USER INTERFACE.....	107
FIGURE 6.2: MAIN MENU.....	108
FIGURE 6.3: FILE MENU.....	108
FIGURE 6.4: EDIT MENU	109
FIGURE 6.5: VIEW MENU.....	109
FIGURE 6.6: IMAGE MENU	109
FIGURE 6.7: WINDOW MENU	110
FIGURE 6.8: TOOL BAR.....	110
FIGURE 6.9: STATUS BAR	111
FIGURE 6.10: TOLERANCE DIALOG.....	111
FIGURE 6.11: THRESHOLD DIALOG.....	112
FIGURE 7.1: CONTOUR OF A COLOR SEPARATED COMPONENT OF EXHIBIT 1	117

LIST OF ACRONYMS

BMP Microsoft Windows Device Independent Bitmap.

CAD Computer Aided Design.

CPU Central Processor Unit.

GIF Graphics Interchange Format.

JPEG Joint Photographic Experts Group.

PCX PC Paintbrush file format.

TGA Truevision Graphics Adaptor.

TIFF Tag Image File Format.

WMF Windows Metafile.

GLOSSARY

Compression: A means of reducing the storage for an image so that it can be held in a smaller disk space or transmitted from station to station at a faster rate.

Contour: The pixels that are of the same color or grey level that lie on the edge of a region of the same color or grey level.

Flood fill: An algorithm that, given a pixel position which lies inside a closed contour, finds the position of all the other pixels in the contour and alters their grey level.

Grey Level: A value in a pixel that represents the darkness of the pixel. Normally zero is the darkest value and 2^n is the lightest value where n is an integer, typically a multiple of 4.

Hough Transform: A coordinate change that enables shapes of known dimensions to be found in an image.

Image: A two-dimensional array of pixels (if it is inside the memory of a computer) or a real view that may or may not be digitized. Formally, it may also be described as a two-dimensional light intensity function, or color function denoted by $f(x,y)$, where x and y denote spatial coordinates.

Noise: Undesirable errors in pixel values in an image, typically caused by electrical interference when the image was captured.

Object: A real entity that can be seen partially or wholly in an image.

Quantization: The allocation of grey levels to classes such that the number of initial grey levels is greater than the number of classes. Typically, if the number of grey levels in an image is 256 and the display only shows at maximum 16 grey levels, then a simple quantizing function is to divide the grey levels by 16 and truncate the result.

Pixel: A picture element. It is normally considered to be the smallest addressable part of an image. It can be logical or physical - i.e. it can be the memory allocated to holding an element of

an image or it can be the spot on the screen that can display one of a range of grey levels or colors.

Region: An area in an image.

Resolution: This may refer to the screen size, i.e. the number of horizontal and vertical pixels, or to the number of bits that are used to represent each pixel, giving the number of grey levels or color levels available.

Template: An array of values, normally used to perform a set of arithmetic operations on part of an image before being stepped across the image to perform the same operations again and again.

Thresholding: A quantizing of old grey values into new grey values so as to increase the contrast between selected grey levels.

Transformation: The changing of grey levels in pixels either as a result of statistical operation or a geometric operation. The geometric transformations are normally considered to be translation, rotation, scaling, skewing, and reflection.

CHAPTER 1

INTRODUCTION

1.1 Overview

The storage strategy used for most graphics applications is bitmap based, and the displays used are typically raster based. This is due, in part, to the availability of high-speed CPUs, inexpensive memory and mass storage, and high-resolution input and output devices. Furthermore, bitmap graphics is also driven by the need to manipulate images obtained from raster digitizing devices. Bitmap graphics are important in applications supporting CAD and 3D rendering, business charts and presentations, 2D and 3D modeling, computer art and animation, graphical user interfaces, video games, electronic document image processing, and image processing and analysis. For more extensive information about image processing and its applications, we refer the interested reader to [AL, GW, GB1, NW].

For the same reasons mentioned above, it is interesting to note that, though by volume, the greatest amount of data being stored and exchanged consists of finished images in bitmap format, there has been an increased emphasis on bitmap graphics [RS].

There are several bitmap file formats that are currently being used to store bitmap images. The complexity of storing the bitmap data within a file varies considerably from one format to another. Windows BMP, for instance, has the simplest representation of an image while formats such as TIFF and JPEG use complex data storage techniques to provide improved versatility.

Image processing, in its core, deals with manipulating images, storing them in formats that are appropriate for subsequent processing, and editing and enhancing images. Some of the

major applications of image processing include photography, publishing/prepress, and the automated creation of special effects in video and films.

Image tracing is one of the interesting and most important aspects of image processing fundamental to many applications. It involves walking through the image data to generate one or more contours based on a variety of methods. Tracing is useful in finding basic shapes embedded in an image that could be subsequently used to further enhance the image or create a new image. For example, from a pattern recognition perspective, tracing is a preprocessing operation necessary to extract features from the objects embedded in an image and to recognize the patterns associated with them. Similarly, in a photo-editing application, tracing is necessary to export bitmap files to vector formats. Therefore, due to the increased importance, sophisticated tracing algorithms are being developed to enable the user to generate complex images that can be used for further processing.

1.2 Our Work

Tracing is usually performed on an image in which the objects or patterns are already established. Therefore, to build a system that can derive benefits from tracing, it is very important to decide how the objects are identified. For instance, in recognizing patterns in an image, it is important to separate the objects based on color, contrast, hue, etc. This decision on how the objects are identified, of course, varies from one application domain to another.

In our work, we have made an effort to build a system that reads an image file, identifies the object(s) embedded in it, and then trace the resulting object(s) to generate the associated contour(s). Since we do not intend to confine the scope of our application to any specific image file format viz. RGB, CMYK, Black-and-White, etc., first and foremost, we design a generic

class structure which acts as a place-holder for files of different formats. The intention of using such a structure, of course, is to keep our algorithms independent of the specific file formats. In order to separate an image into one or many objects, we have used a color-enhanced separation technique in which we identify all pixels that lie within a specific tolerance of a single color value to be one object. We have developed a novel tracing algorithm that generates the contour for these objects in one-pass using Sobel's edge detection templates. For the purpose of comparing the tracing results, we have also presented the edge detection algorithm that works on the image data using Sobel's edge detection templates.

1.3 Scope of Our Work

Since we use color separation to identify the objects contained in an image, we quite obviously, can only identify those objects that have clearly marked color boundaries, and thus are closed. To understand this better, consider the following two images:

Figure 1.1: Uniformly and Non-uniformly Colored Images.

In the first image in Figure 1.1 (that of a bird), the object that we identify as a bird is uniformly represented by one color. However, in the second image, the fish that is depicted has scattered regions with the same color. In such cases, we would not be able to recognize the entire fish as one object, unless we utilize some other strategy to identify the whole region as an object. Indeed, we define the goal of our application to work only on images that have closed objects in the form of distinct color separations. Therefore, as long as the objects that are to be identified in an image are clearly defined with color separations, the objects we identify after the color separation process and the contours that we generate for such objects give a fairly good representation of the object. Consequently, in using this application to build a pattern recognition system, the designer needs to pay attention during the training and testing process to ensure that the patterns that we are trying to recognize have distinct color separations. Therefore, in its existing scope, our solution is not effective when

- (i) the uniform region is very small,
- (ii) multiple uniform regions share the same color values, and
- (iii) there is a uniform color gradient across the image.

1.4 Organization

In order to understand the image processing techniques, it is first essential to understand how the image data is stored in the file. In this context, it is useful to get an overview of the different graphic file formats that are currently being used. In Chapter 2, which deals with data formats and the tracing problem in general, we shall give a brief introduction to some of the widely used graphic file formats, with special emphasis on bitmap file formats since we intend to use it in the remainder of our work.

Not only are the Windows Bitmap and TIFF the most commonly found file formats currently in use, they also exhibit a wide range of diversity in terms of the way the bitmap image data is stored. Therefore, we have placed special emphasis on these two formats in Chapter 3, which deals with interface objects, and we study the details of these two formats. The diversity of data representation in these two files, indeed, exposes the fact that almost all file formats have different ways of storing the image representation. As a result of this fact we are led to define our own internal image structure, which we have called the *GenericImage*, that will interface our application with specific data representation in different file formats. We shall present the algorithms that read these two file formats to our structure

Edge detection and Segmentation are two of the most commonly used techniques to identify the regions or objects in an image that are of specific interest. In segmentation, we identify a group of pixels based on certain characteristic features in order to establish the objects. Similarly, in Edge Detection, we identify the edges of the object(s) in an image, and follow them through so as to define the boundary. We shall discuss the merits and the demerits of these two techniques in Chapter 4, and explain why we have chosen a segmentation technique, in the form of color separation, over the other in building our system. We then move on to discuss the color separation technique itself in detail, and present the algorithm along with the results of applying this algorithm to sample images.

We then study the contour generation problem in Chapter 5. Edge Detection in itself has a very interesting effect on a bitmap image. The idea behind edge detection is that we apply a specific edge detection template on a source image in order to convolve the data and store the resulting value in a destination image. The convolution principle, which is the foundation for producing such effects on an image data, is presented in the beginning of this chapter. In order to generate the contour of object(s) embedded in an image, one can opt to detect the edges that

enclose them in the first pass and then use a trivial tracing algorithm to blindly follow the edges. Alternatively, one can opt to trace the image as we perform the edge detection. We have chosen the latter method of generating the contour. In this chapter, we have also discussed the edge detection templates that are used to generate the edges in an image. Since we base our solution to the problem on Sobel's edge detection templates, we have presented the latter in detail. Following this discussion, we have presented the edge detection and tracing algorithms along with the results that we have obtained.

Chapter 6 deals with the design and implementation considerations involved in creating this system. We consider the advantages of implementing our *GenericImage* in the form of a class and discuss the merits of using such a class-based architecture. This is done so as to provide a unique interface to our algorithms so that they remain unchanged with either the addition of extra fields of data, or the new interface routines which read different bitmap file formats. This chapter also presents an overview of our application framework and the user-interface components.

CHAPTER 2

DATA FORMATS AND THE GENERALIZED CONTOUR TRACING PROBLEM

2.1 Introduction

In this chapter, we shall primarily focus on two aspects that are very important to start our work - an understanding of the nature of the problem and a knowledge of the raster image file formats.

In Section 2.2, we shall formally define the problem that we try to address in the rest of our work and briefly discuss the proposed goal of the solution.

In Section 2.3, we will discuss some of the different graphic file formats that are in existence. Since we base our entire work on manipulating bitmap image data, an insight into bitmap file format is very essential. Therefore, in this section, we shall also discuss the internals of a general bitmap file structure in detail. In Chapter 3, we will get into the specific bitmap files of interest.

2.2 Goal of The Solution

The complexity of images coupled with the vivid colors that are found in them makes it really difficult to take any single approach towards identifying the various shapes which are embedded in them. As an example, consider the image of a flower consisting of two colors shown in Figure 2.1 below.

Figure 2.1: The Image of a Flower

We can pose the question we encounter as follows: how do we separate this flower into useful shapes so that they can be used for subsequent processing? The problem becomes interesting because, we would want to use these basic shapes to establish a particular pattern, which for example, may be a flower. Thus, though it is not necessary, most often, separating an image based upon color, contrast, and hue will give sufficiently well defined shapes. For example, in the figure above, it is easy to see that even an untrained human eye can perceive two distinct portions of the flower, namely the petals and the midrib. Such decomposition can be made solely on the basis of the color and color changes.

When we color separate the above image, we get the following two distinct images - one consisting of the petals (see Figure 2.2) and the other consisting of the midrib (see Fig 2.3).

Figure 2.2: Color separated petals of the flower of Figure 2.1

Figure 2.3: Color separated midrib of the flower of Figure 2.1

The primary objective of our work is to develop an algorithm for generating one or more basic shapes and associated contours found in an image. Thus, in the above example, our task would be to generate the contours of two basic shapes such as those shown in Figures 2.2 and 2.3 respectively. The solution to the problem will be obtained by creating color masks for an image so as to separate the image into its composite layers. Each layer will then be traced to obtain the appropriate contour.

We would like to mention here that in the remainder of this thesis work, we intend to use the images presented in Figures 2.1 and 2.4 (which is shown below) as sample sources for testing our algorithms and refer to them as “Exhibit 1” and “Exhibit 2” respectively.

Figure 2.4: Exhibit 2

2.3 Data Library and Formats

In order to proceed with color masking and tracing techniques, it is very essential to understand how data is internally organized into streams of bits in a bitmap. Although we operate with each individual pixel in a bitmap, in order to determine the associated data, we need to know how we can access the data resident in the storage device. To achieve this, we need an understanding of the different graphics file formats.

Once we have read a graphics file from the archived data storage medium, we shall store the relevant information in memory in our internal data structure. We discuss the storage of data in memory in the next chapter. In this chapter we shall emphasize and highlight the various existing graphics file formats.

We shall now present a brief description of the various file formats which are currently used to represent bitmap structures. Although we consider Windows BMP files in the development of our application, in our exposition, we shall utilize a generic data structure to represent the bitmap format so as to ensure that any of the available graphics file formats can be mapped to our data structure. Thus, if we are provided with an appropriate interface, our solution would be applicable for a variety of file formats currently available.

2.3.1 Graphics File Formats

There are a number of different types of graphics files. Each type stores graphics data in a different way. Bitmap, vector and metafile formats are, by far, the most commonly used formats. The increasing popularity of the World Wide Web has made some of the other formats such as the various animation, multimedia and hypertext formats, more popular. We shall briefly explain the salient features of each of these formats.

Bitmap Formats:

Bitmap formats are used to store graphics data completely specified by bitmaps. Files of this type are particularly well-suited for the storage of real-world images such as photographs and video images. Bitmap files, sometimes called *raster files*, essentially contain an exact pixel-by-pixel map of an image. A rendering application can subsequently reconstruct this image on the display surface of an output device. An excellent store of information about these formats is given in [KL].

Microsoft BMP, PCX, TIFF, and TGA are examples of commonly used bitmap formats.

Vector Formats:

Vector format files are particularly useful for storing line-based elements such as piecewise linear diagrams and polygons, or those that can be decomposed into simple geometric objects, such as text. Vector files contain mathematical descriptions of image elements, rather than pixel values. A rendering application uses these mathematical descriptions of graphical shapes to construct a final image. In general, vector files are structurally simpler than most bitmap files and are typically organized as data streams.

AutoCAD DXF is an example of a commonly used vector format.

Metafile Formats:

Metafiles can contain both bitmap and vector data in a single file. The simplest metafiles resemble vector format files; they provide a language or grammar that may be used to define vector data elements, but they may also store a bitmap representation of an image. Metafiles are frequently used to export bitmap or vector data between software applications.

Examples of commonly used metafile format are the WMF and Macintosh PICT formats.

Animation Formats:

The basic idea of animation formats is to rapidly display one image superimposed over another to make it appear as if the objects in the image are moving. Very primitive animation formats store entire images that are displayed in sequence, usually in a loop. Slightly more advanced formats store only a single image, but also store multiple color maps of the image. By loading in a new color map, the colors in the image change, and the objects appear to move. Advanced animation formats store only the differences between the two adjacent images and update only the pixels that have actually changed as each frame is displayed.

AVI and TDDD are examples of animation file formats.

Multimedia Formats:

Multimedia formats are relatively new, but are becoming increasingly important. They are designed to allow the storage of different types of data in the same file. Multimedia formats usually allow the inclusion of graphics, audio, and video information.

MPEG and Apple's QuickTime are examples of this file format category.

We shall now spend a little time to examine the general structure of a bitmap file.

2.3.2 Bitmap Files

Bitmap files vary greatly in their details, but they all share the same general structure. Bitmap files consist of a header, bitmap data, and other information, which may include a color palette and other data.

The basic components of a simple bitmap files are the following:

Header
Bitmap Data

If a file contains no image data, only a header will be present. If additional information is required that does not fit in the header, a footer will usually be present as well:

Header
Bitmap Data
Footer

An image file may store a palette in the header, but it will more likely appear immediately after the header:

Header	
Palette	Bitmap Data
Footer	

A palette can also appear immediately after the image data, like a footer, or be stored in the footer itself as follows:

Header
Bitmap Data
Palette

Scan-line tables and *color correction tables* sometimes appear after the header and before or after the image data. In such a case the file would be as:

Header			
Palette	Scan Line Table	Color Correction Table (here)	Bitmap Data
Footer			

If an image file format is capable of holding multiple images, an *image file index* may appear after the header, holding the offset values of the starting positions of the images in the file:

Header				
Palette	Bitmap Index	Bitmap 1 data	...	Bitmap n data
Footer				

Finally, if the format definition allows each image to have its own palette, the palette will most likely appear before the image data with which it is associated. In such a case, the following would be the structure of this general file:

Header	
Palette	Bitmap Index
Palette 1	Bitmap 1 data
...	...
Palette n	Bitmap n data
Footer	

We shall now spend some time considering each of these elements in detail.

Header

The header is usually a section of ASCII-format data normally found at the beginning of the file, and it contains specific information about the bitmap data found elsewhere in the file. All bitmap files have some sort of header, although the format of the header and the information stored in it varies considerably from format to format. Typically, a bitmap header is composed of fixed fields commonly containing the following information:

Header
Palette
Bitmap Index
Palette 1
File Identifier
File Version
Number of Lines per Image
Number of Pixels per Line
Number of Bits per Pixel
Number of Color Planes
Compression Type
X Origin of Image
Y Origin of Image
Text Description
Unused Space

We shall look at the significance of each of these fields.

File Identifier

The header usually starts with some sort of unique identification value called a **file identifier**, **file ID**, or **ID value**. Its purpose is to allow a software application to determine the format of the particular graphics file being accessed. The ID values can be a series of ASCII

characters, such as BM or GIF, or a 20 or 4-byte word value such as 4242h. The pattern is usually unique.

File Version

Following the identification value in the header is usually a field containing the **file version**. Naturally enough, successive versions of bitmap formats may differ in characteristics such as header size, bitmap data supported, and color capability. After having verified the file format through the ID value, an application will typically examine the version value to determine if it can handle the image data contained in the file.

Image Description Information

Image description information consists of a sequence of fields that describe the image itself. Bitmaps are usually organized, either physically or logically, into lines of pixels. *Number of lines per image*, also called *image length* or *image height*, holds a value corresponding to the number of lines comprising the actual bitmap data. Similarly, the *number of pixels per line*, also called the *image width* or *scan-line width*, indicates the number of pixels stored in each line.

The *number of bits per pixel* indicates the size of the data needed to describe each pixel per color plane. This may also be stored as the *number of bytes per pixel*, and is more properly called *pixel depth*. If the bitmap data is stored in a series of planes, the *number of color planes* indicates the number of planes used. Often the value defaults to one.

The number of bits in a line of the image can be calculated by computing the product of the *number of bits per pixel*, the *number of pixels per line*, and the *number of color planes*. The number of bytes per scan line can then be determined by dividing the resulting product by eight.

Compression Type

If the bitmap format supports the compression of the data (to reduce the physical size), the **compression type** field will be found in the header. Most of the bitmap formats allow

multiple compression types. Windows BMP, TIFF, and many other commonly used formats allow compression of bitmap data.

x and y origins

The **x and y origin** of an image specify a coordinate pair on the output device where the data is to be rendered. Most commonly, it will start at (0,0), the origin of the output device. Changing this value causes the image to be displayed at different locations when it is rendered.

Most bitmap formats are designed to model an actual or virtual device in the form of a *device context* or *drawing surface*. The drawing surface has an implied origin, which defines the starting position of the image, and an implied orientation, which defines the direction in which successive lines are drawn as the output image is rendered. While some formats place the origin in the upper-left corner of the display surface, others use the lower-left corner or the center of the drawing surface to represent the origin.

Unused Space

Most formats contain an additional unused field, sometimes referred to as the *padding* or the *filler* that contains no data. This can be used to incorporate new data while still maintaining backward compatibility with programs supporting older versions of the format. This is a common method used to minimize version problems.

Bitmap Data

The actual bitmap Data consists of numerical values specifying the colors of the individual *pixels* or *picture elements (pels)*. Pixels are dots of color arranged on a regular grid in a pattern representing the form to be displayed. Technically, the bitmap consists of an *array of numerical values* used to set, color, or “turn on” the corresponding pixels on an output device when the bitmap is rendered. *Bit depth* determines the number of colors a pixel value can

represent. A 1-bit pixel can be one of the two colors, a 4-bit pixel one of the 16 colors, and so on.

The most commonly used pixel depths are 1, 2, 4, 8, 16, 24, and 32.

Bitmap data in file:

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 255, 255, ...

Figure 2.5: Bitmap Data

In many bitmap file formats, the actual bitmap data is found immediately after the end of the file header. It may be found elsewhere in the file, however, to accommodate a palette or other data structure which also may be present. If this is the case, an offset value will appear in the header indicating where to find the beginning of the image data in the file.

The actual data is normally organized in one of the two ways in a scan-line manner or planar manner, each of which are explained below.

Scan line data

The first, and simplest method to store bitmap data is the organization of pixel values into rows or scan lines. If we consider every image to be made up of one or more scan lines, the pixel data in the file describing that image will be a series of sets of values, each set corresponding to a row of the image. Multiple rows are represented by multiple sets written from start to end in the file. This is the most common method for storing image data which is organized as a sequence of rows.

If we know the size of each pixel in the image, and the number of pixels per row, we can calculate the offset of the start of each row in the file. For example, in an 8-bit image every pixel is one byte long. If the image is 19 pixels wide, rows in the file are represented by sets of pixel values 19 bytes wide. Rows in the file starts at offsets of 0, 19, 38, 57, etc. bytes from the start of the bitmap data.

In some formats, however, rows of image data must be some even-byte multiples in length. An example is the common rule requiring bitmap row data to end on long-word boundaries, where a long word is four bytes long. In practice, the storage inefficiency arising out of this situation is compensated for by an increase in the speed gained while working with image data that is multiple of 2 or 4.

In a 24-bit image, each image pixel corresponds to a 3-byte long pixel value in the file. Many times, they are stored as a series of 4-byte long pixel values for two reasons:

- The storage is long-word aligned, which may significantly improve the performance of data manipulation.
- This extra byte can be used to store transparency value associated with the byte.

Scan line data can be further reorganized as: *contiguous data*, or *strips*, or *tiles*.

Contiguous data

The simplest method of row organization is where all of the image data is stored contiguously in the file, one row following the last. To retrieve the data, file is read row by row, which delivers rows in the order in which they were written. The data in this organizational scheme is stored in the file equivalent of a 2D array. Data in the file can be indexed into, if the width of the row in pixels and size of the pixel in bytes are known.

Figure 2.6: Contiguous scan lines

The advantage of this pattern is that the data can be read quickly, in large chunks, and assembled in memory quite easily.

Strips

In this method of organization, images are stored in strips, which also consist of rows stored contiguously. The total image, however, is represented by more than one strip, and the individual strips may be widely separated in the file. Strips divide the image into number of segments, which are always just as wide as the original image.

Figure 2.7: Strips of scan lines

Strips make it easier to manage image data on machines with limited memory. An image 1024 rows long, for instance, can be stored in file as eight strips, each strip containing 128 rows. Arranging an image into strips helps buffer the data for easier and faster manipulation.

In applications that reads compressed data in to a buffer and then decode it into another buffer that is of the same size or larger than the first, arranging the compression on a per strip basis greatly reduces the memory constraints.

Tiles

Another method of bitmap data organization is tiling. Tiles are similar to strips in that each is a delineation of a rectangular area of an image. However, unlike strips, which are always the width of the image, tiles can have any width; from single pixel to the entire image. In practice, tiles are arranged so that the pixel data corresponding to each tile is between 4Kb and 64Kb in size. Also, when an image is tiled, generally all tiles that make up the image are of the same size.

Figure 2.8: Tiles of scan lines

Tiling allows faster decoding and decomposition of larger images.

Planar data

The second method of pixel value organization involves the separation of image data into two or more planes. Files in which the bitmap data is organized in this way are called planar files.

A *composite image*, an image with many colors, can be represented by three blocks of bitmap data, each block containing just one of the component colors making up the image. Constructing each block is akin to the photographic process of making a separation - using the filters to break up a color photograph into a set of component colors, usually three in number. The original photograph can be reconstructed by combining the three separations, as in the simpler storage method explained above; in this case, more than one block is now needed to reconstruct the image. The blocks may be stored consecutively or may be physically separated from one another in the file.

Figure 2.9: Organization of pixel data into color planes

As an example, a 24-bit image two rows by three columns wide might be represented in RGB format as six RGB pixel values:

(0x02, 0xFF, 0x04) (0x03, 0x03, 0x02) (0x00, 0x0E, 0xFF)

(0x09, 0x0A, 0x00) (0x06, 0x07, 0x00) (0x00, 0x0D, 0x0A)

but be written to the file in planar format as follows:

red plane

(0x02) (0x03) (0x00)

(0x09) (0x06) (0x00)

green plane

(0xFF) (0x03) (0x0E)

(0x0A) (0x07) (0x0D)

blue plane

(0x04) (0x02) (0xFF)

(0x00) (0x00) (0x0A)

Notice that the exact same data is being written; it's just arranged differently. Each method takes up exactly the same amount of space.

In practice, most bitmap files are stored in a non-planar format. Supporting hardware will usually disassemble the pixel data to create multiple color planes in memory.

Footer

The **footer** is a data structure similar to a header and is often an addition to the original header, but appended to the end of a file. A footer is generally added when the file format is upgraded to accommodate new types of data and it is no longer convenient to add or change information in the header. It is mainly included to support backward compatibility with previous versions of the file format.

Besides headers, footers, and palettes, bitmap files may contain additional data structures, which are usually added to aid manipulation of the image data by the rendering application. A file format which allows more than one image in the file needs to provide some method of identifying the start of each image.

2.4 Conclusion

Bitmap files are specially suited for the storage of real-world images. Any application that needs manipulation of image data needs a clear understanding of the bitmap file formats, and image tracing is no exception. In this chapter we have considered many bitmap file formats that are used in common practice. This knowledge should help us build our own structure to represent any bitmap file which is to be used by a typical application. In the next chapter, we shall create

our own image representation structure which takes advantages of the insight we have gained in this chapter.

CHAPTER 3

INTERFACE OBJECTS

3.1 Introduction

In the last chapter, we discussed the various graphic file formats that are currently in use. In this chapter, we shall present two specific graphic file formats in detail and later take a look at the organization of color palettes in bitmaps. Subsequently, we shall present the data structure which we intend to utilize, which interfaces our application to diverse file formats.

In Sections 3.2 and 3.3, we shall consider Microsoft Windows Bitmap and TIFF formats to highlight the diversity of structural formats of different graphics files in existence. Our focus in Section 3.4 is to catalogue the specific characteristics of the Windows Bitmap format, which is used in our application to accomplish the goal described in the previous chapters. This section also presents an insight into color palettes and issues associated with them.

In order to be able to manipulate image data in different graphics files that have different file formats, we need to develop our own data structure, one that is generic enough to store image information contained in any of them in a way that our application can easily interpret it. Indeed, this is a common technique being used by most image processing applications. In Section 3.5, we present our generic structure that will be used for this specific purpose.

In Section 3.6, we present the algorithm that reads file information into the data structure which we have proposed. Section 3.7 contains a note on the results and we conclude this chapter in Section 3.8.

3.2 Microsoft Windows Bitmap

The Microsoft Windows Bitmap (BMP) file format is one of the several graphics file formats supported by the Microsoft Windows operating environment. BMP is the native bitmap format of Windows, and is used to store virtually any type of bitmap data. We consider BMP Version 3 for the purpose of our discussion. We refer the reader to [MS] for a more detailed understanding of Windows Bitmap format.

Windows BMP files contain four sections: a file header, a bitmap information header, a color palette, and the bitmap data. Of these four sections, only the palette information is optional, depending on the bit depth of the bitmap data. The bitmap file header is 14 bytes in length and is followed by a second header called the bitmap header. The organization of the file is as follows:

File Header
Bitmap Header
Color Palette
Bitmap Data

3.3 TIFF

TIFF files are organized into three sections: the Image File Header (IFH), the Image File Directory (IFD), and the bitmap data. Of these three sections, only the IFH and IFD are required. It is therefore quite possible to have a TIFF file that contains no bitmap data at all, although such a file would be highly unusual. A TIFF file that contains multiple images has one IFD and one bitmap is stored per image. TIFF has a reputation for being a complicated format, this being is in part, because the location of each Image File Directory and the data that the IFD points to --

including the bitmapped data – may vary. In fact, the only part of a TIFF file that has a fixed location is the Image File Header, which is always the first eight bytes of every TIFF file. All other data in a TIFF file is found by using information found in the IFD. Each IFD and its associated bitmap are known as a TIFF subfile. There is no limit to the number of subfiles that a TIFF image file may contain.

Each IFD contains one or more data structures called *tags*. Each tag is a 12-byte record that contains a specific piece of information about the bitmapped data. A tag may contain any type of data and tags are always found in contiguous groups within each IFD.

Figure 3.1: TIFF file organization

Figure 3.1 shows three possible arrangements of the internal data structure of a TIFF file containing three images. In each example, the IFH appears first in the TIFF file. In the first example, each of the IFDs has been written to the file first and the bitmaps last. This arrangement is the most efficient for reading IFD data quickly. In the second example, each IFD is written, followed by the respective bitmap data. This is perhaps the most common internal format of a multi-image TIFF file. In the last example, the bitmap data is written first, followed by the IFDs. This seemingly unusual arrangement might occur if the bitmap data is available to be written before the information that appears in the IFDs.

Each IFD is a road map of where all the data associated with a bitmap can be found within a TIFF file. The data is found by reading it directly from within the IFD data structure or by retrieving it from an offset location whose value is stored in the IFD. The offset values used in a TIFF file are found in three locations. The first offset value is found in the last four bytes of the header and indicates the position of the first IFD. The last four bytes of each IFD is an offset value to the next IFD. The last four bytes of each tag may contain an offset value to the data it represents, or possibly the data itself.

Figure 3.2 shows the way the data structures of a TIFF file are linked together.

Figure 3.2: TIFF file data structure links

We refer the reader to [TF1], [TF2] for a more detailed overview of TIFF file organization.

3.4 Windows Bitmap and Color Palette

3.4.1 File Details

The Windows bitmap file header has the following structure:

```
typedef struct _WinBMPFileHeader
{
    WORD    FileType;
    DWORD   FileSize;
    WORD    Reserved1;
    WORD    Reserved2;
    DWORD   BitmapOffset;
} WINBMPFILEHEADER
```

FileType holds a 2-byte “magic value” used to identify the file type; it is always 4D42h or “BM” in ASCII.

FileSize is the total size of the BMP file in bytes and should agree with the file size supported by the system. This field only stores a useful value when the bitmap data is compressed, and this value is usually zero in uncompressed BMP files.

Reserved1 and **Reserved2** do not contain useful data and are usually set to zero in a BMP header written to disk. These fields are instead used by an application when the header is read into memory.

BitmapOffset is the starting offset (in bytes) of the bitmap data from the beginning of the file.

Following the file header is a second header called the **Bitmap Header**. This header contains information specific to the bitmap data and is structured as below:


```
typedef struct _Win3xBitmapHeader
{
    DWORD   Size;
    LONG    Width;
    LONG    Height;
    WORD    Planes;
    WORD    BitsPerPixel;
    DWORD   Compression;
    DWORD   SizeOfBitmap;
    LONG    HorzResolution;
    LONG    VertResolution;
    DWORD   ColorsUsed;
    DWORD   ColorsImportant;
} WIN3XBITMAPHEADER
```

Size is the size of the bitmap header in bytes.

Width and **Height** are the width and height of the image in pixels, respectively. If the height is a positive number, the image is a “bottom-up” bitmap with the origin in the lower-left corner. If the height is a negative number, then the image is a “top-down” bitmap with the origin in the upper-left corner. Width does not include any scan-line boundary padding.

Planes is the number of color planes used to represent the bitmap data. BMP files contain only one color plane, and so this value is always unity in such files.

BitsPerPixel is the number of bits in each pixel.

Compression indicates the type of encoding method used to compress the bitmap data. 0 indicates that the data is uncompressed; 1 indicates that the 8-bit RLE algorithm is used; 2 indicates that the 4-bit RLE algorithm is used. The reader is encouraged to refer to [MV] for an overview of various image compression techniques.

SizeOfBitmap is the size of the stored bitmap in bytes. This value is typically zero when the bitmap data is uncompressed.

HorzResolution and **VertResolution** are the horizontal and vertical resolutions of the bitmap in pixels per meter. These values are used to help a BMP reader choose a proper resolution when printing or displaying a BMP file.

ColorsUsed is the number of colors present in the palette. If this value is zero, and the value of *BitsPerPixel* is less than 16, then the number of entries is equal to the maximum size possible for the *colormap*. BMP files with a *BitsPerPixel* value of 16 or greater will not have a color palette. This value is calculated by using the value of the *BitsPerPixel* field:

$$\text{ColorsUsed} = 1 \ll \text{BitsPerPixel};$$

ColorsImportant is the number of significant colors in the palette, determined by their frequency of appearance in the bitmap data; the more frequent the occurrence of a color, the more important it is. This field is used to provide as accurate a display as possible when using graphics hardware supporting fewer colors than are defined in the image. The most important colors are always stored first in the palette; **ColorsImportant** is 0 if all of the colors in the palette are to be considered important.

Following the bitmap header is the color palette data. A color palette is always present in a BMP file if the bitmap contains 1-, 4-, or 8-bit data. 24-bit bitmap data never uses a color palette (nor does it ever need to). Each element of the palette is three bytes in length, and has the following structure:

```
typedef struct _Win3xPaletteElement
{
    BYTE Blue;
    BYTE Green;
    BYTE Red;
    BYTE Reserved;
} WIN3XPALETTEELEMENT
```

Blue, Green, and Red hold the color component values for a pixel; each is in range 0 to 255.

The size of the color palette is calculated from the value of the *BitsPerPixel*.

The last entity in the bitmap file format is the **Bitmap data**. It is a stream of bits that stores the pixel values appropriate to the corresponding image. For a more detailed survey of the Microsoft Windows Bitmap, we refer the reader to [ST, MS]

3.4.2 Color Palette

A palette, which is sometimes referred to as a *color map*, *index map*, *color table*, or *look-up table (LUT)*, is a one-dimensional array of color values. As the synonym *look-up table* suggests, it is the cornerstone of the method whereby colors can be referred to indirectly by specifying their positions in an array. Using this method, data in a file can be stored as a series of index values (usually small integer values) which can drastically reduce the size of the pixel data when only a small number of colors need to be represented. Bitmaps using this method of color representation are said to use *indirect*, or *pseudo-color* storage.

Four-bit pixel data, for instance, can be used to represent images consisting of 16 colors. These 16 colors are usually defined in a palette that is almost always included somewhere in the file. Each of the pixel values making up the pixel data is an index into this palette, and consists of one of the values 0 to 15. The job of the rendering application is to read and examine a pixel value from the palette, which it then uses to specify a colored pixel on the output device.

Figure 3.3 illustrates how a color palette may be used to specify a color.

Figure 3.3: Palette and Color Mapping

The palette is an array of colors defined as accurately as possible. In practice, each palette element is usually 24-bits (or three bytes) long, although to accommodate future expansion and machine dependencies, each element is sometimes stored using 32 bits, or four bytes. What this means is that palettes are three or four times as large as the maximum number of colors defined. For instance, a 4-bit color palette is 48 bytes or 64 bytes in length as:

- (i) if 3 bytes is used in each color definition, colors would require 48 bytes in length, and
- (ii) if 4 bytes is used in each color definition, 16 colors would require 64 bytes in length.

In a similar way, 8-bit pixel data may be used to represent images consisting of 256 colors. Each of the pixel values, in the range 0 to 255, is an index into a 256-color palette. In the case of the palette:

- (i) if 3 bytes is used in each color definition, 256 colors would require 768 bytes in length, and
- (ii) if 4 bytes is used in each color definition, 256 colors would require 1024 bytes in length.

Issues with Color Palette

We assume that the value (255,0,0) represents the color “red” in the color model used by our image format. Also we assume that our example palette defines 16 colors, arranged as an array of 16 elements as below:

(0, 0, 0)

(255,0,0)

...

(128,128,255).

Because (255,0,0) happens to be the second element, we can store the value ‘1’ (assuming that the array is zero-based), with the implied convention that the values are to be interpreted as index values into the array. Thus every time a specification for the color red occurs in the pixel data, we can store ‘1’ instead, and we can do likewise for other colors found in the image.

In larger, more complex image formats, indirect storage through the use of palettes saves space by reducing the amount of data stored in the file. Thus it is more than clear that the use of palettes is very advantageous. First of all, if we need to know how many actual colors are stored in an image, it is a simple task to read thorough the palette. Also, when we want to change the colors in an image, all red pixels to blue for instance, it is just a matter of traversing the palette as opposed to the image itself.

On the other hand, the use of palettes is not appropriate in every case. A palette itself uses a substantial amount of space. Therefore, if the amount of storage that the data and palette uses itself amounts to considerably more than what it would be without a palette, the use of the

palette is not advisable. Indeed, in practice, any image that does contain more than 256 colors does not utilize a palette.

3.5 Generic Image Format

Having said so much about different file formats, we shall now develop a very simple data structure with which we can handle even the most complex of the image formats such as TIFF. Our application interacts with this data structure for the purpose of interpretation and manipulation of the data. Using this structure, we shall present interfacing algorithms and specify details of their implementation for reading paletted Windows Bitmap files as a part of the this study. Additionally we shall see that it is just as easy to be able to write interfacing routines that can read in any file format and store it in our structure so that our color masking and image tracing routines work with minimal change. Figure 3.4 illustrates the relationship between the various file formats to our data structure and the application.

Figure 3.4: Interaction between different file formats with our application

Image Format Details

We now introduce the structure “GenericImage” which is described below and we shall explain all its fields in greater detail individually.

Structure GenericImage

```
{
    unsigned int    unType;
    unsigned int    unComprType;
    POINT           nWidth;
    POINT           nHeight;
    int             nPlanes;
    int             nBitsPerPlane;
    int             nBytesPerLine;
    PALETTE         palette;
    BYTE            *pData;
}
```

unType:

This field designates the type of file format in use. In general, images are classified as RGB, PALETTERED, CMYK, BLACK&WHITE, GRAYSCALE, LAB, etc., depending on how data is stored internally. While each pixel in an RGB image consists of 3 bytes - one each to contain information about Red, Green, and Blue colors, each pixel in a paletted image is made up of one pixel that stores an index into the color palette, and so on. For details on color models we refer the reader to [HR].

Windows Bitmap and TIFF both fall into the category of RGB type.

unComprType:

Compression is the process used to reduce the physical size of a block of information. By compressing graphics data, we are able to fit more information in a physical storage space. Because graphics images usually require a very large amount of storage space, compression is an

important consideration for graphics file formats. Almost every graphics file uses some compression strategy.

A list of common compression techniques currently in use includes:

- (i) Run-length encoding (RLE) – This is a very common compression algorithm used by various bitmap formats such as BMP, TIFF, and PCX. The actual technique itself is described in [AL].
- (ii) Lempel-Ziv-Welch (LZW) – This is a technique used by GIF and TIFF. Its properties can be found in [PD, ZL].
- (iii) CCITT encoding – A form of data compression detailed in [MV] which is used for facsimile transmission and standardized by the CCITT.
- (iv) Joint Photographic Experts Group (JPEG) – This is a toolkit of compression methods used particularly for continuous-tone image data and multimedia. Full description of JPEG can be found in [KL].

Indeed, the details of the algorithms and a host of others and their associated compression techniques are found in [KL].

nWidth:

This field stores the width of the image in pixels.

nHeight:

This field records the height of the image in pixels.

nPlanes:

This field specifies the number of color planes used to represent bitmap data.

nBitsPerPlane:

This field specifies the number of data bits that correspond to each plane.

nBytesPerLine:

This field stores the number of bytes that make up any one row of the image.

pData:

This variable is a pointer, in memory, to the stream of data representing the given image. Data could be organized in any one of the schemes mentioned earlier viz., contiguous, strips, tiles, etc. as discussed in chapter 2. In our application, we shall use a contiguous memory scheme for the ease of implementation.

Reading multiple image data formats

It is not distinctly obvious from the above presentation how a complex file format like TIFF (that consists of multiple images) can be processed with using our data structure. The solution to work with such images is to create multiple instances of the GenericImage structure, each instance holding information pertinent to one image. The advantage of such a scheme is that it not only keeps all images in one format, but also avoids possible confusion that arises out of complex offset techniques used in such data formats. In addition, as mentioned earlier, our application and data structure is not affected by the specific peculiarities of the other data formats. We present an algorithm that reads a TIFF file into our structure in the next section to show how simple it is to do this task.

3.6 Reading Paletted Images

In this section, we shall present an algorithm that provides the interface to our application by reading a paletted Windows Bitmap file and storing it in our structure. Though no assumption is made about any specific programming language, we have opted to use C++ for the

implementation. Also, we would like to emphasize that the correctness of the algorithm is quite straight forward. The algorithm merely makes a correspondence between the file format and the generic data structure which we have introduced.

ALGORITHM ReadPalettedImage: This algorithm reads a paletted bitmap image into an instance of our GenericImage structure. Input to this function is the image file name along with the complete path where it is located on the hard drive. If the image to be read is not of the bitmap type, this function does not do any reading of the image data. For simplicity, this algorithm is divided into different modules, each performing a specific task.

```
ReadPalettedImage( STRING FileName )
BEGIN
    GENERICIMAGE OurImage
    FILE pStartFile, pImageFile

    // STEP 1. Open the image file
    pImageFile := FileOpen( FileName )

    // STEP 2. Save the start of the file
    pFileStart := pImageFile

    // STEP 3. Read Image Header
    Success := CALL ReadHeader( pImageFile )

    IF( Success )
    BEGIN
        // STEP 4. Read Bitmap Info Header
        CALL ReadBitmapInfoHeader( OurImage, pImageFile )

        // STEP 5. Read Image Data
        CALL ReadImageData( pImageFile )
    END

    // STEP 6. Done, close the file
    FileClose(pStartFile);
END // ReadPalettedImage
```

PROCEDURE ReadHeader: This procedure reads the file header section of any bitmap image. The file header section contains the file specifier - bitmap file, TIFF file etc. This function returns TRUE if we read a bitmap file, FALSE value otherwise. If we are reading a bitmap file, it also augments the file pointer, *pImageFile*, to the next section of the bitmap file - Bitmap Info Header.

```
ReadHeader( FILE pImageFile )
BEGIN
    // STEP 1. Read file header
    BitmapFileHeader := Read data of the size of BitmapFileHeader

    // STEP 2. Verify that it is a bitmap
    IF( BitmapFileHeader.bfType .NE. "BM" )
        RETURN FALSE;

    // STEP 3. Offset the file pointer to point to info header
    pImageFile := pImageFile + SIZEOF(BITMAPFILEHEADER)

    RETURN TRUE
END // ReadHeader
```

PROCEDURE ReadBitmapInfoHeader: This procedure reads all the file attributes viz. Width, Height, Resolution, etc. into an instance of our structure which is passed in as a parameter. At the end of reading all the attributes, it increments the file pointer to point to next section in the bitmap file - which could either be the palette information, if there is a palette, or the data itself.

```
ReadBitmapInfoHeader( GENERICIMAGE OurImage, FILE pImageFile )
BEGIN
    // STEP 1. Read Bitmap info header size
    BitmapInfoHeaderSize := Read First 4 bytes pointed to by pImageFile

    // STEP 2. Read bitmap info header data
    BitmapInfoHeader := Read data of the size of BitmapInfoHeaderSize

    // STEP 3. Fill-in our structure
    OurImage.SetWidth( BitmapInfoHeader.nWidth )
    OurImage.SetHeight( BitmapInfoHeader.nHeight )
    OurImage.SetNumberOfPlanes( BitmapInfoHeader.Planes )
    OurImage.SetNumberOfBytesPerLine(
        BitmapInfoHeader.nWidth * BitmapInfoHeader.bitsPerPixel / 8 )
    OurImage.SetXResolution( BitmapInfoHeader.HorzResolution )
    OurImage.SetYResolution( BitmapInfoHeader.VertResolution )

    // STEP 4. Offset the file pointer to point color palette
    pImageFile := pImageFile + BitmapInfoHeaderSize

    // STEP 5. Read Color Palette
    IF( BitmapInfoHeader.ColorsUsed .GT. 0 )
        CALL ReadColorPalette(
            pImageFile, OurImage, BitmapInfoHeaderSize, BitmapInfoHeader.ColorsUsed )
END // ReadBitmapInfoHeader
```

PROCEDURE ReadColorPalette: This procedure reads the palette information from the file to OurImage, an instance of our structure that is being passed in. The color palette is an array of RGB triplets, the size being specified by the *NumberOfColorsUsed* parameter. As with the other procedures that read in different sections of bitmap file, this function too adjusts the offset of the file pointer to the next section.

```
ReadColorPalette(  
    GENERICIMAGE OurImage,  
    FILE pImageFile,  
    INTEGER BitmapInfoHeaderSize,  
    INTEGER NumberOfColorsUsed )  
BEGIN  
    // STEP 1. Compute the size of color palette  
    ColorPaletteSize := NumberOfColorsUsed * SIZEOF(RGBTRIPLE)  
  
    // STEP 2. Read bitmap info header data  
    ColorPalette := Read data of the size of ColorPaletteSize  
  
    // STEP 3. Fill-in our structure  
    OurImage.SetPalette( ColorPalette )  
  
    // STEP 4. Offset the file pointer to point to data  
    pImageFile := pImageFile + ColorPaletteSize  
END // ReadColorPalette
```

PROCEDURE ReadImageData: This procedure reads bitmap image data into the supplied structure instance. At the termination of this procedure, the caller function will be ready to close the file and return the success value to the caller.

```
ReadImageData( GENERICIMAGE OurImage, FILE pImageFile )  
BEGIN  
    OurImage.SetData( pImageFile )  
END // ReadImageData
```

One of the primary reason why we have chosen to use the paletted bitmap files for the purpose of implementing our system is because of the simplicity of reading-from and writing-to data structures which use this format. As we can see from the algorithm that has just been presented, the strategy involves a straightforward mapping of different fields of the bitmap file into the appropriate fields of our structure. However, reading files of another format is not too

complex either. It may involve performing seek operations to the specific fields of the file back and forth before mapping onto our structure. Thus if we consider a TIFF file in which each IFD is followed by the respective bitmap data (See Figure 3.1), it is very interesting to note how similar the interface functions that read in the file information into our structure are, even though the bitmap file format varies a lot from its TIFF counterpart.

ALGORITHM ReadTIFFImage: This algorithm reads a TIFF file, whose format was presented in the earlier section of this chapter, into our structure. Since a TIFF file may contain multiple images, it is necessary to create multiple instances of our structure, each for an image. *ReadIFD* in STEP 6 does the job that its counterpart *ReadBitmapInfoHeader* does in case of a paletted bitmap image.

```
ReadTIFFImage( STRING FileName )
BEGIN
    GENERICIMAGE [] OurImage
    FILE pStartFile, pImageFile
    INTEGER ImageCount

    // STEP 1. Open the image file
    pImageFile := FileOpen( FileName )

    // STEP 2. Save the start of the file
    pFileStart := pImageFile

    // STEP 3. Read Image Header
    Success := CALL ReadHeader( pImageFile )

    // STEP 4. Initialize
    ImageCount := 0

    IF( Success )
        BEGIN
            WHILE( .NOT. EndOfFile )
                BEGIN
                    // STEP 5. Create new instance of OURIMAGE
                    OurImage[ImageCount] := create new instance of OURIMAGE

                    // STEP 6. Read IFD
                    CALL ReadIFD( OurImage[ImageCount], pImageFile )

                    // STEP 7. Read Image Data
                    CALL ReadImageData( pImageFile )

                    // Keep image count up to date
                    ImageCount := ImageCount + 1
                END
            END

            // STEP 8. Done, close the file
            FileClose(pStartFile);
        END // ReadTIFFImage
```


3.7 Results

The results obtained by using the interface algorithms presented in this chapter are used in conjunction with the algorithms which we are going to discuss in the succeeding chapters. Consequently, we do not see a need to present any image representation results here. However, the image file data that we read in here can be displayed independently, and can indeed be viewed in our interactive application environment.

3.8 Conclusion

In this chapter, we have described various graphics file formats in order to understand the distinction between the different formats, and have therefore encountered the necessity to create our own data structure. The intention, of course, is to eliminate the core application algorithms from having to deal with multiple file types.

At this juncture, it may be advantageous to mention that our data structure is not complete because it does not contain placeholders for all the information that is contained in each of the existing image type. This is because, at this point of time, our application is designed to work only with Windows bitmap images. Therefore, we have tried to incorporate only those data attributes that are necessary to work with this format. However, where ever needed, we can easily add additional attributes to the current structure. Details on how to add extra data attributes to GenericImage is presented in Chapter 6 which highlights the design and implementation details of the thesis.

Using the GenericImage structure, we are now ready to process the image data and build our application. In the chapters that follow, we use the information contained in this structure to

color separate the image data to form regions, perform edge detection on the resulting regions, and finally, to trace them for contour generation.

CHAPTER 4

COLOR SEPARATION

4.1 Introduction

In Chapters 2 and 3, we considered how we could read the image data from a bitmap file into a data structure which we introduced called *GenericImage*. It is now time for us to interpret this image data in order to generate the contour of the shapes embedded in it. Therefore, it is very important for us to understand how the basic shapes of objects are determined in order to proceed with the question of how we can obtain the vector associated with the boundary of such objects.

There are various methods of establishing the basic shapes embedded in an image. The specific approach used often depends on the application context as well as the underlying image data. In an image known to contain closed shapes, a simple color separation technique is enough to extract the shape. In images that contain shapes that are not closed, it is more difficult to establish the identity. In such cases, a variety of different techniques are used to generate the shape.

Although a host of such techniques are available [HS], in the interest of brevity, in Section 4.2, we discuss two different ways of finding the basic shapes - edge detection and segmentation. Following this, in Section 4.3, we shall present two well accepted methods of segmentation.

In Section 4.4, we will propose our strategy of using color separation. We present the algorithm in Section 4.5, and the results of applying this algorithm to our exhibits in Section 4.6. Section 7 contains concluding notes for this chapter.

4.2 Finding Basic Shapes

There are two standard techniques which are predominantly used in order to find the shape of an object - segmentation and edge detection. While segmentation is concerned with splitting an image up into segments or regions that each hold some property distinct from their neighbor, edge detection is seen as the mirror of segmentation in that the completion of an edge is equivalent to breaking one region into two. Although ideally, edge and region operations should give the same result, in practice they rarely correspond. In this section, we examine these two techniques briefly. For a more elaborate discussion of these topics, we refer the reader to [HM].

4.2.1 Edge Detection

Edge detection involves the detection of the edges of closed shapes in a given image. In order to do this, we start with a pixel on the boundary of each of the object, and follow through it all the way back to the start in order to establish its shape. However, as soon as we use this technique to determine the shape of objects in an image with vivid data, we encounter a lot of issues. If this were not the case, edge detection, followed by tracing an edge detected shape would be more than enough for a user to realize the goal of our solution - to find the contour of all closed shapes in an image. Both edge detection and tracing are covered in detail in the next chapter.

The issues that we face with edge detection are:

- How should the edge detection technique establish the first point on the boundary of any object in an image that contains many objects?

- How should the technique proceed if different objects overlap with each other?
- What is the guarantee that we will trace our path back to the starting pixel once we have begun an edge tracing?

The existence of different colors, contrast, and hue in an image makes it difficult to answer all of these questions. Also, establishing the starting point in an image with vivid data is no trivial task. If the threshold value which is used to decide on the edges is not selected properly, this method could result in an edge traversal that does not coincide with the starting point, and consequently, time-out has often to be signaled to abort the whole operation.

4.2.2 Segmentation

Segmentation involves identifying different regions in an image by distinguishing them based on any characteristic feature. This feature could be one of color, contrast, hue, saturation, and so on. This type of division of the image data into segments based upon a distinct feature is an essential part of scene analysis – in answering questions such as where and how large the object is, where the background is, how many objects there are, and if there are any objects that we are trying to look for, etc.

Segmentation is a basic requirement for the identification and classification of objects in a scene. It focuses subsequent algorithms on the shape and texture of a homogeneous area. It is therefore vital to use a good characteristic feature which can serve to separate the regions, so that the algorithms that depend on the output of segmentation also yield good results. The exact feature that is used for this purpose usually depends on the application domain itself. For additional information on segmentation, please refer to [FM, BF, KK].

There are two primary ways of separating an image into multiple regions. We deal with these two in the next section.

4.3 Region Operations

Discovering regions can be a very simple exercise. However, more often than not, the application domain requires regions that cover a substantial area of the scene rather than a small group of isolated pixels. In the following paragraphs, we shall present two standard methods that are used to regroup an image as a set of regions:

4.3.1 Method 1

In this method, the regions are identified as containing pixels of the same grey level. The boundaries of the regions (contours) are at the “crack” between these pixels rather than at the pixel positions themselves.

A region detection strategy (such as the one described above) may yield far too many regions to be useful, unless the number of grey levels is relatively small. Consequently, a simple approach is to group pixels into ranges of values close to each other, a technique known as quantizing or bunching, for example in intervals as, 0-24, 25-73, ... 161-255. The ranges can be estimated by considering bunches or values on the grey-scale histogram. In our implementation, we provide the users with an easy way of setting the threshold value that is used to set the grey levels. For more details on this method, we ask the reader to kindly refer to [KR, WNR].

4.3.2 Method 2

The technique used in this method is to do the rough grey-level split and then to perform some technique on the cracks between the regions – not to enhance the edges, but to identify when whole regions are worth combining – thus reducing the number of regions from the method mentioned above (method 1). In essence, the goal of this method is to reduce the number of regions, combine fragmented regions and finally to determine which regions are really part of the same area. We shall study this scheme in greater detail so as to get an insight into how this method works.

Let s be a crack difference, i.e. the absolute difference in grey levels between two adjacent (above, below, left, or right) pixels.

Then, given a threshold value T , we can identify, for each crack

$$w = 1 \text{ if } s < T$$

$$w = 0 \text{ otherwise.}$$

Thus, w is set to 1 if the crack is below the threshold (suggesting that the regions are likely to be the same), or 0 if it is above the threshold. When w is 1, we say that the cracks are significant because we may deduce that the regions with this crack difference may be joined.

We now consider the full length of the boundary of each of the regions that meet at the crack. If these are b_1 and b_2 , we calculate the quantity computed by adding the w 's along the length of the crack between the regions and dividing the sum by the minimum of b_1 and b_2 . Let this quantity be δ , where,

$$\delta = \frac{\sum w}{\text{Min}(b_1, b_2)}$$

If δ is greater than a further threshold, we deduce that the two regions should be joined.

An example should clarify the method further.

Example 4.1:

Consider an image with the following data in which each entry can vary between 0-255,

1	1	1	2	2	2	2	4	4
1	1	1	1	2	2	2	4	4
1	1	1	1	1	1	2	2	2
1	1	1	1	1	1	1	1	1

We define A to be the region consisting of all grey pixels with value '1', B to be the region that contains all grey pixels with value '2', and C to be the region that has all grey pixels with value '4'.

If we make $T = 3$, then all the cracks are significant because for each crack, we obtain $w = 1$, as it can be seen below:

1	1	1	2	2	2	2	4	4
1	1	1	1	2	2	2	4	4
1	1	1	1	1	1	2	2	2
1	1	1	1	1	1	1	1	1

Now, let us compute the boundaries of each of these regions.

Let b_A = Boundary of A = number of all pixels in the region (with value '1') that adjoin any other region. It is easy to see that b_A is 17.

Let b_B = Boundary of B = number of all pixels in the region (with value '2') that adjoin any other region. In this case, $b_B = 9$.

Let b_C = Boundary of C = number of all pixels in the region (with value '4') that adjoin any other region. Here $b_C = 4$.

Next, let us calculate the sum of the w 's between these three regions.

Let w_{AB} = sum of w 's that are along the length of crack between the regions A and B and

Let w_{BC} = sum of w 's that are along the length of crack between the regions B and C. In this case, w_{AB} and w_{BC} have values 9 and 4 respectively.

The data we have obtained can now be tabulated as:

<u>Region</u>	<u>Boundary</u>	<u>Sum of Ws</u>		
		<u>With A</u>	<u>With B</u>	<u>With C</u>
A	$b_A = 17$	-	9	0
B	$b_B = 10$	9	-	4
C	$b_C = 4$	0	4	-

Table 4.1: Length of crack between regions

Now, $\delta = \sum w / \min(b_1, b_2)$ for A, B, and C is as follows:

<u>Region</u>	<u>A</u>	<u>B</u>	<u>C</u>
A	-	0.9	0
B	0.9	-	1.0
C	0	1.0	-

Table 4.2: Full length of the boundary of each of the regions

If we take the threshold for combining regions as 0.95, then B will be combined with C.

If the combining threshold is set lower, A will also be combined with B and C causing the whole image to be just one region. It is interesting to note that although the differences between the pixels are bigger, the algorithm combines the small region with the larger region, rather than the two larger regions.

While measuring both boundaries (as in the technique above) is better in some cases than dividing by the boundary length between the two regions (as it takes into account the size of the regions involved), it is not in some other cases. For instance, if the objects are established to be bound within a region within a specific color tolerance, as is the case in our present study, one could use the simple algorithm to get different regions. However, if the objects are scattered across an image with different grey separations, we would want to use the latter approach. In general, the particular technique used in an application generally depends on the specific context of operation.

For more information on this method of region separation, please refer to [AL].

4.4 Color Separation Technique

Since edge detection does not work quite accurately on images with multiple regions, each with different features, it is a good idea to apply edge detection technique only on images that have already been marked with clear regions. In this way, we can ensure that edge detection does not encounter any problem that may otherwise arise due to the complexity of the data present in an image. Therefore, it is better to use any one of the region separation techniques to extract one or more images from the given image, where each one is composed of different regions present in the original image, and then to finally use the edge detection and tracing algorithms on these color separated images to realize the contour. In order to do this, we shall, in our work, use color changes in an image as a characteristic feature to identify regions. Specifically, we shall use the *quantizing* method of separating an image into multiple regions based upon the grey scale value changes.

Our decision is also influenced by the fact that our goal is to be able to generate the contour of closed shapes in an image, and in general, contour tracing problems are more meaningful only in images that contain closed shapes. For applications that intend to work on images with object data scattered all across, this strategy may not provide satisfactory results, but it is questionable whether any simple strategy will work in such a setting.

4.5 Algorithm Color Separation

Having presented an understanding of what the different methods of region separations do and the goal of our solution, we are now ready to present the algorithm that works for our present problem. Indeed, it processes image data that we have read into the data structure introduced in chapter 3, and as a result, creates one or more images, each containing distinct regions present in the original image.

We have presented two predominant algorithms here - one to color separate the image and the other to convert any RGB-image to Black and White image. The latter is used in the tracing algorithm which is presented in the next Chapter. Procedures *GetColorValues* and *ComputeMinMaxRGBValues* serve as helpers for the *ColorSeparate* algorithm.

ALGORITHM ColorSeparate: This algorithm walks through the image data of the *SourceImage*, identifies all pixels that falls within the specified *Tolerance* value of the user selected color, and copies them to the *DestinationImage*. Any pixel in the *SourceImage* that does not lie within the color tolerance limits specified by the user would be substituted with a white pixel. The user specified color value, that identifies the color to separate, is obtained in the subroutine *GetColorValues* and the min/max tolerance values for Red -, Blue -, and Green - components is determined in the procedure *ComputeMinMaxRGBValues*.

```

ColorSeparate(
    GENERICIMAGE SourceImage,
    GENERICIMAGE DestinationImage,
    INTEGER Tolerance )
BEGIN
    INTEGER Row, Column
    OURPOINT CurrentPixel
    INTEGER RedValue, GreenValue, BlueValue
    INTEGER RedSeed, GreenSeed, BlueSeed
    INTEGER MinRed, MaxRed, MinGreen, MaxGreen, MinBlue, MaxBlue

    // STEP 1. Initialize variables
    Row := 0
    Column := 0

    WHILE( Row .LT. SourceImage.GetHeight() )
    BEGIN
        WHILE( Column .LT. SourceImage.GetWidth() )
        BEGIN
            // STEP 2. Get the user specified color value for separation
            CALL GetColorValues( RedSeed, GreenSeed, BlueSeed)

            // STEP 3. Get the min/max on user specified color values
            CALL ComputeMinMaxRGBValues( Tolerance,
                                         MinRed, MaxRed,
                                         MinGreen, MaxGreen,
                                         MinBlue, MaxBlue,
                                         RedSeed,
                                         GreenSeed,
                                         BlueSeed )

            // STEP 4. Get the current pixel color components
            ValueRed := SourceImage.GetRedAt( Row, Column )
            ValueGreen := SourceImage.GetGreenAt( Row, Column )
            ValueBlue := SourceImage.GetBlueAt( Row, Column )

            // STEP 5. Determine the color value at (Row,Column) on the
            // destination image
            IF( .NOT.((RedValue .LE. MaxRed) .AND. (RedValue .GE. MinRed) .AND.

```

```
(GreenValue .LE. MaxGreen) .AND. (GreenValue .GE. MinGreen) .AND.  
(BlueValue .LE. MaxBlue) .AND. (BlueValue .GE. MinBlue)) )  
BEGIN  
    DestinationImage.SetRedAt( Row, Column, 255 )  
    DestinationImage.SetGreenAt( Row, Column, 255 )  
    DestinationImage.SetBlueAt( Row, Column, 255 )  
END  
ELSE  
BEGIN  
    DestinationImage.SetRedAt( Row, Column, RedValue )  
    DestinationImage.SetGreenAt( Row, Column, GreenValue )  
    DestinationImage.SetBlueAt( Row, Column, BlueValue )  
END  
END  
END  
STOP  
END // ColorSeparate
```

PROCEDURE GetColorValues: This procedure gets the user selected color which should be separated in an image. This algorithm corresponds with the user-interface object in order to get the color. Algorithm *ColorSeparate* uses this seed value, with appropriate tolerance added, to color separate the image.

```
GetColorValues( INTEGER RedSeed, INTEGER GreenSeed, INTEGER BlueSeed )  
BEGIN  
    RedSeed := Red Component of the user selected Color  
    GreenSeed := Green Component of the user selected Color  
    BlueSeed := Blue Component of the user selected Color  
END // GetColorValues
```

PROCEDURE ComputeMinMaxRGBValues: This procedure adds the necessary tolerance values to the Red -, Blue -, and Green - components of the specified seed RGB color. The *Tolerance* value is supplied by the user by interacting with the application user-interface.

```
ComputeMinMaxRGBValues(  
    INTEGER Tolerance,  
    INTEGER MinRed,  
    INTEGER MaxRed,  
    INTEGER MinGreen,  
    INTEGER MaxGreen,  
    INTEGER MinBlue,  
    INTEGER MaxBlue,  
    INTEGER RedSeed,  
    INTEGER GreenSeed,  
    INTEGER BlueSeed )  
BEGIN  
    MinRed := RedSeed - Tolerance .LT. 0 ? 0 : RedSeed - Tolerance  
    MaxRed := RedSeed + Tolerance .GT. 255 ? 255 : RedSeed + Tolerance  
  
    MinGreen := GreenSeed - Tolerance .LT. 0 ? 0 : GreenSeed - Tolerance  
    MaxGreen := GreenSeed + Tolerance .GT. 255 ? 255 : GreenSeed + Tolerance  
  
    MinBlue := BlueSeed - Tolerance .LT. 0 ? 0 : BlueSeed - Tolerance  
    MaxBlue := BlueSeed + Tolerance .GT. 255 ? 255 : BlueSeed + Tolerance  
END // ComputeMinMaxRGBValues
```

ALGORITHM ConvertToBlackAndWhite: This algorithm creates a black and white version of the source image. It creates a white pixel in destination image for all source pixels that are in the tolerance limits specified by *Tolerance* from the given seed color, an RGB triplet, *RedSeed*, *GreenSeed*, and *BlueSeed*. Though we intended to use this algorithm in conjunction with the tracing technique that we present in the next chapter, the code presented here is generic to convert any RGB image to a black and white format.

```

ConvertToBlackAndWhite(
    GENERICIMAGE SourceImage,
    GENERICIMAGE DestinationImage,
    INTEGER Tolerance,
    INTEGER RedSeed,
    INTEGER GreenSeed,
    INTEGER BlueSeed )
BEGIN
    INTEGER MinRed
    INTEGER MaxRed
    INTEGER MinGreen
    INTEGER MaxGreen
    INTEGER MinBlue
    INTEGER MaxBlue

    WHILE( Row .LT. SourceImage.GetHeight() )
    BEGIN
        WHILE( Column .LT. SourceImage.GetWidth() )
        BEGIN
            // STEP 1. Get the min/max on user specified color values
            CALL ComputeMinMaxRGBValues( Tolerance,
                                         MinRed, MaxRed,
                                         MinGreen, MaxGreen,
                                         MinBlue, MaxBlue,
                                         RedSeed,
                                         GreenSeed,
                                         BlueSeed )

            // STEP 2. Get the current pixel color components
            ValueRed := SourceImage.GetRedAt( Row, Column )
            ValueGreen := SourceImage.GetGreenAt( Row, Column )
            ValueBlue := SourceImage.GetBlueAt( Row, Column )

            // STEP 3. Determine the color value at (Row,Column) on the
            // destination image
            IF( .NOT.( (RedValue .LE. MaxRed) .AND. (RedValue .GE. MinRed) .AND.
                (GreenValue .LE. MaxGreen) .AND. (GreenValue .GE. MinGreen) .AND.
                (BlueValue .LE. MaxBlue) .AND. (BlueValue .GE. MinBlue)) )
            BEGIN
                DestinationImage.SetRedAt( Row, Column, BLACK )
            END
        END
    END

```

```
        DestinationImage.SetGreenAt( Row, Column, BLACK )
        DestinationImage.SetBlueAt( Row, Column, BLACK )
    END
    ELSE
    BEGIN
        DestinationImage.SetRedAt( Row, Column, WHITE )
        DestinationImage.SetGreenAt( Row, Column, WHITE )
        DestinationImage.SetBlueAt( Row, Column, WHITE )
    END
    END
END
END // ConvertToBlackAndWhite
```

4.6 Results

To demonstrate the power of our strategy, we display the effect of invoking our algorithms on Exhibit1 and Exhibit2 that we presented in Chapter 2. The results which we see demonstrate the effect of algorithm *ColorSeparate*. Figures 4.1 - 4.3 and 4.4 - 4.6 show the boundaries obtained when threshold values of 20, 40, and 80 are used on selected colors in these exhibits to color separate them into different regions. It can be easily seen in the following figures that an increase in the threshold value results in the revelation of greater image detail but renders the boundary to be less skeletal.

Figure 4.1: Color Separation of components of Exhibit 1 at threshold = 20

Figure 4.2: Color Separation of components of Exhibit 1 at threshold = 40

Figure 4.3: Color Separation of components of Exhibit 1 at threshold = 80

Figure 4.4: Color Separation of Exhibit 2 at threshold = 20

Figure 4.5: Color Separation of Exhibit 2 at threshold = 40

Figure 4.6: Color Separation of Exhibit 2 at threshold = 80

4.7 Conclusion

Edge detection and segmentation play a key role in trying to figure out what to do with the data contained in an image. Often, the application of edge detection algorithms or segmentation depends on what the subsequent processing is. If the sole purpose of an application is just to work with cracks in order to fix up noisy “dust and scratches”, or to render the black and white edge detected image on the screen to identify the crisp edges, then it is not necessary to separate the image data into multiple regions at all. However, in many applications, we would want to figure out the shape embedded in an image in order to proceed with other interesting tasks on the image data such as scene analysis, pattern recognition, etc.. In all of these cases, color separation and contour generation is very essential while edge detection acts as an aid in achieving the desired result. Indeed, we shall use an edge detection algorithm in the next chapter to help us generate the contour of the basic shapes that we have obtained by using the color separation techniques discussed in this chapter.

Details on how we let the user determine the threshold values is discussed in Chapter 6 which deals with the design and implementation details. But before that, we want to continue to obtain our final goal of the solution - that of tracing the image data for contour extraction.

CHAPTER 5

EDGE DETECTION AND TRACING

5.1 Introduction

In the previous chapter, we studied how to separate an image based on colors to obtain multiple images. In this chapter, we focus on another important aspect of our work - generation of contour for each of these color separated image. The contour for an image is obtained in the form of a vector, that can be used later for further synthesizing the image data.

There are many ways of generating the contour of any given image. We can first detect all the edges of an image by using one of the many techniques available [HS], and then follow through the resulting black and white image data to realize the vector associated with the contour. Alternatively, we can also establish the vector by following the edges in just one pass by adding intelligence to the edge detection algorithms that work on standard templates. In any case, it is very essential to understand the techniques behind using the edge detection templates that play key role in the detection and tracing of edges.

In Section 5.2, we discuss the concepts of edge detection and *gradient magnitude* observing that latter is crucial in the notion of edge detection.

We start with the simple template for edge detection in Section 5.3 and proceed to the sophisticated Sobel's template. We consider Sobel's template in detail in order to help simplify the understanding of the tracing algorithm that we present later. We shall also list some of the shortcomings of templates such as Sobel's and present a brief introduction to second-order templates.

While we could use Sobel's template to generate the black and white images and use them to follow the edges later, we have chosen an alternative technique to trace the image.

Indeed, we intend to do just one pass on the image data using Sobel's template enhanced by incorporating extra intelligence. Section 5.4 describes the algorithm we have developed that does the job of tracing.

Figures of the sample images we considered in Chapter 2 are depicted to highlight the results. Indeed, all the important techniques that we discuss are supplemented with their corresponding output images.

5.2 Edge Detection

The primary goal of an edge detection algorithm is to identify all the lines that outline the objects in an image. A basic edge detector gives a measure of how likely each pixel is part of a 'real' edge. This measure is termed the *gradient magnitude*. It is often necessary to decide what threshold or gradient magnitude values constitute a real edge and what do not. Depending on the application, threshold might be at a high value if over identification is preferred, or at a low value if only valid edges are to be identified.

In the following section, we shall consider some of the existing templates that are used to detect edges.

5.3 Templates for edge detection

In an image that contains conspicuous edges enclosing an object, we can use a trivial algorithm that walks through the pixels in a direction of changing gradient magnitude at any given pixel, such an algorithm need not consider the neighboring pixels, in order to detect the edge. However, most of the images contain filled regions which necessitates the need to take the

adjoining pixels also into consideration while investigating the status of any given pixel. Worst of all, in some cases, it may not be possible to detect the edges at all when the image contains a spectrum of color changes.

In the following subsections, we discuss a trivial edge detection algorithm as well as Sobel's template that takes adjacent pixels into account. Advanced templates that deals with images whose data represents gradient changes is beyond the scope of this study. We rather present a brief introduction to the Laplacian template, and refer the reader to other references for further details [HS]. But before that, for the purpose of understanding, we shall briefly explain the convolution process which is extensively used with the edge detection templates.

5.3.1 Convolution Principle

To convolve an image, the filter array, $f[x]$, is aligned with an equal sized subset of the image. Every element in the convolution kernel array corresponds to a pixel in the subimage. At each step of the convolution, the color values of each pixel corresponding to a kernel array element, $g[i]$, are read, and then scaled by their corresponding kernel element. The resulting values are all summed together into a single value and stored in the array $h[x]$. Thus, every element of the kernel, and every pixel under the kernel, contribute values that are combined into a single convolved pixel color. One of the kernel array elements corresponds to the location where this output value is written so as to update the output image.

After each convolution step, the convolution kernel filter position is shifted by one unit, thus covering a slightly different set of pixels in the input image. At this juncture, a new convolution step is performed. The cycle continues, convolving consecutive pixels in a scanning pattern, until the entire image has been convolved.

Generally, convolution is achieved with separate input and output images, and so the input image is of a read-only format, and the outputs of the individual convolution steps do not affect each other.

A linear one-dimensional convolving function is given by:

$$h[x] := \sum_{i=0}^{\text{width} - 1} f[x + i] * g[i]$$

For a two-dimensional image, the convolving function is given as:

$$h[x][y] := \sum_{j=0}^{\text{height} - 1} \sum_{i=0}^{\text{width} - 1} f[x + i][y + j] * g[i][j]$$

The algorithm *Convolve* that we have presented in section 5.3.4 uses this principle to compute $h[i][j]$ at a given i and j for any kernel.

5.3.2 Simple Edge Detection

If we are looking for any horizontal edges, it would seem sensible to calculate the difference between one pixel value and the next pixel value, either up or down from the first one. This quantity is called the crack difference. Thus, we are processing the point with an origin at the top-left corner of the image.

Therefore,

$$Y_difference(x,y) = \text{value}(x, y) - \text{value}(x, y + 1)$$

In effect, this is equivalent to convolving the image with a 2 x 1 template

$$\begin{matrix} -1 \\ 1 \end{matrix}$$

Similarly,

$$X_difference(x,y) = value(x, y) - value(x - 1, y)$$

uses the template

$$\begin{matrix} -1 & 1 \end{matrix}$$

Occasionally it is useful to plot both $X_difference$ and $Y_difference$, combining them to create the single measurement of the gradient magnitude which represents the “strength” of the edge.

Combining them by simply adding them could mean two edges canceling each other out (one positive and one negative). Consequently, it is better to sum the absolute values (ignoring the sign) or sum their squares, and then, possibly, take the positive root of the result.

Alternatively, is also useful to divide the $Y_difference$ by the $X_difference$ and to thus identify a gradient direction which represents the angle of the edge between the regions.

$$gradient_direction(x, y) = \tan^{-1} \{ Y_difference(x, y) / X_difference(x, y) \}$$

Consider the example below. It can be seen that the gradient direction is at right-angles to the edge itself and points towards (and away from) the center of the areas whose edges have been detected. This can be used to detect the validity of an edge by looking at the continuity of the gradient, as well as identifying the image center which is useful for matching purposes. For more information about how to identify the image center using Hough transformation, we refer the reader to [ZH].

Image					X difference					Y difference					Gradient Direction				
1	1	1	1	1	0	0	0	0	*	0	0	0	0	0	*	*	*	*	*
1	1	1	1	1	0	0	0	0	*	0	0	0	1	1	*	*	*	↕	↕
1	1	1	0	0	0	0	-1	0	*	0	0	1	0	0	*	*	↗	*	*
1	1	0	0	0	0	-1	0	0	*	0	0	0	0	0	*	↔	*	*	*
1	1	0	0	0	0	-1	0	0	*	*	*	*	*	*	*	↔	*	*	*

Table 5.1: Image data and the associated gradient direction

Clearly, this small template does not take into account any of the surrounding pixels. In real images, the lines are rarely so well defined as in the above figure. More often, the change between regions is gradual and noisy.

The following image represents a typical real edge. It is apparent that a larger template is needed to average out the gradients over a number of pixels, rather than looking at two adjacent pixels only.

0	0	0	0	0	0	2	0	3	3
0	0	0	1	0	0	0	2	4	2
0	0	2	0	3	4	3	3	2	3
0	0	1	3	3	4	3	3	3	3
0	1	0	4	3	3	2	4	3	2
0	0	1	2	3	3	4	4	4	3

5.3.3 Sobel Edge Detection

The Sobel filter consists of two kernels which detect horizontal and vertical changes in an image. If both are applied to an image, the results can be used to compute the magnitude and direction of the edges in the image. If the application of the Sobel kernels results in two images

which are stored in the arrays $Gh[0..(height-1)][0..(width-1)]$ and $Gv[0..(height-1)][0..(width-1)]$, the magnitude, M_{sobel} , of the edge passing through the pixel x, y is given by:

$$M_{sobel}[x][y] = \sqrt{Gh[x][y]^2 + Gv[x][y]^2} = |Gh[x][y]| + |Gv[x][y]|$$

The direction, ϕ_{sobel} , can also be derived from Gh and Gv as:

$$\phi_{sobel}[x][y] = \tan^{-1}(Gv[x][y] / Gh[x][y])$$

The 3x3 Sobel kernels (or templates) are:

Horizontal

-1	0	1
-2	0	2
-1	0	1

and

Vertical

1	2	1
0	0	0
-1	-2	-1

For more information about the associated properties of Sobel's edge detection templates, we refer the reader to [IS].

To see how these templates work, let us take the following example:

Consider an image whose data is:

Image

0	0	0	0	0	0	2	0	3	3
0	0	0	1	0	0	0	2	4	2
0	0	2	0	2	4	3	3	2	3
0	0	1	3	3	4	3	3	3	3
0	1	0	4	3	3	2	4	3	2
0	0	1	2	3	3	4	4	4	3

Then, $\text{abs A} + \text{abs B}$ would be:

$\text{abs A} + \text{abs B}$

4	6	4	10	14	12	14	4
6	8	10	20	16	12	6	0
4	10	14	10	2	4	2	4
1	12	12	2	2	4	6	8

and the resulting image at threshold set to 12 would be:

Threshold at 12

0	0	0	0	1	1	1	0
0	0	0	1	1	1	0	0
0	0	1	0	0	0	0	0
0	1	1	0	0	0	0	0

Any change from 0 to 1 or vice-versa is detected as an edge. This is exactly the basis for the algorithm formally presented below.

5.3.4 The Edge Detection Algorithm

ALGORITHM EdgeDetect: This algorithm walks through the image data of the *SourceImage*, computes the Sobel's gradient magnitude at each pixel using the convolution matrix, and copies the resulting data to the *DestinationImage*. The Sobel's gradient magnitude is computed in the procedure *GetSobelGradientMagnitudeAt*. The value of *Threshold* is used to determine what values of gradient magnitudes constitute a real edge.

EdgeDetect(GENERICIMAGE SourceImage, GENERICIMAGE DestinationImage, INTEGER Threshold)

BEGIN

 INTEGER Row

 INTEGER Column

 INTEGER Value

 OURPOINT CurrentPixel

 // STEP 1. Initialize variables

 Row := 0

 Column := 0

 WHILE(Row .LT. SourceImage.GetHeight())

 BEGIN

 WHILE(Column .LT. SourceImage.GetWidth())

 BEGIN

 // STEP 2. Get Pixel value at the current Row/Column

 CurrentPixel := SourceImage.GetPixelAt(Row, Column)

 // STEP 3. Compute the Sobel's Gradient at Row/Column

 Value := CALL GetSobelGradientMagnitudeAt(Row, Column)

 // STEP 4. Determine if it should form an edge

 // in the destination image

 IF(Value .GE. Threshold)

 DestinationImage.SetPixelAt(Row, Column) := 1

 ELSE

 DestinationImage.SetPixelAt(Row, Column) := 0

 END

 END

 END

 STOP

END // EdgeDetect

PROCEDURE GetSobelGradientMagnitudeAt: This procedure computes the Sobel's gradient magnitude of a pixel specified by the row, *Row* and column, *Column* of the given source, *SourceImage*. We will first determine the horizontal and vertical Sobel values using the algorithm *Convolve* and then calculate the gradient magnitude.

```
GetSobelGradientMagnitudeAt( GENERICIMAGE SourceImage, INTEGER Row, INTEGER
Column )
BEGIN
    INTEGER [8] SobelHorzTemplate {-1, 0, 1, -2, 0, 2, -1, 0, 1}
    INTEGER [8] SobelVertTemplate {-1, 2, 1, 0, 0, 0, -1, -2, -1}

    INTEGER HorizontalValue
    INTEGER VerticalValue
    INTEGER GradientMagnitude

    // STEP 1. Convolve using horizontal Sobel template
    HorizontalValue := Convolve(SobelHorzTemplate, SourceImage, Row, Column )

    // STEP 1. Convolve using horizontal Sobel template
    VerticalValue := Convolve(SobelVertTemplate, SourceImage, Row, Column )

    // STEP 3. Compute the Sobel's Gradient Magnitude
    GradientMagnitude := SQRT(HorizontalValue2 + VerticalValue2 )

    RETURN(GradientMagnitude )
END // GetSobelGradientMagnitudeAt
```

PROCEDURE Convolve: This procedure convolves an image pixel specified by the row, *Row* and column, *Column* of the given source, *SourceImage*. *Template* contains the kernel matrix with which we convolve the source pixel. This algorithm takes care of boundary conditions in that it ignores the appropriate (either previous and next as the case may be) rows and columns for pixels that are along the image boundary.

Convolve(MATRIX Template, GENERICIMAGE SourceImage, INTEGER Row, INTEGER Column)

BEGIN

 INTEGER ProductofPreviousRow

 INTEGER ProductofCurrentRow

 INTEGER ProductofNextRow

 INTEGER Pixel00, Pixel01, Pixel02

 INTEGER Pixel10, Pixel11, Pixel12

 INTEGER Pixel20, Pixel21, Pixel22

 // STEP 1. Initialize

 Pixel00 := 0

 Pixel01 := 0

 Pixel02 := 0

 Pixel10 := 0

 Pixel11 := 0

 Pixel12 := 0

 Pixel20 := 0

 Pixel21 := 0

 Pixel22 := 0

 // STEP 2. Compute ProductofPreviousRow

 IF(Row .GT. 0)

 BEGIN

 IF(Column .GT. 0)

 Pixel00 := SourceImage.GetPixelAt(Row - 1, Column - 1)

 Pixel01 := SourceImage.GetPixelAt(Row - 1, Column)

 IF(Column .LT. SourceImage.GetWidth() - 1)

 Pixel02 := SourceImage.GetPixelAt(Row - 1, Column + 1)

 ProductofPreviousRow :=

 Pixel00 * Template[0, 0] + Pixel01 * Template[1, 0] + Pixel02 * Template[2, 0]

 END

 // STEP 3. Compute ProductofCurrentRow

 IF(Column .GT. 0)

 Pixel10 := SourceImage.GetPixelAt(Row, Column - 1)

 Pixel11 := SourceImage.GetPixelAt(Row, Column)

 IF(Column .LT. SourceImage.GetWidth() - 1)

 Pixel12 := SourceImage.GetPixelAt(Row, Column + 1)

 ProductofCurrentRow :=

 Pixel10 * Template[0, 1] + Pixel11 * Template[1, 1] + Pixel12 * Template[2, 1]

```
// STEP 4. Compute ProductofNextRow
IF( Row .LT. SourceImage.GetHeight() - 1 )
BEGIN
    IF( Column .GT. 0 )
        Pixel20 := SourceImage.GetPixelAt( Row + 1, Column - 1 )
        Pixel21 := SourceImage.GetPixelAt( Row + 1, Column )
        IF( Column .LT. SourceImage.GetWidth() - 1 )
            Pixel22 := SourceImage.GetPixelAt( Row + 1, Column + 1 )
        ProductofNextRow :=
            Pixel20 * Template[0, 2] + Pixel21 * Template[1, 2] + Pixel22 * Template[2, 2]
    END

    RETURN( ProductofPreviousRow + ProductofCurrentRow + ProductofNextRow )
END // Convolve
```

5.3.5 Results

In the following pages, we shall show the application of Sobel's edge detection algorithm to Exhibit 1 and Exhibit 2 that we considered in Chapter 2. The input image for this algorithm is obtained by using the color separation techniques discussed in Chapter 4 and the output image will be a Black and White image.

Let us consider the images that were generated as a result of applying color separation techniques to Exhibit 1 and Exhibit 2:

Figure 5.1: Color Separated Components of Exhibit 1

Figure 5.2: Color Separated Component of Exhibit 2

Now, let us see the effect of applying Sobel's edge detection algorithm to the color separated components of exhibit 1 with threshold set to different values.

Figure 5.3: Edge Detected Components of Exhibit 1 at Threshold = 5

Figure 5.4: Edge Detected Components of Exhibit 1 at Threshold = 127

Figure 5.5: Edge Detected Components of Exhibit 1 at Threshold = 250

The analogous results for Exhibit 2 are shown in Figure 5.6 - 5.8.

Figure 5.6: Edge Detected Component of Exhibit 2 at Threshold = 5

Figure 5.7: Edge Detected Component of Exhibit 2 at Threshold = 127

Figure 5.8: Edge Detected Component of Exhibit 2 at Threshold = 250

5.3.6 Shortcomings of first-order edge detection

Consider the following image:

1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9

If we use the basic Sobel vertical edge operator (as described above), the resultant image will have non-zero data all across it. For example, if the kernel

-1	0	1
-2	0	2
-1	0	1

is used, the result is:

8	8	8	8	8	8	8
8	8	8	8	8	8	8
8	8	8	8	8	8	8

Implementing the same template on this 'all eights' image would yield

0	0	0	0	0
---	---	---	---	---

This is just the same as the differentiation operator applied to a straight line, because if

$$y = 3x + 2,$$

then

$$dy/dx = 3$$

and

$$d^2y/dx^2 \text{ would be zero.}$$

This means that if we differentiate the gradient and the result is zero, then the original edge was a straight line.

The above example suggests that first-order edge detection algorithms like Sobel's will not generate any edges at all. In fact, images often come with a grey-level trend on them, since one side of a region is lighter than the other but there is no edge to be discovered in the region. Indeed if the shading is even, it perhaps, indicates the presence of a light source that is stronger at one end, or a gradual color change over the surface.

To find out more discrete changes, one could use higher-order templates. However, such templates no longer detect edges and are rather used to discover and enhance noise in an image.

Let us consider a second-order template, such as one derived from the Laplacian Set:

$$\begin{array}{ccc} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{array}$$

or

$$\begin{array}{ccc} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{array}$$

This template effectively removes the general gradient of lighting or coloring from an image and only enhances noise. It does not produce any information on direction which is seen as a function of gradual change. For more information on such higher-order templates, we refer the reader to [NM, NE, TT].

5.4 Tracing

If it is known that an object in an image has a discrete edge all around it, then it is possible, once a position on the edge has been found, to follow it around the object and back to the beginning.

There are a variety of image tracing techniques that are currently in practice which extracts the contour of images. For a detailed survey of these techniques, we refer the reader to [AL]. Most of these techniques, however, follow a two-step procedure in order to accomplish the task. They first

- (i) detect the edges of the given image based on a specific tolerance or threshold value to obtain a black and white image, and then
- (ii) follow the edges of the resulting black and white image to generate the vector of the contour.

There are quite a few advanced techniques that actually avoid going through the process of edge detection and edge following separately in order to avoid scanning through the whole image data more than once. This becomes a serious issue when the image size is very large. In addition, any loss of accuracy during the edge detection phase will be carried over to the edge following phase, resulting in more loss of accuracy.

In our study, we have developed an algorithm that will generate the contour of an image in one pass by combining both edge detection and edge following together. Our goal in the next section is to present this algorithm and its underlying principle. We shall present the formal algorithm that reads in the data from our data structure presented in Chapter 3 and generate the

required output. In Section 5.4.2, we will also present the results of the application of our algorithm to color separated images such as those considered in Chapter 2.

5.4.1 Contour Tracing Algorithm

The algorithm for tracing a given image is very simple. We start at a given position on the edge of a closed region (x, y) and evaluate all the 3×3 Sobel gradient values centered on each of the eight pixels surrounding (x, y) . We then choose the three pixels with the greatest absolute (neglecting the sign) gradient magnitude and place them in a three-column array, one column for each pixel position and order them in the row according to gradient magnitude. We thereafter choose the one with the greatest gradient magnitude. This pixel with the greatest gradient magnitude will be in one of the directions 0-7 with respect to the pixel (x, y) given by the following map, where * is the position of the current pixel (x, y) :

0	1	2
7	*	3
6	5	4

For example, if the maximum gradient magnitude was found from the Sobel operator centered around the pixel $(x + 1, y)$, then the direction would be '3'. If we denote the direction of travel by d , assuming that the shape is not "very irregular", we repeat the above algorithm by looking only in directions d , $(d + 1) \bmod 8$, and $(d - 1) \bmod 8$. If no suitably high value of the gradient magnitude is found, we will remove this pixel from the list and choose the next one of the three stored. If all three have been removed from the list, we then move up a row and choose the next best from the previous row. When the travel reaches the original pixel, or execution has gone on too long, or the number of rows in the list is very large, we terminate the search process.

Once a region is traced, we shall floodfill the whole region with a known color in order to continue tracing for the other closed regions. When the entire image is filled with the flood color, we stop the process.

We present all the algorithms that together perform the task of image tracing. We shall first introduce the structure `OURPOINT` which is used extensively in the algorithms that are presented, and later concentrate on the individual algorithms themselves.

```
// Structure used for tracing images
STRUCTURE OURPOINT
BEGIN
    INTEGER Row
    INTEGER Column
    BOOLEAN HaveWeTraced
    BOOLEAN BelongsToContour
END
```

In the above structure, *Row* and *Column* specify the exact position of the pixel under consideration. *HaveWeTraced* is used by the tracing algorithm to mark if the pixel has already been traced. *BelongsToContour* is used to establish whether belongs to any traced region.

ALGORITHM TraceImageForAllContours: This algorithm generates the contour of all the closed regions within the image, *SourceImage* which is obtained by color separating the image using the algorithm *ColorSeparate* which was presented in Chapter 4. The parameter *Threshold* is used to compute the Sobel's gradient magnitude at each pixel of the source image. The algorithm traverses through the image data, starting at the left-top of the image, and upon encountering a color difference, it starts to trace a single contour by invoking *TraceImageForOneContour*. Procedure *FloodFill* is used to fill the regions with known color.

```
TraceImageForAllContours( GENERICIMAGE SourceImage, INTEGER Threshold, INTEGER
Color = BLACK)
```

```
BEGIN
```

```
    INTEGER    Row
```

```
    INTEGER    Column
```

```
    OURPOINT StartPixel
```

```
    // STEP 1. Initialize variables
```

```
    Row := 0
```

```
    Column := 0
```

```
    WHILE( Row .LT. SourceImage.GetHeight() )
```

```
    BEGIN
```

```
        WHILE( Column .LT. SourceImage.GetWidth() )
```

```
        BEGIN
```

```
            IF( CALL SourceImage.GetPixelAt(Row, Column) != Color )
```

```
            BEGIN
```

```
                // STEP 2. Get a position on the edge of the image.
```

```
                StartPixel.Row := Row
```

```
                StartPixel.Column := Column
```

```
                StartPixel.HaveWeTraced := FALSE
```

```
                StartPixel.BelongsToContour := FALSE
```

```
                // STEP 3. Trace for a contour formed by StartPixel
```

```
                CALL TraceImageForOneContour( SourceImage, StartPixel, Threshold )
```

```
                // STEP 4. Mark the traced region as done
```

```
                CALL FloodFill( SourceImage, Row, Column, Color )
```

```
            END
```

```
        END
```

```
    END
```

```
    STOP
```

```
END // TraceImageForAllContours
```


ALGORITHM TraceImageForOneContour: This algorithm traces the given source image, *SourceImage*. *StartPixel* specifies the starting point on the image where the tracing should begin and *Threshold* specifies the threshold value to be used with Sobel's Kernel.

```

TraceImageForOneContour( GENERICIMAGE SourceImage, OURPOINT StartPixel,
INTEGER Threshold )
BEGIN
    INTEGER      Row, Column
    INTEGER      Direction1, Direction2, Direction 3
    OURPOINT NewPixel
    OURPOINT OldPixel
    OURPOINT NewPixel1, NewPixel1, NewPixel1
    INTEGER[8]   GradientMagnitude
    BOOLEAN      WeFoundHighValueOfGradientMagnitude

    // STEP 1. Loop
    WHILE()
    BEGIN
        // STEP 2. Initialize values
        NewPixel.HaveWeTraced := TRUE
        Row := NewPixel.Row
        Column := NewPixel.Column

        // STEP 3. Evaluate all the 3x3 Sobel gradient values
        // centered on each of the eight pixels surrounding (Row, Column)
        CALL ComputeGradientMagnitudes(SourceImage, GradientMagnitude, Row, Column )

        // STEP 4. Get three directions that have greatest absolute gradient magnitudes
        // The three directions are in the decreasing order of their absolute values
        WeFoundHighValueOfGradientMagnitude :=
            CALL GetDirectionOfThreeLargestGradientMagnitudes(
                GradientMagnitude, Direction1, Direction2, Direction3, Threshold )

        // STEP 5. Given the original pixel, NewPixel, and direction of any of the
        // eight pixel around it, we now determine the actual pixel value
        NewPixel1 := GetPixelFromDirection( NewPixel, Direction1 )
        NewPixel2 := GetPixelFromDirection( NewPixel, Direction2 )
        NewPixel3 := GetPixelFromDirection( NewPixel, Direction3 )

        // STEP 6. If any of these new pixels have higher GradientMagnitude, start to
        // trace through them. NewPixel1, NewPixel2, and NewPixel3 are in the
        // descending order of their Gradient Magnitudes, and we give priority to
        // the one with the highest Gradient Magnitude
        IF( .NOT(WeFoundHighValueOfGradientMagnitude) )
        BEGIN
            OldPixel := Pop()
            OldPixel.BelongsToContour := FALSE
        END
    END

```

```

ELSE
BEGIN
    ThisPixel.BelongsToContour := TRUE

    IF( GradientMagnitude[Direction3] )
    BEGIN
        IF( (NewPixel3.Row .EQ. StartPixel.Row) .AND.
            (NewPixel3.Column .EQ. StartPixel.Column) )
        BEGIN
            BelongsToContour := TRUE
            STOP
        END
    ELSE
        CALL Push( NewPixel3 )
    END
    IF( GradientMagnitude[Direction2] )
    BEGIN
        IF( (NewPixel2.Row .EQ. StartPixel.Row) .AND.
            (NewPixel2.Column .EQ. StartPixel.Column) )
        BEGIN
            BelongsToContour := TRUE
            STOP
        END
    ELSE
        CALL Push( NewPixel2 )
    END
    IF( GradientMagnitude[Direction1] )
    BEGIN
        IF( (NewPixel1.Row .EQ. StartPixel.Row) .AND.
            (NewPixel1.Column .EQ. StartPixel.Column) )
        BEGIN
            BelongsToContour := TRUE
            STOP
        END
    ELSE
        CALL Push( NewPixel1 )
    END
END
END

// Step 7. Choose the pixel with highest gradient magnitude
NewPixel := CALL GetTopElementOnStack()
END
STOP
END // TraceImageForOneContour

```

PROCEDURE FloodFill: This procedure fills all the pixels that are connected to the pixel denoted by the indices *Row* and *Column* in the given image *SourceImage* with the new color specified by *Color*. The algorithm starts by looking into the pixel at the given indices and

recursively traverses the image in order to walk through all the pixels that are connected to one another.

```
FloodFill(GENERICIMAGE SourceImage, INTEGER Row, INTEGER Column, COLOR Color)
BEGIN
```

```
    // STEP 1. Escape Hatch
```

```
    IF( SourceImage.GetPixelAt(Row, Column) .EQ. Color )
        STOP
```

```
    // STEP 2. Set this pixel to gray
```

```
    SourceImage.SetPixelAt( SourceImage, Row, Column, Color )
```

```
    // STEP 3. Manipulate adjoining pixels in the current row
```

```
    IF( Column .GT. 0 )
```

```
        CALL FloodFill( Row, Column - 1, Color )
```

```
    IF( Column .LT. SourceImage.GetNumAlignBytes() - 1 )
```

```
        CALL FloodFill( Row, Column + 1, Color )
```

```
    // STEP 4. Manipulate adjoining pixels in the previous row
```

```
    IF( Row .GT. 0 )
```

```
        BEGIN
```

```
            CALL FloodFill( Row - 1, Column, Color )
```

```
            IF( Column .GT. 0 )
```

```
                CALL FloodFill( Row - 1, Column - 1, Color )
```

```
            IF( Column .LT. SourceImage.GetNumAlignBytes() - 1 )
```

```
                CALL FloodFill( Row - 1, Column + 1, Color )
```

```
        END
```

```
    // STEP 5. Manipulate adjoining pixels in the next row
```

```
    IF( Row .LT. SourceImage.GetHeight() - 1 )
```

```
        BEGIN
```

```
            CALL FloodFill( Row + 1, Column, Color )
```

```
            IF( Column .GT. 0 )
```

```
                CALL FloodFill( Row + 1, Column - 1, Color )
```

```
            IF( Column < SourceImage.GetNumAlignBytes() - 1 )
```

```
                CALL FloodFill( Row + 1, Column + 1, Color )
```

```
        END
```

```
        STOP
```

```
    END // FloodFill
```

PROCEDURE ComputeGradientMagnitudes: This procedure computes the gradient magnitudes of all the pixels around the given pixel specified by *Row*, and *Column*. The values are returned in the parameter *GradientMagnitude*, which is an array of integer values. This procedure uses *GetSobelGradientMagnitudeAt* which was presented in Section 5.3.4

```

ComputeGradientMagnitudes(GENERICIMAGE      SourceImage,      INTEGER      []
GradientMagnitude, INTEGER Row, INTEGER Column )
BEGIN
    // STEP 1. Initialize
    GradientMagnitude[0] := 0
    GradientMagnitude[1] := 0
    GradientMagnitude[2] := 0
    GradientMagnitude[3] := 0
    GradientMagnitude[4] := 0
    GradientMagnitude[5] := 0
    GradientMagnitude[6] := 0
    GradientMagnitude[7] := 0

    // STEP 2. Compute gradient magnitudes of pixels in the previous row
    IF( Row .GT. 0 )
    BEGIN
        IF( Column .GT. 0 )
            GradientMagnitude[0] := GetSobelGradientMagnitudeAt(SourceImage, Row-1, Column-1
            )

            GradientMagnitude[1] := GetSobelGradientMagnitudeAt(SourceImage, Row - 1, Column )

            IF( Column .LT. SourceImage.GetWidth() - 1 )
                GradientMagnitude[2] := GetSobelGradientMagnitudeAt(SourceImage, Row-1, Column+1
                )
        )
    END

    // STEP 3. Compute gradient magnitudes of pixels in the previous row
    IF( Column .GT. 0 )
        GradientMagnitude[3] := GetSobelGradientMagnitudeAt(SourceImage, Row, Column - 1 )
    IF( Column .LT. SourceImage.GetWidth() - 1 )
        GradientMagnitude[4] := GetSobelGradientMagnitudeAt(SourceImage, Row, Column + 1
        )
    )

    // STEP 4. Compute ProductofNextRow
    IF( Row .LT. SourceImage.GetHeight() - 1 )
    BEGIN
        IF( Column .GT. 0 )
            GradientMagnitude[5] := GetSobelGradientMagnitudeAt(SourceImage, Row+1, Column-1
            )

            GradientMagnitude[6] := GetSobelGradientMagnitudeAt(SourceImage, Row + 1, Column
            )
        )
    END

```

```
    IF( Column < SourceImage.GetWidth() - 1 )
      GradientMagnitude[7] := GetSobelGradientMagnitudeAt(SourceImage, Row+1,
        Column+1 )
    END
  END // Convolve
```

PROCEDURE GetDirectionOfThreeLargestGradientMagnitudes: Given the array of gradient magnitudes, *GradientMagnitude*, this procedure establishes three directions at any given pixel that have the maximum gradient values. If any of these gradient magnitudes is less than the specified *Threshold*, then -1 would be returned to signal the tracing algorithm not to continue in that direction.

```
GetDirectionOfThreeLargestGradientMagnitudes( INTEGER [] GradientMagnitude,
    INTEGER Direction1, INTEGER Direction2, INTEGER Direction3, INTEGER Threshold )
BEGIN
```

```
    INTEGER max
```

```
    // STEP 1. Initialize
```

```
    Direction1 := -1
```

```
    Direction2 := -1
```

```
    Direction3 := -1
```

```
    max := 0
```

```
    // STEP 2. Find direction with first maximum gradient
```

```
    FOR( Direction = 0 thru 7, Direction++ )
```

```
    BEGIN
```

```
        IF( (abs(GradientMagnitude[Direction]) - Threshold) .GT. max )
```

```
        BEGIN
```

```
            max := abs(GradientMagnitude[Direction]) - Threshold
```

```
            Direction1 := Direction
```

```
        END
```

```
    END
```

```
    // STEP 3. Find direction with second maximum gradient
```

```
    max := 0
```

```
    FOR( Direction = 0 thru 7, Direction++ )
```

```
    BEGIN
```

```
        IF( ((abs(GradientMagnitude[Direction]) - Threshold) .GT. max) .AND.
            (Direction .NE. Direction1) )
```

```
        BEGIN
```

```
            max := abs(GradientMagnitude[Direction]) - Threshold
```

```
            Direction2 := Direction
```

```
        END
```

```
    END
```

```
    // STEP 4. Find direction with third maximum gradient
```

```
    max := 0
```

```
    FOR( Direction = 0 thru 7, Direction++ )
```

```
    BEGIN
```

```
        IF( ((abs(GradientMagnitude[Direction]) - Threshold) .GT. max) .AND.
            (Direction .NE. Direction1) .AND.
            (Direction .NE. Direction2) )
```

```
        BEGIN
```

```
            max := abs(GradientMagnitude[Direction]) - Threshold
```

```
        Direction3 := Direction
    END
END

// STEP 5. Decide if we found any higher gradient value at all
IF( (Direction1 .NE. -1) .OR. (Direction2 .NE. -1) .OR. (Direction3 .NE. -1)
    RETURN TRUE

RETURN FALSE
END // GetDirectionOfThreeLargestGradientMagnitudes
```

PROCEDURE GetPixelFromDirection: This procedure determines the row and column values of a pixel in an image, given the source point *OriginPixel*, and the direction value, *Direction*. The mapping between the directions and the corresponding pixel values from the source pixel is represented by the matrix:

0	1	2
7	*	3
6	5	4

This function returns the new pixel value which is established using the obtained row and column values.

GetPixelFromDirection(OURPIXEL OriginPixel, INTEGER Direction)

BEGIN

OURPIXEL NewPixel

SWITCH(Direction)

BEGIN

CASE 0:

BEGIN

NewPixel.x := Origin.x - 1

NewPixel.y := Origin.y - 1

END

CASE 1:

BEGIN

NewPixel.x := Origin.x

NewPixel.y := Origin.y - 1

END

CASE 2:

BEGIN

NewPixel.x := Origin.x + 1

NewPixel.y := Origin.y - 1

END

CASE 3:

BEGIN

NewPixel.x := Origin.x + 1

NewPixel.y := Origin.y

END

CASE 4:

BEGIN

NewPixel.x := Origin.x + 1

NewPixel.y := Origin.y + 1

END

CASE 5:

BEGIN


```
    NewPixel.x := Origin.x
    NewPixel.y := Origin.y + 1
  END

  CASE 6:
  BEGIN
    NewPixel.x := Origin.x - 1
    NewPixel.y := Origin.y + 1
  END

  CASE 7:
  BEGIN
    NewPixel.x := Origin.x - 1
    NewPixel.y := Origin.y
  END
END

RETURN NewPixel
END // GetPixelFromDirection
```

5.4.2 Results

We now submit the color separated images of Exhibit 1 and Exhibit 2.

Figure 5.9: Traced components of Exhibit 1

Figure 5.10: Traced components of Exhibit 2

We can see the success of our tracing algorithm in Figure 5.9 and Figure 5.10. The contour we have generated clearly defines the basic shape of the objects that we obtained by using the color separation technique. However, before being used by a specific image processing application, it may be necessary to eliminate the contours which are generated within the basic shape. We shall discuss this issue in Chapter 7 on conclusion and future works.

5.5 Conclusion

In this chapter we have focused our attention on the most interesting topic of our work - edge detection and tracing. Though it is a common practice to do region separation on the image data before establishing the shapes contained in it, it is not so common to combine edge detection and tracing together. We have made an attempt to do this in our scheme, and the results are far better than our expectations.

In the next chapter of this thesis, we shall discuss the implementation-related issues, and the points which we considered for the object oriented modeling required for the design and development of the entire system.

CHAPTER 6

DESIGN AND IMPLEMENTATION CONSIDERATIONS

6.1 Introduction

In this chapter, we will discuss the object oriented design considerations and specific implementation techniques used in our application. As has been discussed before in Chapter 3, the purpose of our generic image data structure has been designed to shield the application core from dealing with any specific graphics file format. In essence, in order to do edge detection and tracing, all of the algorithms that we have discussed in Chapters 4 and 5 will not have to be changed, if we decide to read image data from a different graphic file. The questions which we are now posed with are:

- How easy is it to read a graphic file of any format into our data structure? What are the changes required to accomplish this read operation?
- What effect would it have if we wish to add additional image attributes to our data structure?

We shall answer these and other questions in Section 6.2 of this chapter. We shall also take a look at the design aspects of our GenericImage data structure and the considerations involved in enhancing it.

In Section 6.3, we will discuss the design of our application frame work, and in Section 6.4 we shall describe the user-interface components, along with their screen snapshots, that have been built into our application in order to help us validate the algorithms that we have designed. We conclude this chapter in Section 6.5.

6.2 Design considerations with GenericImage

It is our intention to use GenericImage as a class so that we can reap the benefits from object oriented design techniques. We first consider what we mean by “class-based design” for GenericImage and what purpose it would serve.

For convenience, we shall again present the GenericImage data structure that we have discussed in Chapter 3:

```
structure GenericImage
{
    unsigned int          unType;
    unsigned int          unComprType;
    POINT                nWidth;
    POINT                nHeight;
    int                   nPlanes;
    int                   nBitsPerPlane;
    int                   nBytesPerLine;
    PALETTE               palette;
    BYTE                  *pData;
}
```

Now, by making a class out of GenericImage, we intend to hide all data attributes by reading them private to the class, and providing public message interfaces to manipulate the data. Thus we can use this mechanism to decide what data attributes should be available for use by the class instance, and what attributes must not. The class looks like:

```

class GenericImage
{
    private:
        unsigned int          unType;
        unsigned int          unComprType;
        POINT                nWidth;
        POINT                nHeight;
        int                  nPlanes;
        int                  nBitsPerPlane;
        int                  nBytesPerLine;
        PALETTE              palette;
        BYTE                 *pData;

    public:
        // Retrieval functions
        unsigned int GetType();
        unsigned int GetCompressionType();
        virtual unsigned int GetWidth();
        virtual unsigned int GetHeight();
        virtual PALETTE int GetPalette();
        BYTE * GetData();

        // Set functions
        void SetType( int nType );
        void SetCompressionType( int nType );
        void SetWidth( int nType );
        void SetHeight( int nType );
        void SetPalette( PALETTE palette );
        void SetData( BYTE *pData );

        virtual unsigned int GetNumberOfAlignedBytes() = 0;
}

```

Clearly, this class is an abstract class. There is one pure virtual function in the form of `GetBytesPerPixel()` that has to be implemented by any subclass of `GenericImage`. We shall presently see the advantages of specifying late-binding attributes to some of the messages and in the process, we hope to answer the questions posed earlier.

Consider first question that we encountered before: How easy is it to read a graphic file of any format. The data attributes that we have added into the `GenericClass` class is common to many different file formats. When reading any of these files, we just have to write a serialization

function that reads the image data and attributes from the archive into our class instance. Writing such an interface routine is just as straightforward as the one that we presented for reading Windows bitmap files in Chapter 3.

Next, it is interesting to see how we can use the concept of late-binding for object oriented systems to our advantage in interfacing our algorithm with those which deal with specific file format related issues.

In any image processing technique, data manipulation needs to know the number of aligned bytes that constitute a row of data *a priori*. The number of aligned bytes per row is a measure of the number of bytes per pixel, and this, of course, varies between different image types. For instance, in an RGB image, 3 bytes are used to represent a pixel (one each for Red, Green, and Blue colors); in case of CMYK images, there are four bytes for every pixel, and so on. Since our aim is to interface the application in spite of such differences, we shall abstract the class and let the specific subclass that corresponds to a specific image type implement the message that has a different interpretation of data. The advantage of this strategy is that we can work with the abstract base class in our image processing algorithms and still be able to get image specific information using the late-binding technique that is applauded in Object Oriented Design.

To be more specific, we now consider the RGB and CMYK file formats, and use *inheritance* to subclass RGBImage and CMYKImage from GenericImage. Now, assuming that we have algorithms to read image data from these two file formats, our subclasses look like:

```
class RGBImage
{
    public:
        unsigned int GetNumberOfAlignedBytes();

        PALETTE int GetPalette() { if (this->palette) return palette; }
        void SetPalette( PALETTE palette ) { this->palette = palette; }
}

class CMYKImage
{
    unsigned int GetNumberOfAlignedBytes();

    PALETTE int GetPalette() { return NULL };
    void SetPalette( PALETTE palette ) { this->palette = NULL; }
}
```

For an RGB image, the implementation of GetNumberOfAlignedBytes() would be of the following form:

```
unsigned int RGBImage:: GetNumberOfAlignedBytes()
{
    unsigned int nAlignedBytes;

    if( this->palette )
        nAlignedBytes = ((this->nWidth + 3) / 4) * 4;
    else
        nAlignedBytes = ((this->nWidth * 3 + 3) / 4) * 4;

    return nAlignedBytes;
}
```

The analogous form for a CMYK image is:

```
unsigned int CMYKImage:: GetNumberOfAlignedBytes()
{
    unsigned int nAlignedBytes;

    nAlignedBytes = ((this->nWidth * 4 + 3) / 4) * 4;

    return nAlignedBytes;
}
```


The extra computation of adding 3 and dividing 4 and then multiplying the resultant by 4 is to make sure that pixels in image data are always 4-byte aligned.

It is easy to see that our implementation of the edge detection or tracing algorithms (both described in Chapter 5) remain unchanged because they work with references to the base class instance. Calling any message with a reference to the base class would be automatically interpreted at run-time to get the relevant information from the correct class instance.

To answer the second question that we posed before, let us consider the addition of a resolution attribute to our class. In an image editing application, it is quite common to consider the resolution of the image while manipulating the data. It plays a significant role in determining the width and height of the image. Let us now see how we can add these attributes to our `RGBImage` class and still be able to work with the rest without having to modify anything at all.

```
class RGBImage
{
    private:
        unsigned int nXPixelsPerKM;
        unsigned int nYPixelsPerKM;

    public:
        unsigned int GetWidth();
        unsigned int GetHeight();
}
```

From the above we see that we will implement `GetWidth()` and `GetHeight()` in `RGBImage` to compute the width and height of the image based upon the resolution as defined by the attributes `nXPixelsPerKM` and `nYPixelsPerKM`. The rest of our algorithms will not have to change as a result of this because any call to `GetWidth()` and `GetHeight()` will be appropriately interpreted at run-time.

For a more detailed information on late-binding, inheritance and associated topics, we refer the reader to [GB2]. Information about C++ specific issues can be found in [ES].

6.3 Application Framework

Our application environment is built using Microsoft Foundation Classes (MFC). MFC is a set of base classes that help build the application framework with ease, and is modeled using Microsoft's Document-View paradigm. The basic idea here is to interface the data specific operations in an application in the form of a "Document" class from the user interactions in the form of a "View". For information about the Document-View paradigm, we refer the reader to [MFC].

In our application, CThesisDoc defines our "Document" class and CThesisView defines our "View" class. We shall now discuss the role that each of these two classes play in our application.

```
class CThesisDoc : public CDocument
{
    private:
        CImage *m_pImage;
        ...

    public:
        CImage *GetImage();
        void SetImage(CImage *pImage);

        void ColorSeparateImage();
        void EdgeDetect();
        void Trace();

        void Serialize(CArchive& ar);

        ...
}
```

CThesisDoc contains a reference to the GenericImage that we are interested in. It provides the serialization function to read and write the data from and to the image file respectively. CThesisDoc also contains the implementation of all of our algorithms. In addition

to all of this, it also contains other data structures that are used temporarily for storing the vector output from the trace algorithm etc. This class is derived from MFC's generic document class in order to derive benefits out of inheritance.

```
class CThesisView : public CView
{
    public:
        void DrawPoints( CDC *, POINT );
        void DrawImage( CDC *pDC );

        // User-selected menu events
        afx_msg void OnEdgedetect();
        afx_msg void OnColorseparate();
        afx_msg void OnTraceforcontour();

        // User-mouse events
        afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
        afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
        afx_msg void OnMouseMove(UINT nFlags, CPoint point);

        ...
}
```

CThesisView is the view representation of our application. It contains all relevant functionality that is necessary to allow the user to be able to review the functionality that CThesisDoc provides. On one hand, this class enables the user to select the option he wants on the user-interface, and on the other, it passes on the user triggered events to the document class so that these events can be handled in a straightforward manner. The following section on user-interface components lists all of the interfaces that are available to the user. In addition, it is the responsibility of this class to display any of the data on the screen. CThesisView has two drawing functions, one to draw the bitmap image and the other to draw an array of points, which we obtain from the tracing algorithm. Similar to its counterpart, CThesisDoc, this class is also inherited from MFC's base CView class.

6.4 User-Interface Components

In this section, we intend to present all the relevant user-interface components that make up our application workspace. We shall start with the overall application user-interface and move on to specific items of interest.

6.4.1 Application User-Interface

Here is the main application window that pops up when the application is launched:

Figure 6.1: Application User Interface

The main window consists of menus, toolbar, the status bar and the client window. We shall discuss each of these components in detail in the subsections to follow.

6.4.2 Application Menu

There are six main menus in our application. Let us see what each of them is intended for.

Figure 6.2: Main Menu

File Menu:

This menu consists of the standard items that are used to Load, Save, and Print the files.

Figure 6.3: File Menu

Edit Menu:

This is a part of the standard MFC application framework. It provides items that are used for standard clipboard operations such as Cut, Copy, and Paste.

Figure 6.4: Edit Menu

View Menu:

This menu consists of items that control the display of toolbars/status bars in terms of hide and show.

Figure 6.5: View Menu

Image Menu:

This menu consists of items that are used to invoke Edge Detection, Color Separation, or Tracing algorithms on the currently active image.

Figure 6.6: Image Menu

Window and Help Menus:

These are again part of standard MFC application framework. The Window menu contains a list of all the open images in the application so that the user can navigate through them

with ease. It also contains items that may be used to change the window display to be of a Tiled or Cascaded appearance. The Help menu can be used to interface with a help text.

Figure 6.7: Window Menu

6.4.3 Toolbar

The Toolbar is provided as an easy-to-use substitute for the application menu. It contains bitmap representation of various menu items. Figure 6.8 depicts our toolbar.

Figure 6.8: Tool Bar

The ColorPicker tool is of special interest. This tool is used to interactively pick and select a color on the image that is used as a seed for color separation. We have no menu item corresponding to it.

6.4.4 Status Bar

The main idea of a status bar is to adaptively provide status of operations to the user. In our application, we use a special pane (located at the right end of the status bar) to display the color value of a pixel which is identified by the current position of the ColorPicker tool.

Figure 6.9: Status Bar

6.4.5 Client Window

The Client Window region of the application window is used for displaying multiple documents, each of which may contain image data.

6.4.6 Dialogs

In addition to the components mentioned, our application provides dialogs for users to control the tolerance and threshold values while working with Color Separation and Edge Detection algorithms respectively. Figures 6.10 and 6.11 depicts these two dialog boxes. They appear whenever the user selects either Image-Color Separate or Image-Edge Detect menu items.

Figure 6.10: Tolerance Dialog

Figure 6.11: Threshold Dialog

6.5 Conclusion

When an application is built, it is not only necessary to provide fast and efficient methods to execute the task, but also necessary to make accommodations for further expansions. This is where the design of the application plays a crucial role. This is the reason why we have tried to explain the goal and flexibility of our design. GenericClass, which is key to all our functionality here, has been designed with great caution so that our application tasks do not only just support the Windows bitmap file format, but also files of any other format. Besides this, making additions to the data attributes of this class itself would again not pose any serious trouble to the working of the core segment.

In this chapter, we have also made an effort to present the user interface components of our application. Specific control values that are used by the edge detection and color separation algorithms are provided to the user in the form of easy-to-use dialogs. The resulting changes in the image data following the application of these values is depicted instantaneously in the form of new documents containing the changed data, thus providing the user with an opportunity to compare and contrast the original data with the changed one.

Having formally concluded the work we have done in this thesis, we conclude the thesis in the next chapter by considering the possible areas where we can expand the current scope of

our application. We also discuss the open problems that we have encountered as a result of the algorithms that we have designed.

CHAPTER 7

CONCLUSION

7.1 Summary

In our thesis, we have discussed three important algorithms that are fundamental to any image processing application. While we generalized existing black-and-white methods to implement color separation and edge detection, we introduced a novel contour tracing algorithm by devising our own method using Sobel's kernel functions. In addition, we have presented a detailed discussion on storage representation of bitmap files and have thoroughly discussed the Windows bitmap and TIFF file formats. In particular, we have designed a generic structure that has been used to bridge the three core algorithms with the image file details. As has been explained in the previous chapters, the primary objective of using this structure, instead of directly dealing with the image data from the raw file, is to isolate our core algorithms from having to deal with the "idiosyncrasies" of the specific file formats. In order to explain what is involved in reading different file formats into our data structure, we have presented two algorithms in Chapter 3 viz. *ReadPalettedImage* and *ReadTIFFImage* that map the most widely used Windows bitmap and TIFF file formats to our structure. The simplicity of these two algorithms clearly justify the power of this structure.

In chapters 3, 4, and 5, we have discussed the core algorithms and depicted the results of applying them to bitmap images. The results obtained by using color separation algorithm to identify different regions based upon gray-level difference is not surprising because of the very simplicity of the technique itself. Color tolerance values are used in order to just increase the contrast between the different regions. In section 7.2 (which mentions possible future avenues of work), we shall discuss the possibility of using advanced techniques for region separations of an

image. The results of using our tracing algorithm for generating the contour of bitmap images is fairly accurate and could, indeed, very well be used as a preprocessing element to continue with pattern recognition techniques. To verify the results that we obtain by tracing an image using Sobel's template, we have presented edge detection algorithms based on this template and clearly displayed the results. It can be seen that the increase in threshold value reveals significant details about the image.

The interesting fact about all the algorithms that we have presented in this work is that they can be used independent of each other. This means that the edge detection and tracing algorithms can be used independent of the color separation technique applied to our generic image structure. This also means that the changes that are necessary to reuse the current system in a different application domain is very minimal. To be more specific, one can just change the way objects are identified from the color separation technique presented here to any other suitable technique, and still continue to use the rest of the work unchanged, to obtain desired results.

On a different note, we would like to mention that there are certain issues which have not been addressed in this thesis. First of all, in our implementation, we have concentrated only on reading Windows bitmap files into our GenericImage structure. However, it is a straightforward task to extend the scope of our work by being able to read files from at least a few more different formats. In addition, it would be interesting if some benchmark testing files and metrics are used to profile the performance of our contour tracing algorithm. At present, we have just resorted to the perception of the human eye for judging the correctness of the results obtained.

7.2 Future Work

As with any other research work, we see a wide scope for future development for the algorithms presented here. We identify three areas of future work so that the current application can be fine tuned for specific needs.

7.2.1 Automated color separation technique

Identification of basic shapes in an image, based upon the color separation technique is well suited for a pattern recognition system. In such a system, we extract features from the basic shapes contained in the training samples and build up a database of them. Given a new testing sample, we simply identify the shape or shapes embedded in this sample and extract features from it. We use this feature list and try to identify the class which this belongs to using the feature database that has already been built. To extract features from the vector that we generate as a result of establishing the basic shape, we can use one of the many curve-fitting techniques. More details about how one can extract features from a vector of points can be found in [KF].

A typical image could contain many different colors. Manual separation of such an image into objects of different colors, as has been with the current scope of our application, could be time consuming. An automated method to identify such objects can help easily build the pattern recognition system which we have discussed above.

7.2.2 Different region splitting methods

In certain domains, the basic region splitting method which we have used in our application may not be sufficient. In such systems where the objects are identified with much more than a single color within a tolerance limit, we may have to use advanced region separation techniques. We have discussed one such technique in Chapter 4, but there is a plenty of scope to develop more robust methods to suite the needs of the specific application .

7.2.3 Elimination of sub-regions within the basic shape

Often, the purpose of generating the contours of an object is to determine its basic shape. When an object is traced, we sometimes end up with small contours residing within the shape of the object, specially when the object detail contains holes, where by “hole” we mean a closed region within another one that does not belong in there. As an example, let us consider the contour which is obtained by applying our tracing algorithm to one of the components of Exhibit 1:

Figure 7.1: Contour of a color separated component of Exhibit 1

It can be easily seen that there is no need to obtain the vector information of the sub-regions that are formed as a result of the holes in the original image since they serve no purpose. In the current form of the algorithms that we have presented, we do not deal with the elimination

of such sub-regions. However, any modifications to the current algorithm in terms of the removal of sub-regions will improve the overall effect obtained as a result of applying this algorithm. We leave this issue too for the future work.

BIBLIOGRAPHY

- [AL] Low, A., *Introductory Computer Vision and Image Processing*, McGraw-Hill, 1991.
- [BF] Brice, C. and Fennema, C., "Scene Analysis Using Regions," *Artificial Intelligence*, Vol. 1, 1970, pp. 205-226.
- [ES] Ellis, M. and Strostroup, B., *The Annotated C++ Reference Manual*, Addison Wesley, 1995.
- [FM] Fu, K. S. and Mui, J. K., "A Survey on Image Segmentation," *Pattern Recognition*, Vol. 13, 1981, pp. 3-16.
- [GB1] Baxes, G. A., *Digital Image Processing*, John Wiley & Sons Inc., 1994.
- [GB2] Booch, G., *Object Oriented Design with Applications*, Benjamin/Cummings, 1991.
- [GW] Gonzalez, R. C. and Wintz, P., *Digital Image Processing*, Addison-Wesley, 1977.
- [HM] Haralick, R. M., "Edge and Region Analysis for Digital Image Data," *Computer Vision, Graphics, and Image Processing*, Vol. 12, 1980, pp. 60-73.
- [HR] Hunt, R.W.G., *The Reproduction of Color in Photography, Printing and Television*, Fountain Press, Tolworth, England, 1987.
- [HS] Haralick, R. M. and Shapiro, L. G., *Computer and Robot Vision*, Vol. 1, Addison-Wesley. MA, 1993.
- [IS] Iannio, A. and Shapiro, S. D., "An Iterative Generalization of the Sobel Edge Detection Operator," *Proceedings of the IEEE Conference on Pattern Recognition and Image Processing*, Chicago, 1979, pp. 130-137.
- [KF] Fukunaga, K., *Introduction to Statistical Pattern Recognition*, Academic Press, Inc., 1972.

- [KK] Klinger, K., "Data Structures and Pattern Recognition," *Proceedings of the First International Joint Conference on Pattern Recognition*, Washington, D. C., 1973, pp. 497-498.
- [KL] Kay, D. C. and Levine, J. R., *Graphics File Formats*, Windcrest/McGraw-Hill, 1992.
- [KR] Kohler, R., "A Segmentation System Based on Thresholding," *Computer Graphics and Image Processing*, Vol. 15, 1981, pp. 319-338.
- [LJB] Lilly, P., Jenkins, J., and Bourdillon, P., "Automatic Contour Definition on left ventriculograms by image evidence and a multiple template-based model," *IEEE Transactions on Medical Imaging*, 8(2), June 1989, pp. 173-185.
- [LW] Lester, J. M., Willams, H. A., Weintraub, B. A., and Brenner, J. F., "Two Graph searching techniques for boundary finding in white blood cell images," *Computers in Biology and Medicine*, 8(4), 1978, pp. 293-308.
- [MFC] *MFC User guide*, Microsoft Press, 1992
- [MS] Microsoft Corporation, *Microsoft Windows Programmer's Reference*, Vol. 2, v3, Microsoft Press, Redmond, WA, 1990.
- [MV] Murray, J. D. and Vanryper, W., *Graphics File Formats, Second Edition*, O'Reilly & Associates, Inc. 1996.
- [NE] Nagel, H. and Enkelmann, W., "Investigation of Second Order Gray-Value Variations to Estimate Corner Point Displacements," *IEEE Conference on Pattern Recognition and Image Processing*, Munich, 1982.
- [NM] Nagar, M. and Matsuyama, T., "Edge Processing Smoothing," *Computer Graphics and Image Processing*, Vol. 9, 1979, pp. 394-407.
- [NW] Niblack, W., *An Introduction to Digital Image Processing*, 2nd edition, Prentice Hall,

1986.

- [PC] Petzold, C., *What's New in Bitmap Formats: A Look at Windows and OS/2*, PC Magazine, 11 September 1990, pp. 403-410.
- [PD] Philips, D., "LZW Data Compression," *The Computer Application Journal*. Circuit Cellar Ink, Vol. 27, June/July 1992, pp. 36-48.
- [RS] Rimmer, S., *Bit-Mapped Graphics*, Windcrest Books, 1992
- [ST] Swan, T., *Inside Windows File Formats*, Sams Publishing, 1993.
- [TF1] Aldus Corporation, *TIFF Developer's Toolkit*, Revision 6.0, Seattle, WA, June 1992.
- [TF2] Campbell, J., *The Spirit of TIFF Class F*, Cygnet Technologies, Berkley, CA, April 1990.
- [TT] Tomita, F., and Tsuji, S., "Extraction of Multiple Regions by Smoothing in Selected Neighborhoods," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-7, 1977, pp. 107-109.
- [WNR] Weszka, J.S., Nagel, R.N., and Rosenfeld, A., "A Threshold Selection Technique," *IEEE Transactions on Computers*, Vol. C-23, 1974, pp. 1322-26.
- [ZH] Zuniga, O. and Haralick, R. M., "Integrated Directional Derivative Gradient Operators," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-17, 1987, pp. 508-517.
- [ZL] Ziv, J. and Lempel, A., "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, Vol. 23, no. 3, 1977, pp. 337-343.

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc.
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved