

Copyright is owned by the Author of the thesis. Permission is given for a copy to be downloaded by an individual for the purpose of research and private study only. The thesis may not be reproduced elsewhere without the permission of the Author.

The Development of a Visual Language for Image Processing Applications

A thesis presented in partial fulfilment of the requirements for
the degree of Doctor of Philosophy in Computer Science at
Massey University, Palmerston North, New Zealand.

Phillip Michael Ngan

1992

006.42

Nga

DC20

Abstract

The research described in this thesis is based on the hypothesis that computer support for the heuristic development of image processing algorithms can be improved by the provision of a human-computer interface that is suited to the task. Current interfaces are largely text based and are not specifically designed to provide this support. It is suggested that an interface incorporating aspects of menu-based, direct manipulation, and visual languages, can provide the necessary support.

The research of this thesis begins with an analysis of the task of image processing algorithm development. It is found that in development, algorithms are more appropriately viewed as data-oriented networks of imaging operations than as process-oriented lists. The representation of algorithms in most current interfaces, particularly in text based systems, do not clearly convey the multi-threaded data paths in an algorithm. However, a data-oriented representation expresses such parallel paths clearly and naturally.

The second finding of the analysis is that human designers employ a set of problem solving strategies or heuristics in the interactive development of algorithms. These strategies include the top-down decomposition of the imaging task, the identification and focus of critical sub-goals, the progressive refinement of an algorithm, and the modification of existing algorithms. These heuristics are used implicitly in the development of algorithms, but the ease with which they are used in text based interfaces is restricted by the lack of appropriate interactive facilities.

An evaluation of interface techniques suggests that an interface that combines aspects of menu-based, direct manipulation, and visual languages, can support the required interaction for the heuristic development of algorithms. The required data-flow view can be provided by an iconic data flow language. Such a representation is highly visible, and can be interpreted by a user at a glance. Quick and convenient specification and editing of a data flow network can be performed via direct manipulation interaction facilities. The search for suitable operations can be facilitated by menu systems.

On the basis of the arguments for the adoption of a data-flow representation of algorithms, a problem solving approach to algorithm development, and highly interactive facilities, a software package, called OpShop, has been implemented. Examples which compare OpShop to text based systems show that four major tasks involved in algorithm development are better supported with the new interface. These tasks are the visualisation of multi-threaded data paths, the interactive experimentation with algorithm parameters, the modification of algorithm topology, and the comparison of alternative algorithms. In these examples, the OpShop software represents the tangible outcome of the design for an interface that specifically supports the heuristic development of image processing algorithms.

Table of Contents

Abstract.....	ii
Preface	vi
Acknowledgments	ix
1 Introduction	1
1.1 The context	1
1.2 The approach	3
1.3 Thesis overview.....	4
2 Image Processing Algorithm Development.....	6
2.1 Introduction	6
2.2 General structure of an algorithm.....	9
2.2.1 Outside world.....	10
2.2.2 General image.....	11
2.2.3 Segmented image	12
2.2.4 Compact structures	13
2.2.5 Shape measurement	14
2.2.6 Pattern classification.....	15
2.2.7 An example algorithm	15
2.3 Data-oriented view of algorithms.....	17
2.4 Algorithm development is problem solving.....	18
2.5 The solution graph.....	19
2.6 Solution development.....	20
2.7 The heuristic approach.....	22
2.7.1 The broad decomposition of the task.....	22
2.7.2 Identification of a critical subgoal.....	22
2.7.3. Jumping to an arbitrary location.....	23
2.7.4 Application of well known techniques.....	23
2.7.5 Exemplar based development	24
2.7.6 Progressive refinement	24
2.8 Summary and Conclusions.....	25
3 Human Computer Interface Techniques	26
3.1 Introduction	26
3.2 Command line interfaces	27
3.2.1 Advantages	28
3.2.2 Disadvantages.....	29

3.3	Menu based interface.....	30
3.3.1	Advantages	33
3.3.2	Disadvantages.....	33
3.4	Direct manipulation interface	34
3.4.1	Key characteristics.....	37
3.4.2	Disadvantages.....	38
3.5	Visual language interface.....	39
3.5.1	Visual programming vs program visualisation	39
3.5.2	Visually transformed vs naturally visual	41
3.5.3	Program responsiveness	42
3.5.4	Advantages	43
3.5.5	Disadvantages.....	43
3.6	Summary and conclusions	44
4	An Interface Design.....	46
4.1	Design philosophy.....	46
4.1.1	Stepwise refinement.....	48
4.1.2	Dynamic exploration.....	48
4.2	Generation.....	49
4.2.1	An iconic data flow language for image processing.....	49
4.3	Execution	53
4.3.1	Specification of input and output data	53
4.3.2	Specification of parameter values	54
4.3.3	Operation invocation	56
4.4	Evaluation of the results	57
4.5	Summary and Conclusions.....	58
5	OpShop: An Implementation.....	60
5.1	An overview of OpShop	60
5.2	Elements of the visual language environment	63
5.2.1	Whiteboard.....	63
5.2.2	Operations.....	64
5.2.3	Algorithms.....	68
5.2.4	Subflows.....	71
5.3	User Interaction	74
5.3.1	Execution	74
5.3.2	Parameter exploration	74
5.3.3	Topology exploration	75
5.4	The OpShop software design.....	77
5.4.1	Why THINK C?	77
5.4.2	Data structures.....	79
5.4.3	A data-driven execution scheme	83

5.5	Feasibility of continuous interaction	88
5.6	Summary and conclusions	90
6	OpShop Examples	92
6.1	Introduction	92
6.2	Data flow view of an algorithm	93
6.2.1	Colour classification	93
6.2.2	A command line implementation.....	95
6.2.3	OpShop implementation.....	96
6.3	Parameter exploration	97
6.3.1	The Abingdon Cross benchmark.....	97
6.3.2	A command line implementation.....	98
6.3.3	An OpShop implementation.....	100
6.4	Topological exploration	102
6.4.1	Segmentation of a non-uniformly illuminated scene	102
6.4.2	A command line implementation.....	105
6.4.3	An OpShop implementation.....	106
6.5	Choosing between algorithm alternatives.....	107
6.5.1	Generation of alternative solutions.....	107
6.5.2	Command line implementation.....	107
6.5.3	An OpShop implementation.....	108
6.6	Summary	109
7	Summary and Conclusions.....	110
7.1	Suggestions for future work	113
	References.....	117
	Summary of the OpShop Software.....	127
A1.1	Introduction	127
A1.2	System requirements.....	128
A1.3	Description of Operations.....	128
A1.3.1	General Menu	128
A1.3.2	Preprocessing Menu.....	135
A1.3.3	Segmentation Menu	140
A1.3.4	Measurement Menu.....	145
A1.3.5	Miscellaneous interface elements.....	145
A1.4	Operation Icons.....	146
A1.4.1	General Menu	146
A1.4.2	Preprocessing Menu.....	146
A1.4.3	Segmentation Menu	146
A1.4.4	Measurement Menu.....	146

Preface

Background

This project grew out of a perceived need for a highly interactive computing environment to support the heuristic approach to imaging algorithm development. I was introduced to the field of image processing while I was an undergraduate Electrical Engineering student at the University of Canterbury. During the course of studying my B.E. and M.E. I had used, or at least been exposed to, four interactive image processing systems. These systems, written by post-graduate students of Richard Bates and Bob Hodgson, served their intended purposes well; that of providing computing environments to support research into image processing. However, it never occurred to me then, that good engineers can (sometimes) create bad interfaces¹. These were fine systems for performing post graduate research, but they were not necessarily the easiest to use.

Richard Bates, on finding out that I was considering continuing my studies overseas, strongly recommended Mark Apperley to me as a supervisor for Ph.D. studies. As it turned out, Mark Apperley had a happy blend of an electrical engineering background and research interests in both image processing and human-computer interaction. After enrolling in a Ph.D degree course, Bob Hodgson and Don Bailey, who had been key figures in the image processing work at Canterbury, joined the staff at Massey University. They accepted invitations to co-supervise my Ph.D. project. So it was with a definite research objective and a proficient team of supervisors that I started on the work reported in this thesis.

Original contributions

This thesis describes the development of an interactive user interface for image processing applications. The primary goal of the interface is to facilitate the development of imaging algorithms by enhancing a user's ability to directly interact with the imaging task. In the course of pursuing this goal a number of original contributions were made:

- *A new viewpoint concerning the task of image processing algorithm development.* It is recognised that, for development, an algorithm is more appropriately viewed as a data-oriented network of operations than as a process-oriented list of imaging operations. This latter view is prevalent in most current systems for imaging algorithm development.
- *A data-oriented view, called the solution graph, was formulated as a graphical model to represent image processing algorithms.*

¹ Genter, D.R. & Grudin, J. (1990): Why good engineers (sometimes) create bad interfaces, *CHI'90 Conference Proceedings*, Seattle, Washington, 1 - 5 April, 1990.

- *Another new viewpoint concerning the task of image processing algorithm development.* The pragmatic approach to the development of an algorithm can be regarded as an example of a problem solving task. It is demonstrated that the pragmatic approach involves the application of a set of heuristics, to increase the likelihood of finding a satisfactory solution. The developed interface incorporates a graphical language that enables the explicit representation of these heuristics.
- *Demonstration of a user-oriented approach to software design.* The user-oriented approach to software design was shown to be feasible provided that a designer is fully aware of the user's needs and interests regarding the task, and that the designer carefully applies interaction techniques to support users' concerns. In this study, the end-product is an interactive environment for the development of imaging algorithms.
- *Demonstration of ways to simplify algorithm development.* Four demonstrations were given to show how the implemented design simplified the development of imaging algorithms.

As with most software projects, the process of development tends to be more evolutionary than sequential. In practice, a development of a product rarely proceeds in the distinct stages of analysis, design, implementation, and testing, as indicated by the structure of this thesis. Rather, a typical development process involves iteration of these stages. The research for this thesis was conducted in such an evolutionary manner. Many of the insights and perceptions presented were a product of hindsight and reflection of previous iterations. Aspects of the design were formalised before, during, and after, the writing of software. Although the form of a thesis constrains one to present material in a linear sequence of logical steps, the reader should be aware that the ideas were not necessarily developed in that order.

Publications

The following publications and presentations were prepared during the research for this thesis:

- Ngan, P.M., Apperley, M.D. & Hodgson, R.M. (1989): Towards a user model for image processing and analysis, *Proceedings of the 4th New Zealand Image Processing Workshop*, Auckland Industrial Development Division, Department of Scientific and Industrial Research, Auckland, New Zealand, 14-15 August, 1989, 94-100.
- White A.G. & Ngan, P.M. (1989): The measurement of red colour of apple fruit using digital imaging, *Proceedings of the 4th New Zealand Image Processing Workshop*, Auckland Industrial Development Division, Department of Scientific and Industrial Research, 14-15 August, 1989, 13-19.
- Ngan, P.M., Apperley, M.D. & Hodgson, R.M. (1990): The user-oriented development of an interface for image processing, *Proceedings of the 5th New Zealand Image Processing Workshop*, Massey University, Palmerston North, New Zealand, 9-10 August, 1990, 63-68.
- Cochrane, T., Matthew, C., Apperley, M.D. & Ngan, P.M. (1990): Plant root length and diameter determination using an image analysis thinning algorithm, *Proceedings Agronomy Society of New Zealand*, 20, 1990, 77-82.
- Ngan, P.M. (1991): OpShop: an iconic programming system for image processing, *Proceedings of the Australasian Apple University Consortium Conference*, Australian National University, Canberra, Australia, 1-4 July 1991.
- Ngan, P.M. & Apperley, M.D. (1992): Opportunistic design in image processing algorithm development, submitted to *Journal of Visual Languages and Computing*.

Acknowledgments

During this Ph.D., I was fortunate to have had three exemplary researchers as my supervisors: Mark Apperley, Bob Hodgson, and Don Bailey. Each tended to help me in different aspects of my study, yet these differences were complementary. This complementarity somewhat parallels the concept of generation-execution-evaluation discussed in the thesis. Mark, my first supervisor, helped me generate the ideas of this work through his acute insight of important and promising avenues of research. Don helped in the carrying out of the work by providing timely and key advice during both the programming and writing phases. Bob often served as the beta-tester for the research and provided greatly appreciated constructive criticism to ensure the work was carried out to a full professional standard. I greatly appreciate the support, encouragement, and openness, that my supervisors have extended to me during this apprenticeship in the craft of research.

Kirsten, my fiancé and constant source of love and support during this work, exercised her meticulous editing skills on many of the chapter drafts.

I appreciate the contribution of Paul Mudgeway relating to the graphical design of the icon symbols for OpShop. His work demonstrates the value of involving graphics designers in graphics-oriented HCI projects.

Throughout the Ph.D., I received financial support in the form of the VC's Ph.D. Study Award. I am indebted (and thankfully not financially!) to Massey University for this assistance.

A man can do nothing better than to eat and drink and find satisfaction in his work. This too, I see, is from the hand of God ... (Eccles. 2:24). This work has been carried out to the glory of God.

Chapter 1

Introduction

1.1 The context

Humans acquire information primarily through the sense of sight (Barraga, 1986). With this sense, humans have the capacity to perform a rich and sophisticated range of visual tasks, yet do so unconsciously. Examples of human visual tasks are identified in the following excerpt from a popular children's book (Milne, 1928, p.92ff.); the visual tasks are indicated in italics. In this scene, the characters are leaning over the side of a bridge watching sticks float down a river as part of a game call Poohsticks ...

"I can see mine!" cried Roo,

(object detection)

"No, I can't, it's something else. Can you see yours, Piglet?

(object classification)

I thought I could see mine, but I couldn't. There it is!
No, it isn't. Can you see yours, Pooh?" ...

(in need of image enhancement)

"It's coming!" said Pooh.

(motion analysis)

"Are you sure it's mine?" squeaked Piglet excitedly. *(pattern recognition)*

"Yes, because it's grey."

A big grey one. Here it comes! A very - big - grey - *(feature measurement)*

Oh, no, it isn't, it's Eeyore." *(optical character recognition!)*

To augment our natural sense of sight, humans have created vision systems to acquire spatially distributed information of phenomena and scales that cannot be perceived unaided by the eye. Scott (1990) explains that: "These nonbiological channels expose otherwise invisible domains to our systematic scrutiny. They allow us to experiment and understand, to craft new generations of technology from the previously unobserved. Most major innovations in science and technology have been preceded and simulated by significant extensions the natural sensorium by technical means."

Artificial vision systems are created not only to extend human information gathering into other domains, but also are created "to simulate natural sensory processes, endowing instruments with capacities that effectively mimic our own" (Scott, 1990). In many situations, these "instruments" are required to more than mimic human vision, but to operate with a precision and speed that is beyond the normal capability human vision. One such situation is cited by Scott: "Commercially available machine vision system can accurately screen parts passing by on a conveyor belt at a rate at which the human eye perceives just a blur ..."

The key functions of any vision system, be it natural or artificial, are the acquisition of images and the processing images (Corn, 1983). The main function of the processing stage is to attribute meaning to a scene; this is also known as scene interpretation. A vision system interprets a scene by measuring prescribed parameters of a scene. A simple example of interpretation is the situation where a viewer detects the parameters X and Y in a scene, and as a result attributes the meaning Z to it. For humans, the visual parameters are set within the contexts of culture, profession, and experience. For example, a native Chinese person and an Anglo-Saxon would differ in their ability to transcribe a sentence composed of Chinese ideograms or a sentence composed of English characters; an artist may interpret a full moon against a clear night sky differently from an astronomer; a seasoned farmer would probably interpret cloud formation differently from an urban dweller. In artificial vision systems, the visual parameters used in scene interpretation are usually prescribed by a computer program. Therefore, the interpretation of a scene is performed according to the process specified in a computer program.

The processing of image data is typically specified in a computer program as an algorithm. In image processing terminology, an algorithm is a sequence of

mathematical operations which performs some prescribed imaging task. Brumfitt (1984) notes that the development of a system for an imaging application involves two distinct phases: the research phase in which the algorithm is developed, and the engineering phase in which a prescribed algorithm is implemented in such a way as to meet particular performance criteria. This distinction implies two types of users for image processing systems: those who develop algorithms and those who use the algorithm to achieve a specific task. This thesis concerns provision of a computer environment to aid the first type of user in the task of algorithm development.

The central problem in image processing algorithm development is to find an appropriate sequence of mathematical (or imaging) operations which produces the desired result. However, there is usually no single best solution. In some situations, several alternative algorithms may be devised to achieve the result, while in another situation, even one satisfactory algorithm may be extremely difficult to find. The field of image processing lacks a general theoretical underpinning (Haralick, 1986) and therefore algorithms are generally not developed along a pre-determined solution path.

1.2 The approach

It is the view of this author and his supervisors that the development of image processing algorithms - at least in the foreseeable future - is best performed using an empirical approach. By this approach, a human designer develops an algorithm by performing a series of experiments, where each experiment is designed by experience, trial-and-error learning, and intuition. At present, the only alternative to this approach are design methods that seek to emulate natural vision systems (Marr, 1982; Wilson, 1987). While these approaches may ultimately lead to the production of systems that perform optimally for the task to which they are applied, it may take many years for the field of image processing to gain the knowledge of human vision and general vision sufficient for the implementation of such systems. The pragmatic alternative is to devise systems to perform specific tasks proficiently and cost-effectively (Petkovic & Wilder, 1991). Although these systems may perform only the application for which they were designed and may not resemble human vision in the method of computation, nevertheless they may address the immediate needs for the extraction of information from image data.

The research described in this thesis is based on the proposition that support for experimental development of algorithms can be improved over existing methods by the provision of a human-computer interface that is suited to the task. The research performed to uphold this position involved: analysing the task domain of image processing algorithm development, evaluating interface styles currently used in image processing systems, developing guidelines for an interactive

computer environment, implementing a software package according to the proposed guidelines, and evaluating the software package.

The tangible product of the research this reported in this thesis is a program for Macintosh computers, called OpShop, that supports the interactive development of image processing algorithms. The implementation of this package is thoroughly grounded on the principles of user-centred design espoused in this thesis. The interactive environment includes a wide range of imaging operations². The software is written in a high level programming language, "Think C", and contains 42 000 lines of source code³. Features of the OpShop environment are described in detail in Chapter 5 and examples of its use are demonstrated in Chapter 6. Descriptions of each imaging operation in the package and the hardware and software requirements to run the package are described in Appendix 1.

1.3 Thesis overview

The research conducted for this study takes a complete path from an analysis of the application domain, to a review and evaluation of current interface techniques, to an application of a particular (user-centred) design philosophy, and finally culminating in an implementation of a software package, which is evaluated against the development goals. The research reported in this thesis deliberately follows a depth-first path through the field of development possibilities. Many interesting issues are left unvisited so that a complete development could be achieved. The principle restriction of the research is that only image analysis algorithms are considered; however, the structures developed may be readily extended to other application domains. It was not the intent of this thesis to describe the development of a definitive image processing system; such a study would entail many aspects of a system including: extensibility and execution efficiency. However, this thesis contributes towards the design of a definitive system by studying the interaction that occurs between a human designer and an algorithm.

A thorough working knowledge of the application domain is a prerequisite for a comprehensive interface design. Chapter 2 provides an overview of the application domain of image processing algorithm development. Two models that describe algorithms are presented: the first is the traditional model of image operations arranged as a processing pipeline, the second is the result of a

² Twenty-one of the full set of sixty-three operations have been implemented.

³Of this body of code, 21 000 lines of code were written by the author, while the other 21 000 lines were supplied as the Think™ object-oriented class library (TCL).

new perception: that an algorithm can be represented as a data-oriented processing structure. The traditional view provides a well-established framework within which the imaging concepts relevant to this study is defined. The process-oriented view is found to restrict the user's capacity to develop algorithms. To address this deficiency, the data-oriented representation is introduced. Several problem solving strategies specific to image processing are presented within the framework of the data-oriented representation.

Chapter 3 presents an overview of the interface styles used in current image processing systems. Command line, menu-based, direct manipulation and visual language interface styles are examined and evaluated for their suitability for algorithm development. Such an overview represents an essential part of the overall development because any software package proposed in this thesis must not only incorporate features that improve interaction, but also build on the past successes of conventional interface styles.

Chapter 4 presents a design for an environment that supports the interactive development of imaging algorithms; this environment is an amalgam of some of the interface styles evaluated in Chapter 3. The design is directed by the principles of user-centred design, which are advocated by Norman (1990). The specification of the interface is jointly shaped by the issues concerning algorithm development raised in Chapter 2 and by the evaluations of interface styles made in Chapter 3.

Chapter 5 describes OpShop, the practical implementation for the design proposed in Chapter 4. The interactive features reviewed in Chapter 3 considered beneficial to algorithm development are integrated in this environment. Components of the environment are described in detail and the reasons for the particular implementations of these details are presented. The user interaction involved in the fabrication of an algorithm is described, with emphasis on the features that promote exploration of algorithm variations. Details of the system software including the data structures and the implementation of the data-flow execution scheme are described to give an appreciation for the internal operation of the system.

In Chapter 6, the image processing package is tested to demonstrate the way in which its features facilitate algorithm development. Four sample problems, each representative of an interactive task in image processing, are described. Each problem is discussed in two parts. In the first part, a command line implementation is presented to highlight the interaction requirements emphasised by the example task. In the second part, the equivalent solution in the OpShop environment is presented and contrasted with the command line algorithm.

Conclusions of the research and suggestions for future work are presented in Chapter 7.

Chapter 2

Image Processing Algorithm Development

This chapter reviews the concepts and terminology of image processing algorithm development relevant to this thesis. The first part of the chapter includes a conventional description of image processing algorithm development made within the framework of a data processing pipeline. The second part of the chapter introduces the idea of data flow representation and the concept of heuristic development of algorithms.

2.1 Introduction

The computer processing of pictorial information is best described by the diagram of Figure 2.1. This indicates how computer representation for such information falls into three general categories and shows the different types of processing relevant to each representation. Images are a direct representation of a scene and usually take the form of a two dimensional rectangular matrix of picture elements or *pixels*. Measurements derived from the image data characterise the useful information in the image, and these typically take the form of a small set of scalar numbers, such as a symmetry factor. The measurements in turn can be

used to infer some high level description of the objects in an image, for example, "the kiwifruit has a dropped shoulder shape". The field of *image analysis* involves the extraction of measurements from an image (Castleman, 1979), whereas *pattern recognition* is concerned with the interpretation of the measurements to infer abstract facts regarding the image (Freeman, 1986).

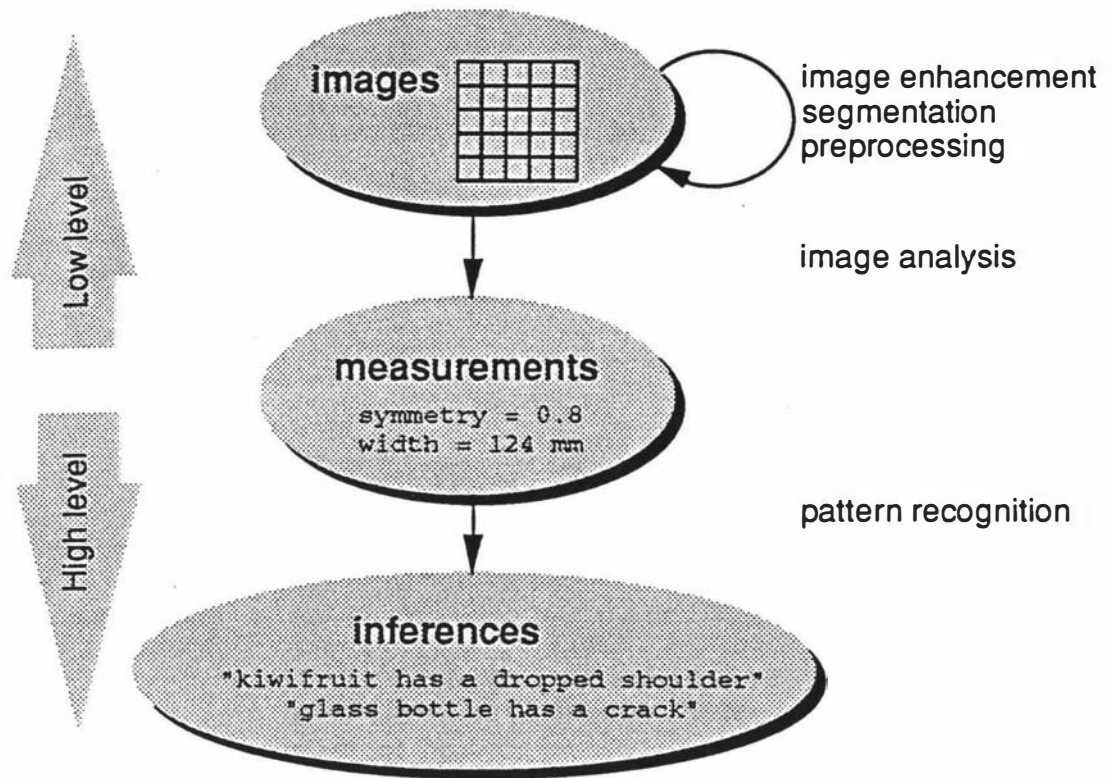


Figure 2.1: Computer processing of images.

The applications that solely employ image to image transforms are *image enhancement*, *segmentation*, and *preprocessing*. The purpose of *image enhancement* is to process an image so that the result is more suitable than the original image for a specific application (Gonzalez, 1987). The term "specific" is important because the processing is domain dependent. An example is to improve the sharpness of an x-ray image so that a physician can better "see" the anatomical structures. Another example is to improve the detectability of an object travelling along a conveyor belt so that its quality may be more readily assessed. In the first example, the enhancement is performed for the sake of a human; in the second, for the sake of a machine. Despite the diversity in imaging tasks, an image enhancement phase is nearly always required in the early stages of processing. Typical enhancement operations include contrast stretching, edge sharpening, and noise suppression. When image enhancement techniques are used to produce images that undergo further processing, the enhancement stage is sometimes called *preprocessing* (Haralick & Shapiro, 1991). *Segmentation* is a process which partitions an image into regions of similar properties; this is discussed more fully in section 2.2.3.

Image enhancement, image analysis, and pattern recognition have been used together successfully in commercial projects in the area of quality control. This application of imaging techniques is called *automated visual inspection* (Batchelor *et al.*, 1985) or *machine vision* (Freeman, 1989). A typical application is the identification and removal of defective products from a conveyor belt before they are packaged and sent to customers. This usually involves a camera set over a conveyor belt to capture the image of every item that passes. Defects are inferred from key measurements made of the imaged objects. If an object is identified as defective, the imaging sub-system signals an electro-mechanical arm to remove the defective item from the conveyor. Often, much of the development effort for a machine vision installation is devoted to the capture of images and the computation of results at high frame rates. A typical example of a machine vision system is Avdel (Hollington, 1984), which inspects the head diameter and stem length of rivets at a rate of 10 items per second. To capture images at this rate, without blurring, requires specialised equipment such as line scan cameras or strobe lighting. At present in 1992, computation at such speeds can only be performed by custom hardware. Temperature and lighting variations, vibration, and dust, together present a harsh environment in which an imaging system must operate, but nevertheless the system must be resilient to these constraints if it is to work properly.

Image enhancement, image analysis, and pattern recognition are regarded as the "low-level" components for vision systems that use high level knowledge to interpret a scene. The essence of the automated visual inspection problem is captured in the question, "Does object X with property Y exist in the image?". The description of X and Y and the method for their extraction is hard coded into the inspection algorithm. In contrast, the task of *image understanding* systems (Lawton & McConnell, 1988; Fischler & Firschein, 1987) or *computer vision* systems (Ballard & Brown, 1982) are to provide a high-level description of a (usually three-dimensional) scene. Scene interpretation is often an open ended task, which is summed up by the question "what objects exist in this three-dimensional scene?". In the United States, image understanding projects are strongly oriented towards military applications because their funding comes principally from the Defense Advanced Research Projects Agency (DARPA). Typical image understanding projects relate to autonomous vehicle navigation (Thorpe & Kanade, 1989), robot vision (Fennema *et al.*, 1989), smart weapons (Bjorklund *et al.*, 1989), and photointerpretation (McKeown *et al.*, 1985). There are currently two broad schools of thought in the sub-discipline of image understanding. One school maintains that theory should lead implementation (Jain & Binford, 1991; Snyder, 1991), while the other school insists that the theory is advanced by practical experimentation (Aloimonos & Rosenfeld, 1991; Huang, 1991; Bowyer & Jones, 1991).

Any study must restrict the scope of its investigation to keep the project to a manageable size and to increase the likelihood of producing meaningful results. In this thesis, image processing algorithms are assumed to relate only to image enhancement and analysis applications to avoid the need to deal with the

artificial intelligence aspects associated imaging understanding systems. Artificial intelligence systems have the complex architectures needed to make inferences from knowledge bases and for handling three dimensional geometric models. A natural distinction exists between image processing and image understanding; this is reflected in the recent division of the journal "Computer Vision, Graphics, and Image Processing" into two separate publications, "Image Understanding" and "Graphical Models and Image Processing" whose first volumes were published in 1991. The conclusions of this thesis, although based only on image enhancement and analysis algorithms, will be of general relevance as all vision systems must include "low-level" processing at some stage in its processing (Rosenfeld, 1988).

2.2 General structure of an algorithm

An *algorithm* can be considered as a prescribed sequence of imaging operations that refines data in stages. As the data progresses through each stage, the information is progressively refined until the final descriptor is formulated. Most algorithms can be represented by the processing pipeline shown in Figure 2.2. The lines with arrows represent groups of imaging operations that work together to transform data from one state to another. Occasionally the transformation between data states can be accomplished by a single operation, but usually it is achieved by several operations. The combined set of operations is generally regarded as an algorithm.

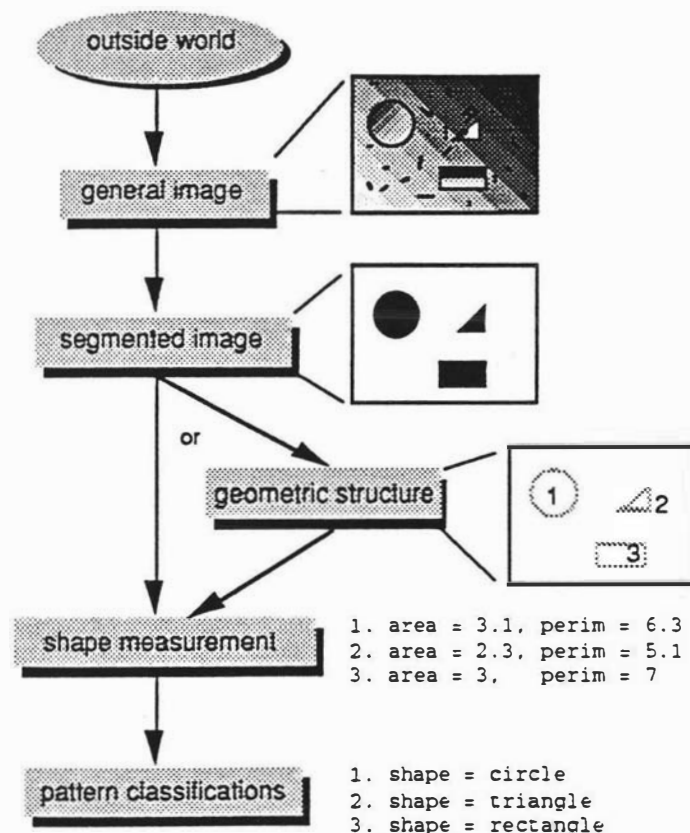


Figure 2.2: Data transformations involved in most image processing algorithms.

As the data passes through each stage of the processing, it increases in structure and decreases in volume but maintains the information that is ultimately extracted (Rosenfeld, 1984). The changes to the data are illustrated by the progression of the example data in Figure 2.2. At the start of the processing, the image data contains geometric objects and noise. At the next stage, the segmented image contains only the "silhouettes" of the geometric objects. Then, the object's outlines are extracted and encoded as a linked lists of points called chain codes. The outlines are then dispensed with altogether and each object is represented by characteristic measurements of area and perimeter. Finally, each shape is named. Detailed discussion of each stage follows in the next six sections.

2.2.1 Outside world

The *outside world* refers to a scene or object whose image will be captured by some sensing device. Freeman (1986) categorises image processing applications by the size of the scene compared to the camera. The first class contain scenes much larger than the camera: remote sensing of earth resources data or aerial reconnaissance fall into this category. The second class refer to scenes much smaller than the camera: for example, viewing blood cells or chromosomes through a microscope. The third class refer to scenes whose scale is comparable to that of the camera: applications include shape measurement of apples (White & Johnstone, 1991), optical character recognition, and the analysis of carpet wear. Applications in the third class resemble most closely human visual tasks because of the similar scale of the world sensed through the eyes.

Image capture involves the conversion of a scene into a representation suitable for input to a digital computer. Among the most common image capture devices are microdensitometers and solid state imaging arrays. A microdensitometer measures the intensity of light transmitted through film. A solid-state array can image either transmissive or reflective light intensity fields. These image sensors have groups of receptive elements arranged as a two dimensional array which allows natural three dimensional scenes to be captured in a single video frame time. Consumer video cameras, which typically use CCD arrays, are frequently used for image capture because of their versatility and relative low-cost.

The image capture devices mentioned above sense the outside world in a rectangular spatial geometry, but not all devices work in this way. Computerised tomography (CT) machines take measurements in the form of projections. A *projection* is a one dimensional distribution of image intensities integrated over a set of parallel or divergent rays travelling through a transverse slice of a body. In a CT scan, a set of projections - each displaced by a small angular increment - is measured through a body for a single transverse plane. In x-ray CT (Hounsfield, 1980), projections are a measure of the x-ray attenuation experienced by the rays passing through the bones and organs of a body. In emission CT (Budinger, 1980; Budinger *et al.*, 1979; Knoll, 1983), the projections are a measure of the density distribution of a radioisotope on the imaged plane. The cross-section of the body

can be inferred from the projection sets by a mathematical transformation technique called *reconstruction from projections* (Herman, 1979; Herman, 1980; Bates & Peters, 1971; Garden, 1984; Lewitt, 1983). *Nuclear magnetic resonance imaging* or *magnetic resonance imaging* (to avoid negative connotations associated with the word 'nuclear') are further examples of imaging devices that do not sense the data in spatial coordinates. These devices measure the frequency spectrum of an electromagnetic field generated by a distribution of hydrogen and other atoms that have been excited into resonance by an intense time-varying magnetic field. The spatial cross-section of the body can be reconstructed from the frequency distribution by a technique known as direct Fourier inversion (Bracewell, 1986). The techniques that generate a spatial image from indirect measurements are collectively known as *image reconstruction* (Bates & McDonnell, 1986).

2.2.2 General image

A *general image* of a scene can be defined as a graphical representation of the spatial distribution of one or more important physical quantities. Most often, an image is a spatial distribution of light intensity as detected by a transducer sensitive to the appropriate wavelengths (Gonzalez & Wintz, 1987). However, general images are not limited recorded light intensities, but encompass other important physical quantities such as range (Jain & Jain, 1990; De Menthon *et al.*, 1987; Naylor, 1987), texture, surface shading, contours, (Aloimonos, 1988; Aloimonos & Swain, 1988; Aloimonos & Weiss, 1988; Marr, 1982), x-ray attenuation, radioisotope density (see above references for CT), ultrasound scatter (Lee & Wade, 1990), and spin-lattice relaxation time (in nuclear magnetic resonance) (Lauterbur, 1973). The physical quantities represented by general images are diverse and encompass less "image" oriented spatial quantities including: terrain (Hawke, 1989) and vegetation type (Smith *et al.*, 1989).

Nonuniform sampling and quantisation - in both amplitude and space - of an image capture system can distort the formation of a digital image. The removal of such distortion before processing is a step is often known as *decalibration* (Castleman, 1979). Barrel and pin-cushion distortions are examples of spatial degradations which occur when the lens of a camera has a different magnification at the centre of the field of view than at the periphery. Geometric models of these distortions can be used to design transformations for their correction. Low contrast in images, a form of intensity distortion, is often a result of improper settings for sensor gain or lens aperture at the time of image capture. Enhancement of low contrast images is performed by scaling the pixel values to span the entire intensity range. The scale factor can be calculated over the entire image (global method) (Hall, 1974); within a fixed sized neighbourhood (local method) (Hummel, 1977); or over a variable size neighbourhood, which is adjusted to meet some image statistic criterion (adaptive method) (Lesczczynski & Shalev, 1989).

2.2.3 Segmented image

Unser and Eden (1988) state, "The goal of image segmentation is to divide an image into regions that are uniform or homogeneous with respect to certain characteristics". Many authors agree with this definition (Ballard & Brown, 1982; Castleman, 1979; Gonzalez & Wintz, 1987; Fu & Mui, 1981; Pavlidis, 1986; Rosenfeld, 1988). The standard use of segmentation is to determine which pixels in an image belong to an object, and which belong to the background. The resultant segmented image usually has two values: one to denote the object and the other to denote background. In some cases, a segmented image may indicate several classes of objects. These images will have several labels, one for each class and one for the background.

The most simple form of segmentation is global *thresholding*. Every pixel in an image is compared to a prescribed intensity value called the threshold value. If an evaluated pixel is less than the threshold, then a black pixel (say) is generated; but if the evaluated pixel is greater than or equal to the threshold, then a white result pixel is generated. A threshold value may be chosen manually, where the user successively applies a range of values to an image and sees which yields the required segmentation. Alternatively, unsupervised methods exist to automatically select the threshold value. For images where the object's intensities are distinctly different from the background, the histogram of the intensities will be bi-modal. In this case, a threshold can be chosen as the intensity that corresponds to the valley of the histogram (Castleman, 1979). In images where the object intensities are not distinct from the background, an edge sharpening operation may be applied to deepen the valley so that it can be detected more easily. For images where the object area is known, segmentation can be achieved by varying the threshold value until the known area is attained.

Global thresholding methods perform poorly on scenes that have been non-uniformly illuminated. A superimposed intensity gradient across an image precludes the use of a single threshold value. Local thresholding techniques address this problem by dividing the original image into smaller images and finding thresholds for each of the subimages. Threshold values can be calculated for every point in an image or just once for each subimage. In the latter technique, discontinuities appear at the subimage boundaries, but these can be reduced by a smoothing operation (Sahoo *et al.*, 1988).

Segmentation of non-uniformly illuminated images can also be achieved by edge detection techniques, which identify a homogenous region by locating its boundary. The application of edge detection techniques for segmentation is based on the assumption that distinct steps in intensity occur at object boundaries. Common methods to detect edges include first and second order differential operations. Among these filters are the Sobel filter, the Robert's product (Gonzalez & Wintz, 1987), and the Laplacian operator (Hildreth, 1980).

The Laplacian operator has also been used as the first step in methods of segmentation based on texture. Techniques, like those of Perry (1989) and Catanzariti *et al.* (1989), derive texture measures from the local orientation and density of zero crossings in an image's second derivative. The implementations of these methods are still in the development stages, but they demonstrate the possibility of segmenting scenes that do not exhibit regions of uniform intensity.

2.2.4 Compact structures

The speed of processing labelled images depends very much on the structures used to represent the data. Compact structures can be often processed more rapidly than two dimensional binary images. The advantages of using compact structures are: increased speed of processing and lower memory requirement; both are desirable properties for an image processing system.

The choice of the data structure to be used depends on the aspect of an image to be represented. Sometimes only the outlines of objects are of interest; at other times not just outlines, but the grey value distributions within objects are required for subsequent processing. A convex hull generally has fewer vertices than its original region and has straight line edges between possible distant vertices in its original region, so it may be better to represent it by a list of vertices (Gonzalez & Wintz, 1987), rather than a Freeman code (Freeman, 1961). The number of control points in a vertex list can be reduced by using B-splines (Foley & Van Dam, 1982) to approximate the paths between points on an outline. Data structures that represent regions include quadtrees (Manohar *et al.*, 1990), distance coded medial axis transforms (Arcelli *et al.*, 1975; Arcelli & Sanniti di Baja, 1986), and interval lists (Rutovtiz, 1989).

The choice of data structure also depends on what processing can be directly performed using these structures. For instance, the area of a Freeman coded outline can be found by tracking around the boundary, adding x-coordinate values when moving upwards and subtracting them moving downwards. This technique must track outer boundaries in an anti-clockwise direction, and holes in a clockwise direction. A circumscribing circle can be efficiently found for a region represented as a list of vertices. Likewise, a convex hull can be found rapidly for a region expressed as a list of line intervals (Rutovtiz, 1989).

Tanimoto and Kent (1990) claim that most specialised hardware systems for image processing handle image-based transformations efficiently, but by contrast, handle transformation of images into compact structures inefficiently. They describe and compare several representative examples of architectures that attempt to improve, what they call, the "intermediate-level" process of transforming image data into compact data structures. The tradeoff between speed and generality of computation was indicated as an important design issue. Their conclusions suggest that this area of research is still at a formative stage but that the possibilities for the development of new and interesting methods are promising.

2.2.5 Shape measurement

The next data transformation in the algorithm sequence is the measurement of geometric or intensity features of the extracted objects. As indicated in Figure 2.2, measurements can be taken from either an image representation or a compact structure representation. For an image analysis application, the measurement procured at this stage is significant because it represents the end product of the analysis. In pattern recognition or machine vision applications, measurements describe key features about objects that can be later used in some decision process. For example, a spanner can be distinguished from a bolt by simply comparing their areas. In other cases, more than one number may be needed to uniquely characterise different objects. Sets of measurements that represent an object are called *feature vectors* (Castleman, 1979).

Typical scalar measurements include area, perimeter, minimum length, maximum length, minimum external circle, maximum internal circle, and ratios thereof. Despite their simplicity, scalar measurements can be effective for the identification of objects from a well defined set. Kruger and Thompson (1981) used six simple measurements and one derived measurement to good effect to identify car parts travelling on a conveyor in a robotic assembly system.

Besides measurements that describe the boundary shape of a region, descriptors also exist to characterise the grey distribution inside regions. The simplest of such measures is the sum of intensities within a region or what Castleman (1979) calls the *integrated optical density* (IOD):

$$IOD = \sum_x \sum_y f(x, y)$$

The intensity distribution of the image is denoted by $f(x, y)$.

The weighted average for the x and y positions, otherwise known as the *centroid*, can be found by applying the formulae:

$$\bar{x} = \frac{\sum_x \sum_y f(x, y)x}{IOD}$$

$$\bar{y} = \frac{\sum_x \sum_y f(x, y)y}{IOD}$$

Using the centroid, the first and n -th order moments are given by:

$$\mu_{11} = \sum_x \sum_y (x - \bar{x})(y - \bar{y})f(x, y)$$

$$\mu_n = \sum_x \sum_y (x - \bar{x})^p (y - \bar{y})^q f(x, y)$$

where n is the order of the moment $n = p + q$.

Numerous shape measurements are possible but it is beyond the scope of this discussion to provide an exhaustive catalogue of all techniques. However, Pavlidis (1978) and Marshall (1989) review the subject in detail.

2.2.6 Pattern classification

To take the analogy of a court proceeding; to classify an object is to deliver a verdict as to what the object is. The verdict is based on an examination of the evidence, which is provided in the form of measurements made in the preceding part of the algorithm.

A typical pattern classifier works by examining a number of feature measurements for each object. Each set of measurements is called a *feature vector* because it identifies the object at a specific location in an n -dimensional feature space or pattern space, where n is the number of measurements. When a large number of objects are measured and located in the pattern space, the distinct objects types tend to form clusters in this space. Given a pattern space occupied by clusters, discriminating functions can then be derived to classify any given feature vector. Therefore any new feature vector can be immediately classified as belonging to one of the known classes. The objects selected to form the initial clusters are called training samples. The more training samples, the more accurately the discriminant functions can be derived.

The computational cost of making a classification increases with each additional measurement. The classification is made in a multidimensional space where each measurement contributes one dimension. Hence a pattern space can occupy huge amounts of memory for even a modest number of measurements. It is therefore important to select a small but highly discriminatory set of measurements to keep the volume of pattern space to a minimum. The only guaranteed way of choosing the best set of measurements is to try all combinations, but of course this is computationally impractical. In practice, heuristics - like the seven techniques surveyed by Mucciardi and Gose (1971) - are used to decide the best subset of measurements for making a classification.

2.2.7 An example algorithm

Figure 2.3 illustrates the sequential operation-by-operation nature of a typical image processing algorithm. This VIPS (Bailey & Hodgson, 1988) algorithm finds the width of a blob wall at its narrowest point as shown in Figure 2.4. This is done by encoding all pixels inside the inner boundary of the blob with its distance to the outer boundary; the smallest value will be the minimum thickness of the blob wall. This algorithm represents a sequence of operations whose structure matches that given in Figure 2.2. Table 2.1 highlights this by showing how the variables in the algorithm correspond to the data types in the general algorithm.

```

declare image (64 64) in inner outer      ! Declare image
                                           ! variables
capture 1                                ! Capture input image into
                                           ! frame buffer
get in                                    ! Put image into 'in'

! Extract inner boundary
threshold in 128 255                      ! Segment blob from background
chain code in c                            ! Chain code blobs
chain sort c c_inner                       ! Keep only inner boundary
chain draw c_inner inner /fill             ! Reconstruct inner boundary

! Repeat for outer wall
threshold in 0 128
chain code in c
chain sort c c_outer 5
chain draw c_outer outer /fill

! Measure wall thickness
distance outer                            ! Distance code outer boundary
and outer inner                           ! Keep only distance codes
                                           ! inside holes
subtract outer 1                           ! Change background 0 -> 255
extreme outer thinnest tmp                 ! Extract minimum wall value
let thinnest = thinnest + 1                ! Replace 1 just taken off
write thinnest                             ! Write result

```

Figure 2.3: VIPS algorithm to find wall thickness at its narrowest point.

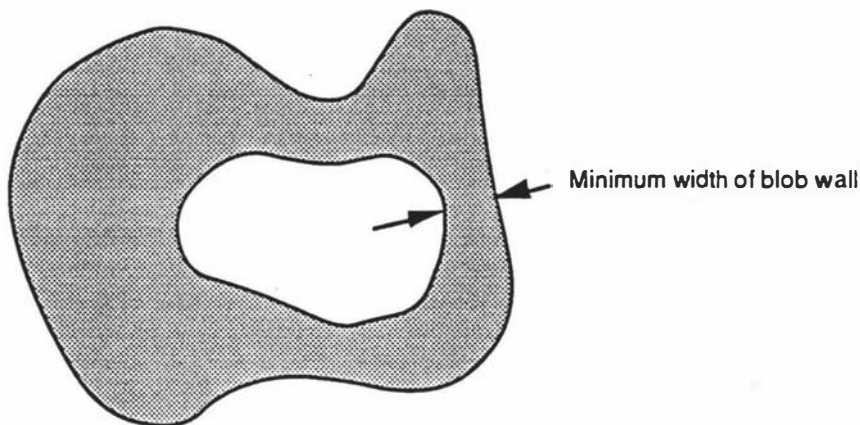


Figure 2.4: An arbitrary object with a hole.

VIPS Variable name in Figure 2.3	Corresponding data Type of Figure 2.2
in (image)	general image
inner, outer (image)	segmented image
c, c_outer, c_inner (chain code)	geometric structure
thickness (integer scalar)	feature measurement

Table 2.1: The match between VIPS variables in Figure 2.3 and the data types in the general algorithm structure.

2.3 Data-oriented view of algorithms

Figure 2.3 presents an algorithm as a sequence of operations, but the sequential view may not be an ideal representation for a user trying to develop an algorithm. The order in which the operations are shown is the order in which they are executed on a computer. This is a *process-oriented* view of an algorithm. In contrast, the user may focus on the changes made to data as it advances through the algorithm. This is a *data-oriented* view of the algorithm. In a process-oriented view, the order that the operations are executed is represented explicitly. In contrast, in a data-oriented view, the progress of data through the system is represented explicitly. To illustrate the contrast between the two views, the diagram shown in Figure 2.5 shows how the algorithm of Figure 2.3 was constructed. Each node in this network corresponds to a unique image, while the links represent imaging operations. No significance is attached to locations in this two dimensional space, so relative spatial positions do not indicate qualitative or quantitative information.

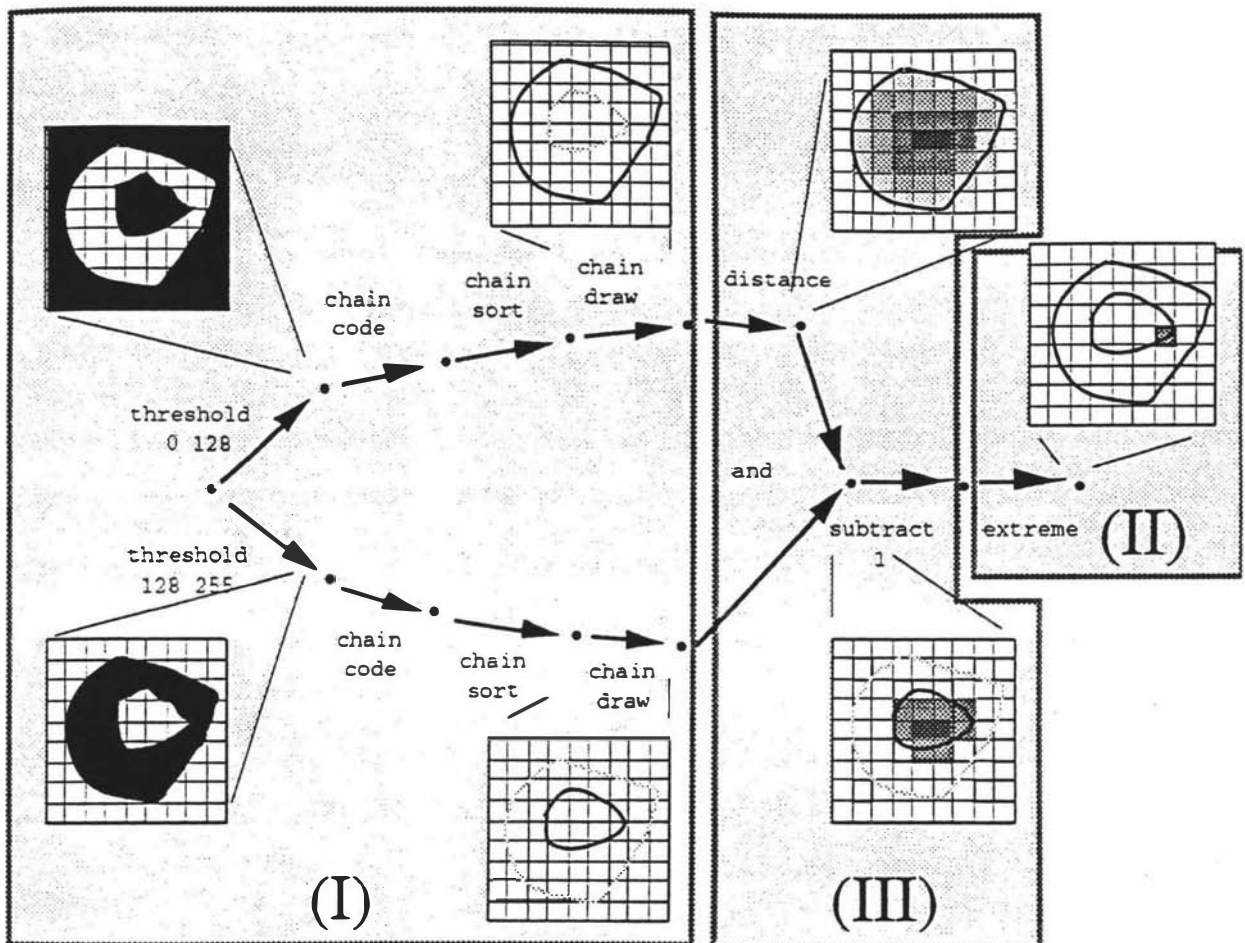


Figure 2.5: Solution network to describe the development of the algorithm in Figure 2.3.

The leftmost node of the network represents the digitised image of the outside scene. The algorithm designer experiments with this raw image and finds that the inner and outer boundaries of the blob can be isolated using various chain code

operations (as shown in Figure 2.5 (I)). The designer now strikes on the key idea of how to solve the measurement problem, which is to code every pixel inside the hole with its distance to the outer boundary. The minimum thickness of the wall is then simply the smallest coded distance (as shown in Figure 2.5 (II)). The remaining task is to distance code only those pixels inside the hole. The designer discovers that this can be done by first distance coding the pixels within the outer boundary of the blob and then setting all pixels to zero except those inside the hole (as shown in Figure 2.5 (III)). This final step links the first and last portions of the algorithm. This example illustrates that the development of an algorithm can be multi-threaded (with branching data paths) and non-sequential. Localised sections of the algorithm are developed sequentially but the global modules of the algorithm are not. Despite the non-sequential progression of the development, the final algorithm shown in Figure 2.3 may give the false impression that the development was sequential and incremental.

2.4 Algorithm development is problem solving

The development of an imaging algorithm is an example of a problem solving task. Polya (1962, p117) states "to have a problem means: to search consciously for some action appropriate to attain a clearly conceived, but not immediately attainable aim. To solve a problem means to find such action". The development of an algorithm can be considered a problem solving activity, where the aim is to find a set of operations that work together to produce a set of desired measurements. The solution to this problem is an algorithm. "An algorithm is a formula for a solution (i.e. a plan for the sequence of steps required in order to find a solution with certainty if one exists)", states Dorner (1983).

Humans commonly resort to one of several approaches to solve problems. One of the earliest studies into problem solving was a performed by Thorndike (1898) who investigated how cats learned to escape from a 'puzzle box' by trial-and-error learning. A hungry cat was placed inside a box and food was placed outside. A piece of string that dangled from the ceiling was connected to the door; the cat could open the door only by pulling the string. Thorndike timed how long it took for the cat to escape from the box. He observed that a cat managed to escape faster and faster the more times the experiment was performed. This demonstrated how experience of a particular way of solving a problem enhanced a subject's ability to solve that problem again. Another problem solving approach, lateral thinking (De Bono, 1969), attempts to break free from set kinds of thinking fostered by trial-and-error learning. The lateral thinking approach involves tackling a problem in a completely new way, as if nothing like it had been come across before.

While trial-and-error and lateral thinking are valid approaches to solve everyday problems, they are too general to easily assimilate into a design for a human-computer interface. What is needed is a new way of thinking about a problem - a new problem-solving paradigm.

A suitable paradigm is provided by Newell, Shaw, and Simon (1958). They have used this paradigm, called *means-ends analysis*, in the General Problem Solver (GPS) program, to solve cryptarithmic and chess problems (Newell and Simon, 1972). The means-ends analysis paradigm includes a representation for situations at the beginning and end of the solution, a representation for the difference between the two states, and a database of actions that could span the difference. GPS solves problems by selecting actions to remove the difference between the present and desired state of the objects or situations. Typically, the initial difference between the start and goal states are so distant that no single action can span this difference. GPS overcomes this by breaking the problem down into a number of smaller sub-goals, which serve to reduce the span of the maximum difference.

2.5 The solution graph

The states and processes involved in means-ends analysis can be graphically illustrated by a solution tree, like that shown in Figure 2.6. Dorner (1983) suggests that the construction of the solution graph is the cognitive activity involved in solving problems. Polya (1957, 1968) advocates the use of diagrams as a tool that systematically identifies the unknowns and the data of a problem. Polya's examples were mainly taken from the problem domain of solid geometry. One feature of Polya's approach is that the problem and the current state of the solution are graphically described on paper. Hence, the need to remember many facts is greatly lessened. The person is then freed to pursue the more creative activities of problem solving such as the use of analogy to associate, for the first time, two facts that are usually unrelated.

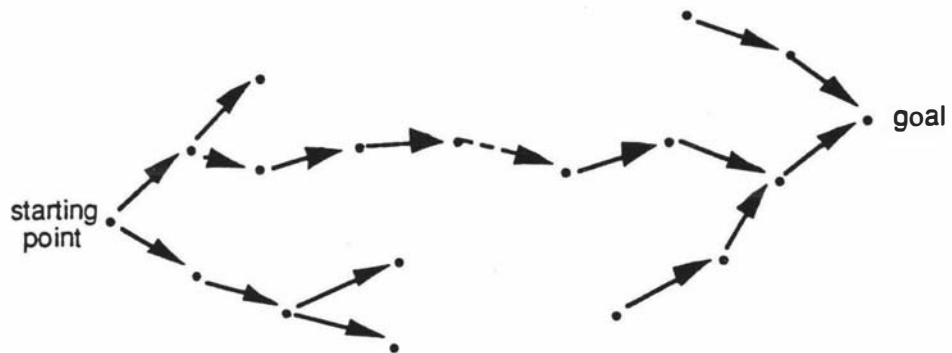


Figure 2.6: A solution graph through the 'field of reality'.

The space occupied by the graph in Figure 2.6 is what Dorner calls the *field of reality*. This space represents all the facts that can be understood, used and manipulated by the problem solver. Another name given to the space is *problem space* because it represents the subject's model of the task (Newell & Simon, 1972, p.59). For image processing, the problem space can be formulated so that each location represents a unique data set, whether it be image data or a chain code etc. In this problem space, imaging operations are represented as a line spanning

two points because they transform data. Taken literally, this space has many "locations". For example, a problem space involving images that have $N \times M$ 8-bit pixels, would have $256^{N \times M}$ locations to account for the image data alone; it would also have additional locations to account for vector and other types of data. In an interface, the solution graph is best used, not as tool to provide an exhaustive representation of all the possible results, but rather as a graphical tool to assist humans with the cognitive processes involved in problem solving.

2.6 Solution development

The use of the solution graph for algorithm development involves finding a network of operations that will transform the input data to the required output data. The naive approach would be to apply a brute force search where every possible combination of operations would be tried until the target results are attained. This, of course, is a ridiculous scenario. The computational demands of such an approach could be enormous; more significantly, it is a blind approach which does not take into account the problem solving ability of the user, nor the advantage that could be gained by partitioning the problem into a set of smaller, more manageable sub-goals. One method of performing means-ends analysis is to first specify a complete set of sub-goals to span the start-goal difference completely, and only then, attempt to specify the executable parts of the solution. In software engineering terms, this is known as *top-down decomposition*. However, the efficacy of this approach is tempered by the fact that if one sub-goal is unattainable, then the entire solution fails, as the overall success relies on the attainment of all the parts.

While the top-down decomposition approach seeks to introduce sub-goals that span the entire gap between the start and the goal, the method of *stepwise-refinement* looks only one step ahead. By this method, the solution graph is extended one node at a time. Each extension requires the algorithm developer to ask three questions:

- (i) Where to from here?
- (ii) How to get there?
- (iii) How good was that move?

Where to from here refers to the decision made concerning the location in the problem space that should be visited next. The problem solver must know the current solution state to make an informed and reasoned decision. If the current path seems to lead towards the goal, then the next step will be in a forward direction. If the path looks as if it will miss the goal, then the problem solver may abandon one or several recent steps to backtrack the solution to a point where it appears to be on the right track again. Figure 2.7 illustrates the abandonment of multiple steps in the solution graph.

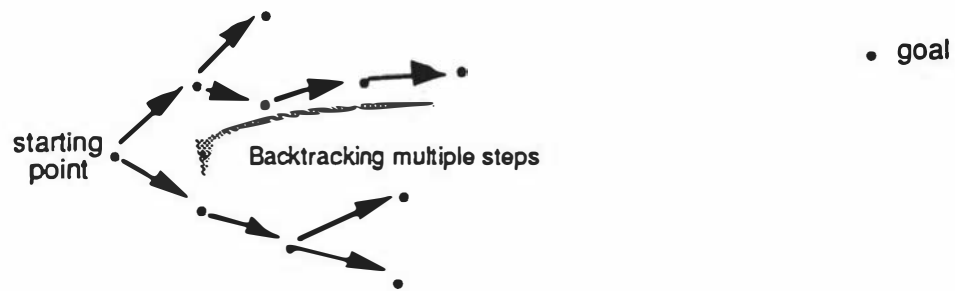


Figure 2.7: Abandonment of multiple steps.

To answer question the *where to from here*, one may choose from two objectives; that of computing the optimum or that of finding a satisfactory answer. In image processing, a classic example of computing the optimum is the use of linear programming to find a path between two known points (Unser *et al.*, 1988). This method is optimal in the sense that it minimises some prescribed cost function. Simon (1981) comments that in real life, a cost function is difficult to compute, except for when it is applied to trivial problems. Simon suggests that a practical alternative to computing the optimum is to look for good or satisfactory answers. This procedure which Simon calls *satisficing* allows designs to be rated as "better" or "worse" rather than "the best". Satisficing is a pragmatic method of searching for solutions to real life problems.

Once the problem solver has set a sub-goal, the attention is turned to *how to get there*. A typical system has a large range of operations, and *how to get there* is a matter of choosing the correct operation. In image processing, the choice of operation often involves guesswork. It is not that humans are unable to understand the theory of the operations involved, as they can all be described by a mathematical procedure. Rather, the difficulty of choosing an operation relates (i) to predicting the effect of the mathematical procedure on a large set of input data (a rectangular array of pixels), (ii) to the fact that more than a single operation may be needed to do the desired processing, and (iii) to the difficulty in determining whether or not an operation moves toward the target.

Once an operation has been executed and the result displayed, the problem solver must evaluate whether the target subgoal has been achieved. If the subgoal has been achieved, the problem solver must then assess whether the subgoal actually advances the solution towards the overall goal. A positive assessment usually means that the operation is kept as part of the algorithm, otherwise it is rejected and another operation is tried.

The stepwise refinement method has drawbacks; the most serious is that a solution can easily get side-tracked. In a local sense, the development of an algorithm may appear to progress smoothly with all its sub-goals easy to specify, all the required operations available, and its sub-goals easy to achieve. In a global sense however, the solution may lead nowhere near the overall goal. This can happen because the search procedure is local to every operation, and

because there is no global way to monitor the combined effect of these searches in the context of the overall solution development.

2.7 The heuristic approach

A *heuristic* is a plan for a sequence of steps that, when followed, increases the likelihood finding a solution (Dorner, 1983). The classic example of a heuristic is the advice given to beginners in chess: *always check, it may be mate* (Newell, 1983). Heuristics are used to limit the search for possible solutions by "suggesting plausible actions to follow or implausible ones to avoid" (Lenat, 1983).

In practice, imaging algorithms are not developed by an exhaustive examination of operation combinations. Experience teaches people efficient ways to attain certain results. This experience is embodied in heuristics which experts apply implicitly during an algorithm development session. Some of these heuristics are catalogued as follows.

2.7.1 The broad decomposition of the task

The general structure of an algorithm, which is discussed in Section 2.2, illustrates the way in which a task can be decomposed into a sequence of smaller steps. In effect, such a decomposition sub-divides the gap between the start and goal positions in the problem space by specifying a number of interim states. Simon (1981, Chapter 7) argues that a complex system can be simplified by decomposing it into a set of interacting subsystems, where each subsystem in turn can be further decomposed. Decomposition continues recursively until elementary systems are reached. Decomposition should be applied whenever a natural hierarchy exists in the problem domain, but avoided where such a hierarchy does not exist. This heuristic is central to Bailey's computer assisted approach to algorithm generation (Bailey, 1988).

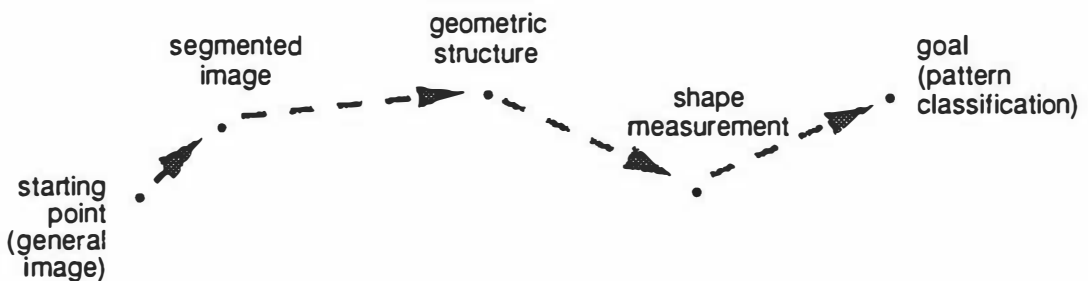


Figure 2.8: Interim goals are set by a broad decomposition of the task.

2.7.2 Identification of a critical subgoal

An algorithm designer may identify a critical subgoal that the solution must pass through in order for that particular approach to be successful. This is represented graphically in Figure 2.9. Here, the designer may give high priority

to the development of partial algorithms to reach this subgoal. If the critical subgoal cannot be achieved, then none of the auxiliary parts will be of any use. For example, if the task is to measure the area distribution of a collection of objects, then the critical subtask might be to separate any touching objects; this may be a non-trivial task if the objects are crowded together. In a sense, this heuristic is a special case of hierarchical decomposition.

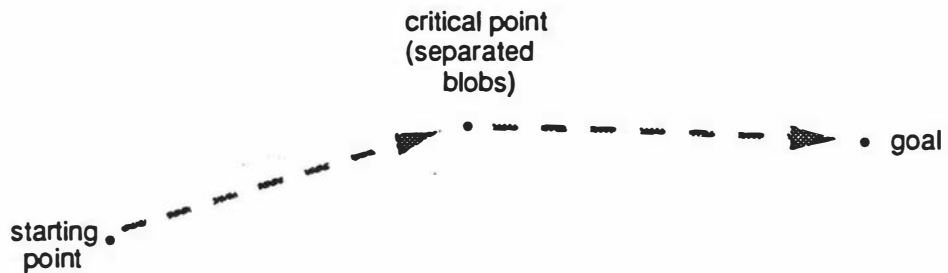


Figure 2.9: Identification of a critical subgoal that must be reached in order for the general approach to work.

2.7.3. Jumping to an arbitrary location

Synthesised or manually processed images can be used to jump the algorithm to an arbitrary location in the problem space. Typically an image is created that is close to a target goal or subgoal state. This simplifies the task of linking the small span, as shown in Figure 2.10.

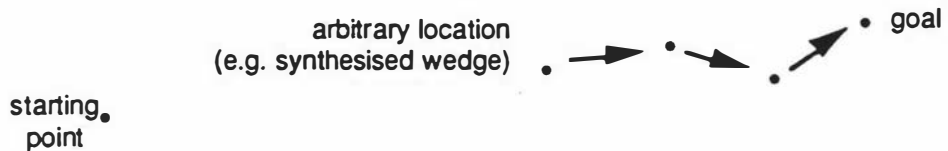


Figure 2.10: Synthesised data can be used to jump to a prescribed location.

2.7.4 Application of well known techniques

The designer may apply well known or robust techniques in order to span a portion of the problem space. A partial solution makes the search easier because the total distance is subdivided and the longest span is reduced. Figure 2.11 shows a robust technique for finding local maxima.

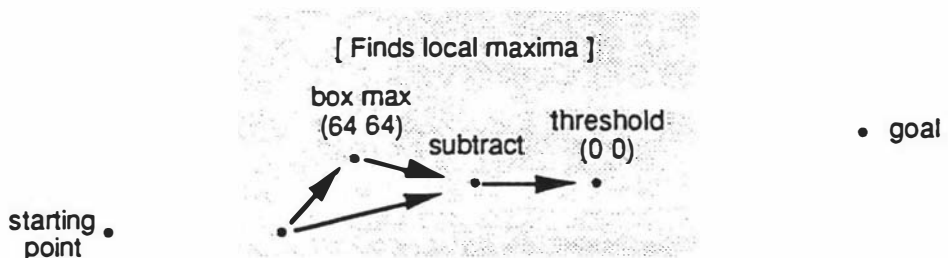


Figure 2.11: Well known techniques allow partial solutions to be prescribed.

2.7.5 Exemplar based development

In the development of an algorithm, a user may not always start with a "clean slate"; instead, a user may adapt an exemplary algorithm. The exemplar algorithm provides a strong starting point for development, especially if it almost performs the required imaging task. The ideal situation would be to use an exemplary algorithm that exactly matched the required imaging task; this could be used without alteration! Figure 2.12 illustrates the adaptation of a related algorithm in order to attain a new goal state.

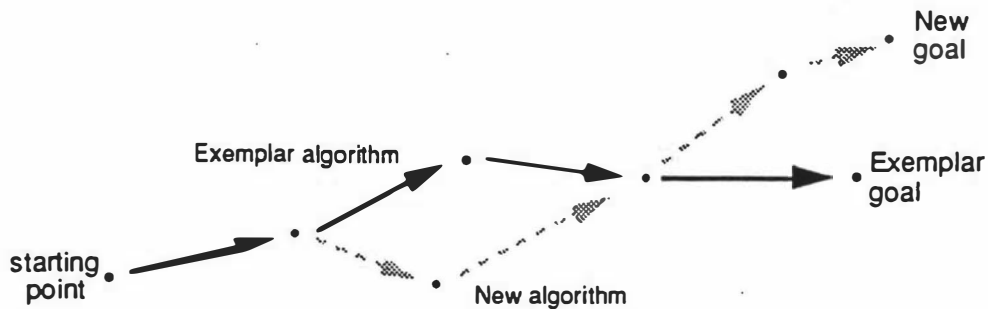


Figure 2.12: The development of an algorithm based on an exemplar algorithm.

2.7.6 Progressive refinement

The development of an algorithm often involves multiple passes. Once a simple algorithm is developed, it is often refined to make it more robust, accurate and efficient. Refinement is typically performed over several passes, where each refinement progressively improves the algorithm in one or more of the above respects. This heuristic is called *progressive refinement* and is presented last in this section because it can only be performed if a complete solution already exists. Figure 2.13 illustrates the refinement of a simple algorithm with a solution graph.

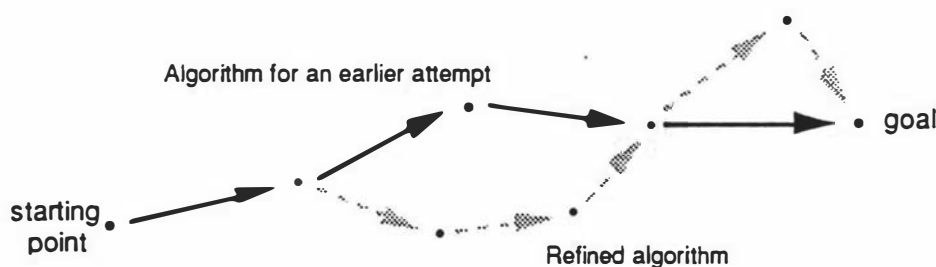


Figure 2.13: The progressive refinement of an algorithm.

The general rule for the production of a robust algorithm is to verify its success with many data sets. For instance, Price (1986) suggests even six "natural" images are unlikely to have the same obscure property that allows a fragile algorithm to work. Of course six is an arbitrary number, but the point made by Price is that an algorithm must be verified with many cases of sample data before it can be regarded as reliable.

2.8 Summary and Conclusions

An image processing system can be viewed as a sequence of operations working in succession to transform data through a series of distinct states. These states are: general image, segmented image, compact structure, measurements, and pattern class. Imaging applications are characterised by the forms of data they support: image enhancement applications deal only with general images; image analysis uses most data forms except the pattern class; machine vision applications typically involve all stages of the data transformation pipeline.

The pipeline view of an algorithm is exemplified in its presentation as a sequence of imaging operations. However, the process-oriented view is not the most natural way to represent the logic of the data transformation in the algorithm. In particular, such a view does not readily express multi-threaded data paths and interconnecting processing modules. A data-oriented view is a more natural way to describe the development of an algorithm.

The data-oriented view can be graphically represented as a solution graph. This device is commonly used in the field of problem solving to represent the construction of a solution. Indeed, imaging algorithm development is a form of problem solving activity, where the initial and goal states are defined by items of data, and the solution is a complete algorithm. Heuristics that are commonly used by algorithm designers include: decomposition of the task into subgoals, identification of a critical subgoal, jumping to arbitrary locations in the problem space, application of well known techniques, exemplar based development, and progressive refinement.

Two distinct views of image processing algorithms are presented in this chapter: process-oriented and data oriented. The process-oriented view is the appropriate representation to describe the execution of an algorithm on serial computers, because the order in which the operations appear in this view maps directly to the order in which they are executed. This view is expedient for algorithm execution, but not for development, because the sequence of operations in an algorithm is not necessarily the order in which the algorithm was developed. In practice, there is no particular order to the development of the constituent parts of an algorithm. Algorithms are developed by the application of heuristics which may effect any part of the algorithm. To handle this non-serial nature of algorithm development, the data-oriented view of an algorithm is suggested as being the more appropriate of the two representations discussed.

Chapter 3

Human Computer Interface Techniques

This chapter evaluates the application of conventional interface styles to the heuristic development of image processing algorithms. The styles discussed are: command line, menu based, direct manipulation, and visual languages.

3.1 Introduction

The essence of the image processing algorithm development task involves interaction between a human designer and an image processing algorithm, as shown by arrowed lines in the diagram of Figure 3.1.

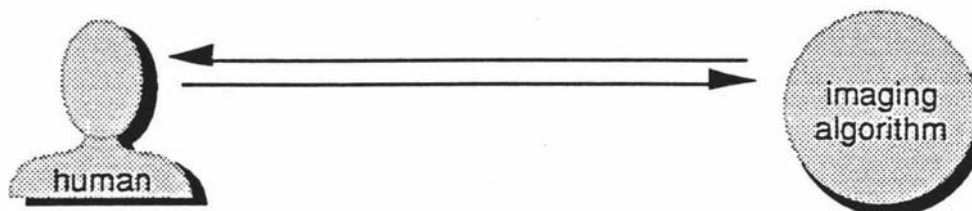


Figure 3.1: The interaction between a human designer and an imaging algorithm.

This chapter is concerned with how a computer can be best used to facilitate the algorithm development task. It is assumed that interaction between the human and the algorithm is made possible by a human-computer interface, as illustrated in Figure 3.2. Users' actions are directed towards the algorithm via the user interface. Changes in an algorithm are perceived by the user only through changes in the algorithm's representation in the interface. The user interface, depicted as a prism in Figure 3.2, not only facilitates interaction, but also shapes the interaction. An ideal interface facilitates the development of the algorithm without influencing the form of the final algorithm. In practice, ideal interfaces do not exist. It is inevitable that the user must perform actions that solely concern the interface and not the goal. For example, a help system assists the designer to use the interface but does not directly contribute to the construction of an algorithm.

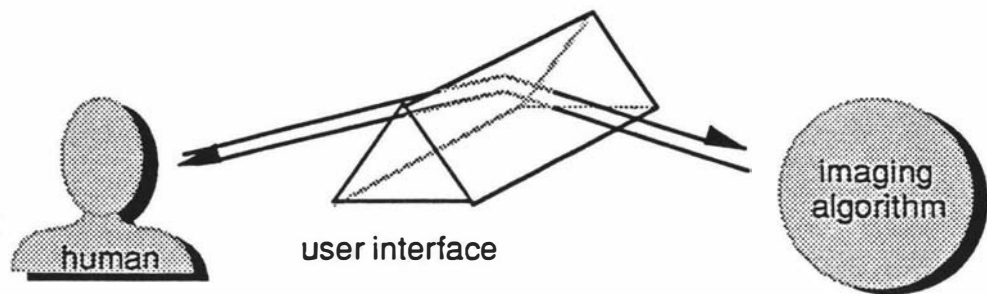


Figure 3.2: Human interaction mediated through a user interface.

Conventional styles of human computer interfaces include: command line based, menu based, direct manipulation, and visual language systems. Each of these approaches are examined for their ability to provide a transparent interface for the task of image processing algorithm development.

3.2 Command line interfaces

The command line interface is the oldest of all the interaction forms discussed in this chapter. In such an interface, the user types the name of an operation on a keyboard as command. This text string is passed to a command line parser which identifies and runs the selected operation. Associated parameters and options are typically included as part of the command syntax. The following are examples of command lines for a thresholding operation taken from VIPs (Bailey & Hodgson, 1988), Improc (Lane, 1988), ITEX 200 (Imaging Technology, 1989a), and Serendip (Wilson, 1987).

```
VIPS>      threshold im 128 255
improc>    threshold im_in im_out 128
1>         binarize( im_in, im_out, 128, 255 );
$-         let im_out = threshold( 128, 255 ) of im_in
```

These examples all use symbols to represent image variables. The command lines to display the results for the respective systems are:

```
VIPS>      display im
improc>    display im_out 0 0
l>         display( im_out );
$-         display im_out at (0, 0)
```

The interaction offered by command line systems became more direct with the improvement in computing power and interaction hardware. In the early days of digital computing, command lines were punched onto cards. The cards were gathered into a pile and submitted to the computer for execution. The time elapsed between the submission of cards and the return of results was measured in hundreds of seconds, although processing times of hours and days were not uncommon. Minimal interaction was afforded by these systems because of the long response time and high effort involved. Interaction improved with introduction of the teletype terminal as an input-output device, which cut the response time down to tens of seconds. Mistakes were less serious because programs could be altered and re-executed with little time penalty. With the development of faster computers and visual display units (VDU), response times were reduced to seconds. Jobs no longer needed to be submitted in batches, but instead could be executed in line-by-line units. Text based VDU's currently remain the predominant type of interaction device used in image processing systems. The following discussion on the advantages and disadvantages of command line systems assumes the use of a VDU terminal.

3.2.1 Advantages

Command line interfaces commonly have three features that assist the interactive development of image processing algorithms. The first and most important feature is the ability to combine groups of commands into a file and to execute this file with an instruction that could be contained on a single line. The command file is in essence an algorithm. Command macro features are usually supported in comprehensive command line systems, including VIPS, ITEX, Improc, and Serendip.

Another beneficial feature of command line systems is that the appearance of the interface does not change when new commands are added. When a new command is programmed and incorporated into the system, a user needs only to type the name of the new command to run it. This feature is significant because commands are frequently added to well used systems.

The third feature is the rapidity with which operations can be invoked. If a user is familiar with the syntax of the desired command, invocation is a simple matter of correctly typing the command line. This feature is of particular value in an image processing system where many operations are invoked, often in quick succession, during a development session.

Command line interfaces usually support a number of other features that speed up the specification of commands. One such feature is the provision of keyboard shortcuts to recall previous command lines. Most command-based operating

systems allow past commands to be recalled using the UP ARROW cursor key. Some command line parsers accept abbreviated command names. For example, VIPS allows the abbreviation `thr` to be entered instead of the full command `threshold`. VIPS also allows users to define aliases for part or full command lines, and so speeding up entry of commonly used commands. An alternative to command abbreviation is command completion. For example, in the UNIX 't-shell', the key sequence `cal` followed by the TAB key causes the operating system to complete the command string to give `calendar`.

3.2.2 Disadvantages

Symbolic references to data objects can be an inconvenience. To specify the parameters for a command, the user must remember the name of a particular image or the contents for a given variable. Any memory load imposed on the user is undesirable because it distracts the user from the real task of problem solving. The maintenance of correct associations between an image and its name is difficult because the contents of an image may often change during an interactive session. Symbolic references are often a problem when a user tries to recall the name of a displayed image. Recollections will be futile if the content of the image has been overwritten subsequent to its display. The assignment of meaningful names helps to reduce the problem. However, the creation of such names can be an onerous task in itself.

Another disadvantage of command line system is that a user can easily lose sight of the current state of the solution; this is epitomised by the question "where am I?" (Nievergelt & Weydert, 1980). It is easy to lose one's place in a development session because a standard text terminal gives a very limited view of the overall algorithm; typically 24 lines of text as a maximum. Furthermore, these 24 lines can be easily and irretrievably lost should the screen be filled with extra information like on-line help information. In a bid to recall the status of the development session, a user might list the data variables in current use, but this may be of limited use because of the difficulties with using variable names discussed above. Some systems, such as VIPS and Serendip, offer command log files to help alleviate this disadvantage. However, log files are of limited use because they cannot be inspected unless the user leaves the command line prompt.

Another drawback of command line interfaces stems from the inherent linear representation of an algorithm. As discussed in Section 2.3, a representation that displays multi-threaded data paths is a more helpful view for image processing algorithms.

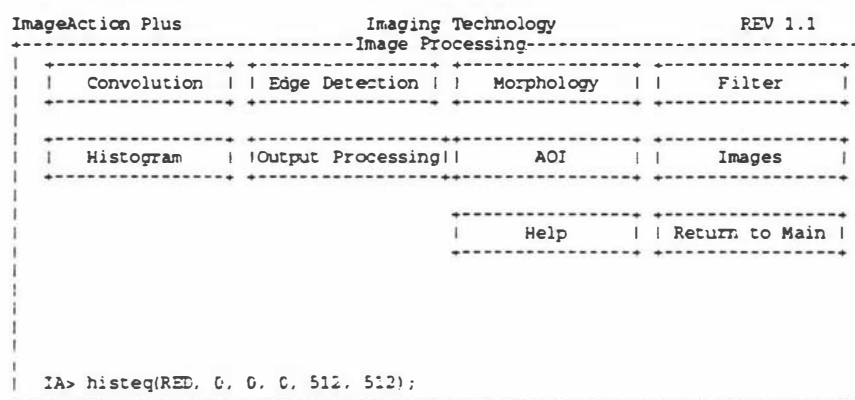
Command line systems are generally difficult to learn and training can take days or weeks. Systems may contain many commands and many variations of each command. This problem is particularly acute for image processing systems because comprehensive systems typically contain hundreds of operations. Shneiderman (1988) urges the use of meaningful, specific and distinctive command names to reduce training time. At the extreme, command line systems

can become an unwieldy assortment of commands; for example, Norman (1981) scorns the UNIX operating system for its inconsistent and cryptic naming conventions.









3.3 Menu based interface

In the context of computer systems, a menu evolved as an extension of command line systems. The essential difference between the two systems is in the way commands are selected. While a command line system presents a prompt which invites a user to type the name of a command, a menu system places commands in lists and invites the user to choose an item from the list. "A menu is a set of selectable representations of actions, parameters, objects (which may be other menus), states and other attributes." (Apperley & Spence, 1989). In a similar vein, Smith and Mosier (1986) define a menu as "a type of dialogue in which a user selects one item out of a list of displayed alternatives, whether the selection is by pointing, by entry of an associated option code, or by activation of an adjacent function key".

Figure 3.3 shows some examples of menus. The selectable items may take the form of buttons (a, d), pull-down lists (b, c), and pop-up lists (f). The menu items can be labelled with text-only (a, c, f), text and graphics (b, e), and graphics-only (d). Most menu structures comprise lists within lists. Some dialogues explicitly show the hierarchy tree of the menu (c, e). The ImageAction program (a) has the unique feature of including a command line field in the menu panel.



(a)

Opers		
	Simple	⇧⌘Z
<hr/>		
	Constant	⇧⌘C
	Match	⇧⌘M
	Persistent	⇧⌘P
	Instance	⇧⌘I
	Get	⇧⌘G
	Set	⇧⌘S
	Local	⇧⌘L

(b)

Image

Map ▶

Adjust ▶

Calculate ▶

Flip ▶

Rotate ▶

Image Size... ▶

Canvas Size...

Histogram...

...

Add...

Blend...

Composite...

Constant...

Darker...

Difference...

Duplicate...

Lighter...

Multiply...

Screen...


Subtract...

(c)

Tools



















































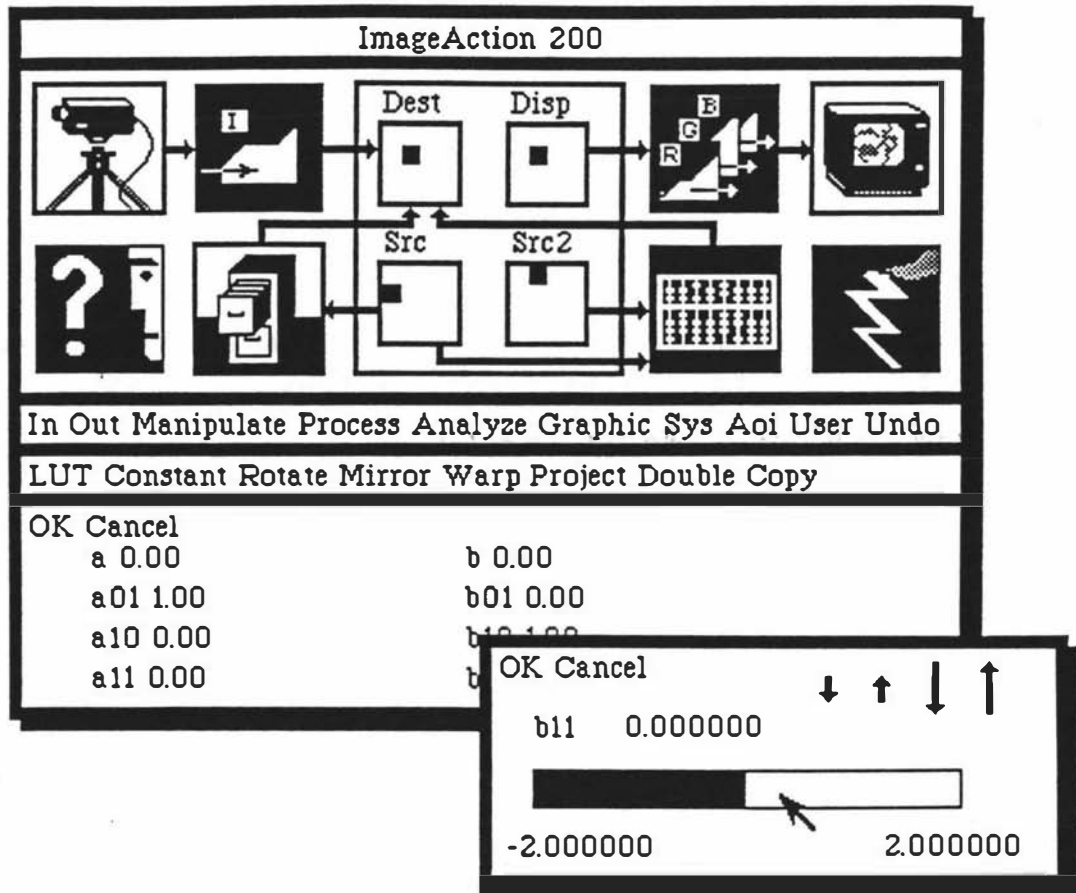




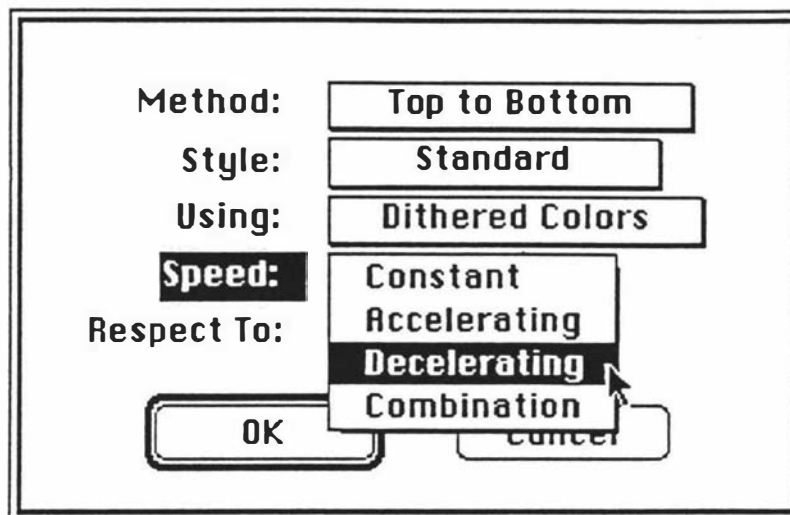




(d)



(e)



(f)

Figure 3.3: Examples of menus from different image processing systems: (a) ImageAction™ (Imaging Technology, 1987), (b) Prograph™ (Pietrzykowski & Matwin, 1984), (c) Photoshop™ (Adobe™, 1991), (d) Image (Rasband, 1992), (e) ImageAction200™ (Imaging Technology, 1989b), and (f) PixelPaint Pro™ (1989).

3.3.1 Advantages

Two aspects associated with the execution of a command are: (i) the search for a command to execute and (ii) the specification of this command for execution. In command line systems, they are distinct activities. In command line systems, a user learns of new commands by reading a user's manual or by consulting on-line help information. Once a command is found, the execution is carried out by typing its name, parameters and qualifiers at a command line prompt. In menu systems, the two aspects are integrated. Commands are found by browsing the menu lists. Once found, the command is selected and executed, usually by actions integrated with the browsing actions.

The degree to which these activities are integrated can lead to differences in the way an interface is used; these differences are especially pronounced in situations where the command set is largely unknown to the user. In a menu system, a user would probably be inclined to try new commands because of the little difference in time and effort needed to select a new command in preference to a familiar command. Whereas in a command line system, a user cannot use a new command without first learning that such a command exists, and this cannot be done without consulting a user's manual, help system or a colleague. To find a new command, the user's attention is usually turned away from the command line prompt and so away from the tool that is used to actually perform the problem solving. In contrast, a menu system entices a user to try a new command because all operations, new and familiar, are presented in lists.

3.3.2 Disadvantages

Although menus address one of the main weaknesses in command line languages, that of the difficulty in learning commands, they also sacrifice one of their main strengths for image processing, that of being able to create macro files. Without such a macro facility, menus essentially lack the capability to represent algorithms. This is a serious drawback if a system is to be used for algorithm development. Menu interfaces perform well for processing images in a one-off fashion, but perform poorly in situations that require sequences of operations to be repeated.

Another weakness of menu systems may be the difficulty in specifying parameters. Menu systems for image processing often process images from frame-buffer to frame-buffer (e.g. ImageActionplus™ (Imaging Technology, 1987) and Imagelab™ (1987)), and thus restricting the number and type of images that may be processed.

3.4 Direct manipulation interface

A *direct manipulation* interface (Shneiderman, 1983) is best described by example. The package used for the example is *Image*, which is a public domain software package for image analysis (Rasband, 1992). Given an image representing a set of nerve fibres in cross-section, the task is to measure a line profile of intensities through the image. In the *Image* package, which is shown in Figure 3.4, the image data is displayed in a window and the tool for plotting intensity profiles is represented as a button on a tool palette. To make a measurement, the user selects the 'plot' tool and draws a line on the image to indicate the path of the required profile. The program responds immediately by graphing the intensities of the specified line in an auxiliary window.

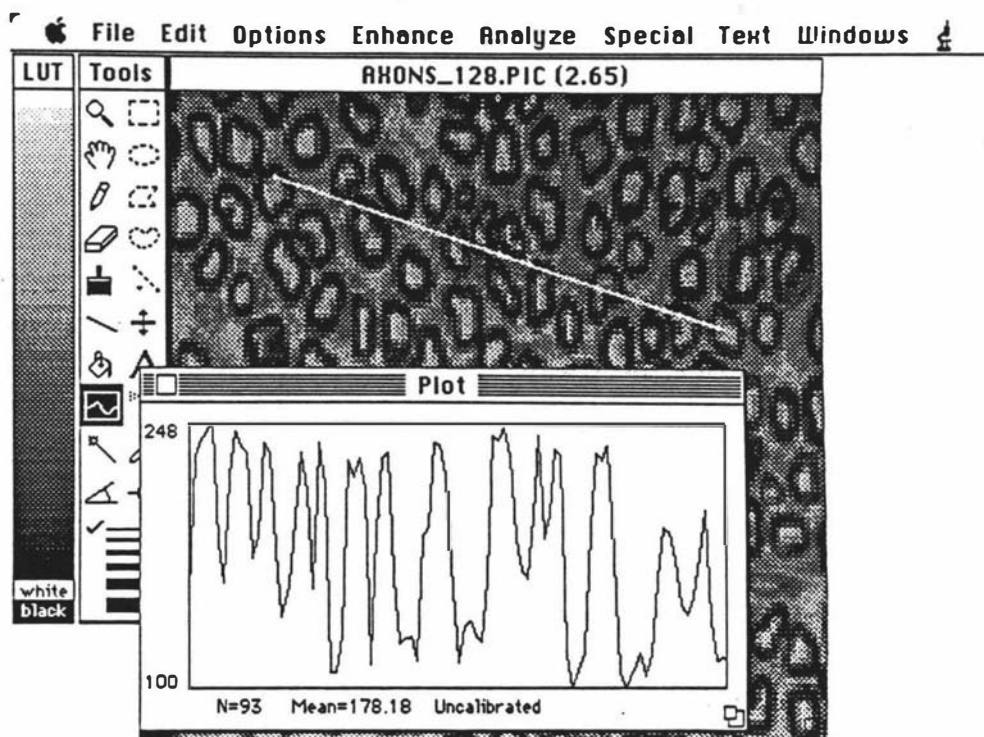
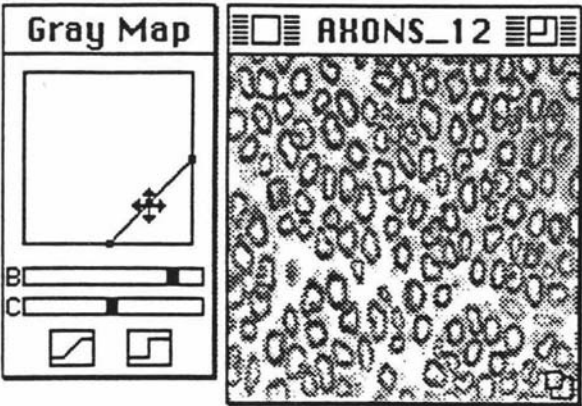
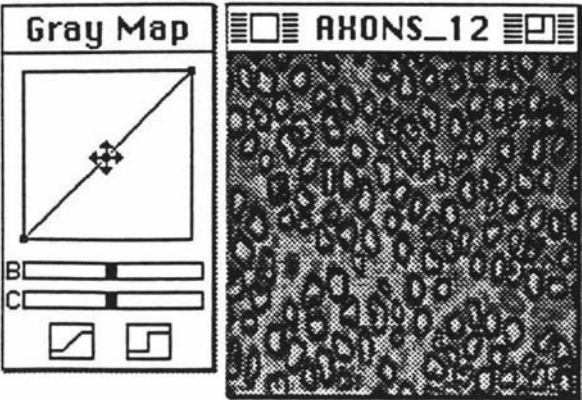


Figure 3.4: A example of a direct manipulation operation: the measurement of a line of intensities across an image.

Now suppose the user wants to uniformly increase the intensity values of the image. This can be performed in the *Image* program by adjusting the look-up table (LUT) function which is displayed in its own window, as shown in Figure 3.5. The user adjusts the offset of the function by placing the mouse controlled cursor over the function plot, depressing the mouse button, and moving the mouse. Any mouse movement will now adjust the function and simultaneously adjust the grey-scale values of the image of the top window. Figure 3.5 shows the LUT functions and the corresponding images for two different settings.



(a)



(b)

Figure 3.5: The intensity values of an image are continuously adjusted by moving the look-up table (LUT) function with the mouse.

Photoshop™ (Adobe, 1991) provides quite a different direct manipulation control to adjust the pixel intensities of an image. The 'brightness' and 'contrast' attributes of an image can be adjusted by the sliders shown in Figure 3.6. To make an adjustment, the user clicks onto the slider tab and moves it sideways. The brightness and contrast of the image respond instantly to changes in the control settings thus giving the user instant visual feedback to indicate the new values.

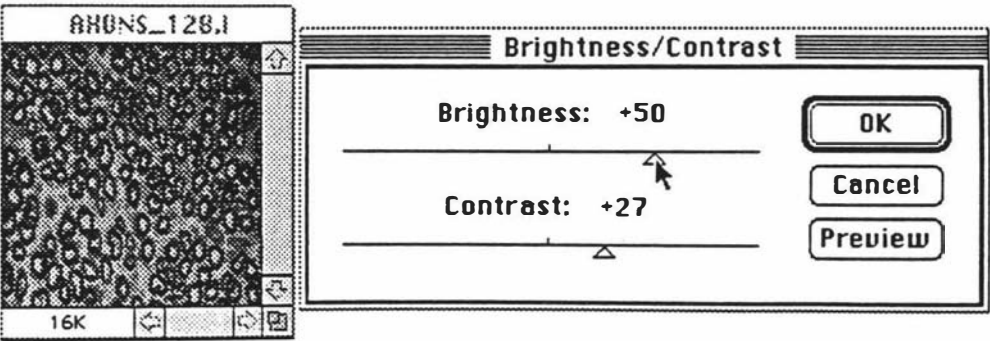
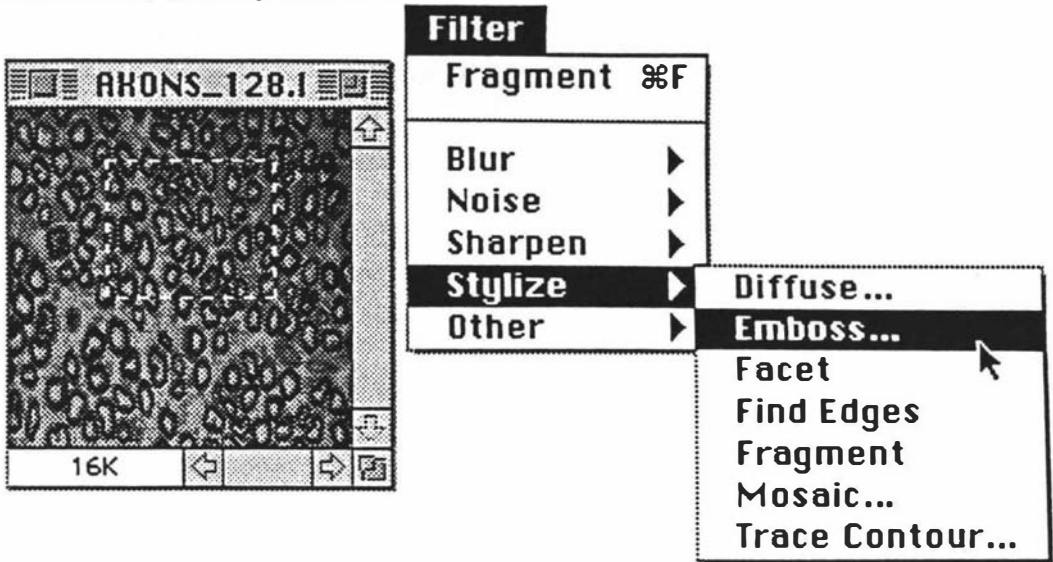
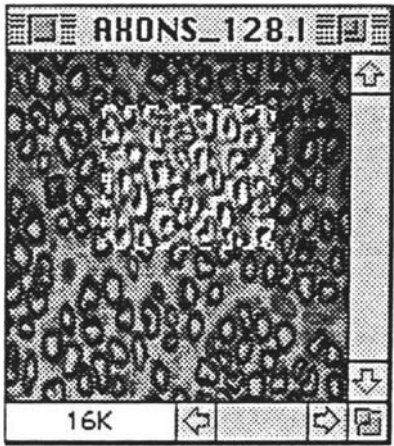


Figure 3.6: Image brightness and contrast can be continuously adjusted with slider controls.

Now, say the user wants to perform an *Emboss*⁴ operation on a rectangular sub-region of the image. To do this, the user first chooses the 'select tool' from the tool palette, and then draws a rectangle on the image (as shown in Figure 3.7) to indicate the sub-region. This is done by moving the cursor to one corner of the desired rectangle, depressing the mouse button and keeping it held down while the cursor is moved to the diagonally opposite corner. The rectangle is superimposed on the image to indicate the selected region. The user then chooses the 'Emboss' operation from a menu. Only the pixels within the selected region are affected by the operation.



(a)



(b)

Figure 3.7: The application of a filter operation to an image in Photoshop™ is performed in two steps: the selection of the area for processing and the selection of an operation, as shown in (a). The result of the Emboss operation in the selected sub-region of the image is shown in (b).

⁴ The Emboss filter makes a selection appear raised or stamped by suppressing the intensities within the selection and tracing its edges with black (Adobe,1991).

3.4.1 Key characteristics

This section briefly discusses the key characteristics of the direct manipulation interface and the applicability of these characteristics to image processing algorithm development. These characteristics are of benefit to most application domains, including image processing.

Metaphor. As a grammatical construct, the Concise Oxford Dictionary (Sykes, 1982) defines a metaphor as the application of a name or a descriptive term to an object to which it is imaginatively but not literally applicable. In the context of an interface, the term metaphor is used to describe the application of a graphically represented world to serve as the interface for a specific application to which it is imaginatively but not literally applicable. A metaphor draws upon the wealth of existing knowledge to provide information about a novel topic (Norman & Chin, 1989). The metaphor used in Image (in Figure 3.4) is of an artist's studio. The window in which the image is displayed is equivalent to the painter's easel and the tool palette is equivalent to the collection of tools a painter may have on a workbench.

Metaphors do not need to be taken from concrete examples in the outside world. If a metaphor is considered primarily as a device for organising the objects in an interface in a consistent and systematic way, then a metaphor need not be derived from a real world situation. A convincing reason for not choosing a real world situation for an interface metaphor is that some situations can imply meanings much richer than is intended. Command line systems and menu systems can be considered to use the metaphors of language and of the restaurant menu (Norman & Chin, 1989), respectively. In these systems, metaphor is used primarily to provide a systematic way to organise the interface, rather than a rich and accurate analogy to an outside world situation.

Direct engagement. Direct manipulation interfaces have point-and-click facilities to allow users to select and move items, icons, and objects on the screen; typing is necessary only to input text (Foley *et al.*, 1990, p.5). This characteristic of being able to directly handle interface elements as physical objects is called direct engagement.

The engagement is *direct*. For instance, to specify the path of the profile plot in Figure 3.4, a user draws a line directly onto the image. The endpoints are identified visually. In contrast, a command line method would require the line to be specified by two position vectors (e.g. (34 13) (124 93)). Here, the vector information is provided indirectly by textual description. Norman (1990) elucidates the concept of directness by referring to the analogy of first-person and third-person interaction. The difference between the two is like the difference between driving a car yourself or being driven by a chauffeur. When you drive the car yourself, you directly control the motion of the car by turning the steering wheel, pressing the accelerator, brake and clutch pedals, and shifting the gear lever. When driven by a chauffeur, you describe where you want to go but you do not manipulate the controls.

Continuous and reversible actions. Continuous and reversible actions are demonstrated in the adjustment of the look-up tables in Figure 3.5. Here, each control can be continuously varied. The appearance of the image changes in immediate response to the smallest mouse movement. Continuous control helps to create a direct relationship between the mouse movement and the result. In addition to being continuous, the actions performed are reversible. Hence results for any setting can be regenerated at any time provided the same control value is specified. Reversible actions allow mistakes in parameter settings to be made without the penalty of losing data. In many cases, an action can be performed without prior contemplation of the result; this is the essence of exploration. A common way to reverse an action is through an UNDO facility, which allows the results of an action to be retracted.

Exploration. The combination of all three characteristics of the interface mentioned so far leads to the secondary characteristic of exploration. Norman (1990) stresses the importance of exploration in the learning and use of a system. It will be recalled in the previous chapter, that the development of an imaging algorithm was described as a search process. Hence, the potential for exploration is important aspect of any system used for algorithm development. Norman says a system requires three properties to make it suitable for exploration.

- (i) *The user must be able to readily see and perform the allowable actions.* Direct manipulation interfaces satisfy this requirement by the provision of a metaphorical representation of the objects and a facility for direct engagement.
- (ii) *The effort of the action must be both visible and easy to interpret.* Direct manipulation fulfils these requirements again through the use of metaphor and direct engagement.
- (iii) *Actions should be readily reversible.*

3.4.2 Disadvantages

In direct manipulation interfaces, a simple re-enaction of a sequence of user actions may not necessary replicate a sequence of processing. The effect of the replayed user actions depends very much on the state of the interface. Take the example where the sequence of user actions that led to the application of the Emboss operation (shown in Figure 3.7) is replayed, but the window containing the image has been moved since the original episode. In this re-enactment, the mouse drag action used to select a region of image would occur in an undefined region of the screen, and so the sequence would fail in its intended effect. The lack of macro facilities in direct manipulation systems means that systems like Image and Photoshop™ are not useful for the development of algorithms. However, they still have their place for the one-off processing of images.

The second drawback of direct manipulation relates to operations that process multiple items of data. In the Macintosh interface, the paradigm for performing

an operation involves selecting the object to be processed, then selecting the operation. However, the identification of input data is not straightforward when an operation requires more than one input. The Subtract operation in Photoshop illustrates this difficulty. The selection of the Subtract menu item invokes a dialogue box in which the user must identify input and result images, as shown in Figure 3.8). Calculation is carried out after these inputs and outputs have been identified. The required engagement is cumbersome and represents very indirect engagement. The difficulty illustrated in this example is not peculiar to the Macintosh interface, but relates to all interfaces that use a sequence of direct engagement gestures to specify the input and output items.

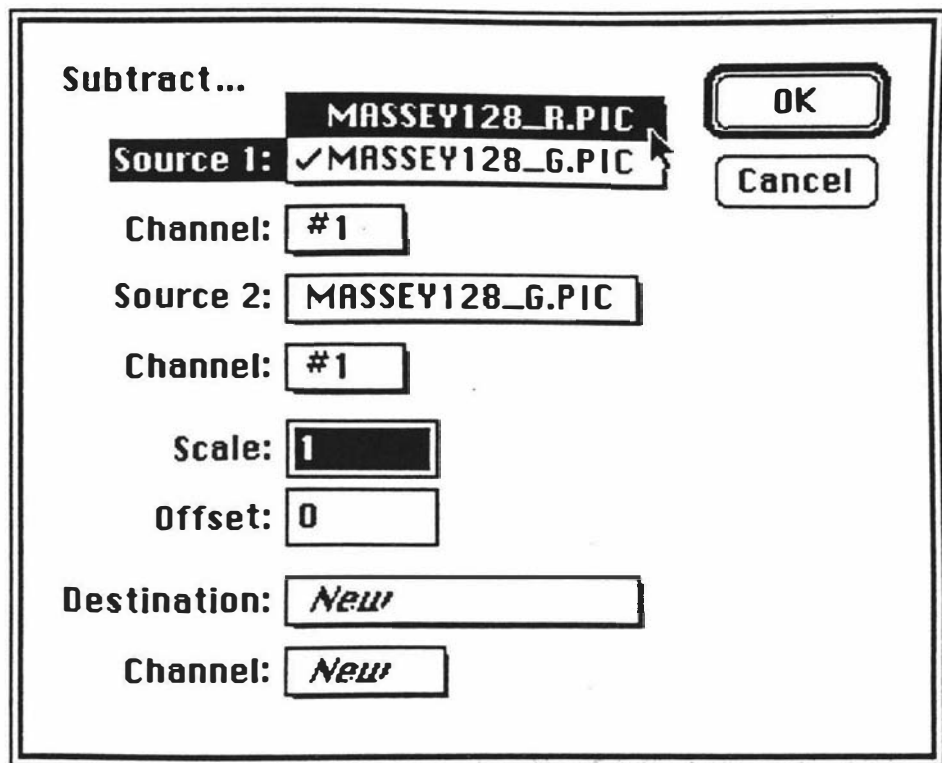


Figure 3.8: Photoshop™ menu to specify the two input images and the result image for the Subtract operation.

3.5 Visual language interface

3.5.1 Visual programming vs program visualisation

Of the interaction styles reviewed in this chapter, the visual language style is the most recent. It combines the linguistic properties of a command language and the visual properties of a direct manipulation system. In general, a program is series of coded instructions to control the operation of a computer. The set of instructions directly or indirectly understood by the computer is a *programming language*. If the instructions are encoded by pictures then the language is said to be a *visual programming language*. These are not to be confused with languages for the manipulation and query of pictorial data; such languages are often text

based. The image processing systems referred to in this section are visual with regard to both language and data.

One of the prime motivations for the development of visual programming languages is to make the programming task easier for users (Myers, 1990). Programming is necessary when the required function is not available on a given system. Systems that allow the end user to modify and change the existing working environment still only offer the possibility of a recombination of the available tools. In many cases, a recombination may not be enough to address the user's actual problem.

HI-VISUAL was one of the first reported visual programming languages for image processing (Monden *et al.*, 1984). Both operations and data are represented by icons. In a HI-VISUAL algorithm, the data icons and imaging operator icons are interleaved; an operation always produces a data object, and only data objects are accepted by operations. Figure 3.9 shows a HI-VISUAL algorithm that detects cracks in an input image. First, the video image is binarized by the operation BINARIZE. The cracks and edges of the objects are then detected with the CRACK DETECT and EDGE DETECT operations, respectively. These outputs are then superimposed with the SYNTHESIZE operation to produce an image that shows the spatial relationship between the cracks and the objects.

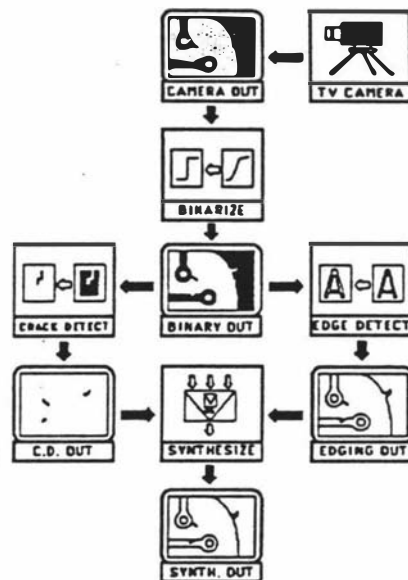


Figure 3.9: An example HI-VISUAL algorithm (from Shu, 1988).

Later versions of HI-VISUAL (Hirakawa *et al.*, 1990; Ichikawa & Hirakawa, 1990) have the distinction of using icons to represent only the objects supported by the system; icons to represent functions are not provided. Operations are specified by moving one object icon over another. The specific operation is inferred from the context of the two data objects. For example, when a 'paper' object - which supports the actions of 'edit', 'print', or 'copy' - is moved onto a 'pen' object, the 'edit' function of the 'paper' object is invoked. This model alleviates the need to design icons to represent functions; this is an advantage as far as software design is concerned as such functions tend to not lend themselves to visual representation.

3.5.2 Visually transformed vs naturally visual

Ambler and Burnett (1989) classify visual languages into one of two types. The first type, *visually transformed languages*, "include those visual languages that are inherently non-visual but have superimposed visual representations". These are visually edited traditional languages. PICT (Glinert & Tanimoto, 1984), SunPict, and C² (Glinert *et al.*, 1990) are examples of these types of visual languages. The second type, *naturally visual languages*, are languages whose expression is inherently visual. These may not have any textual equivalent.

Cantata (Rasure & Williams, 1991; Rasure *et al.*, 1990) is representative of a visually transformed language for image processing. Graphics are used both for the editing and display of the algorithm. When an algorithm is run, a textual equivalent is generated and submitted for execution as a UNIX command. The text and graphical equivalents are shown in separate windows in Figure 3.10.

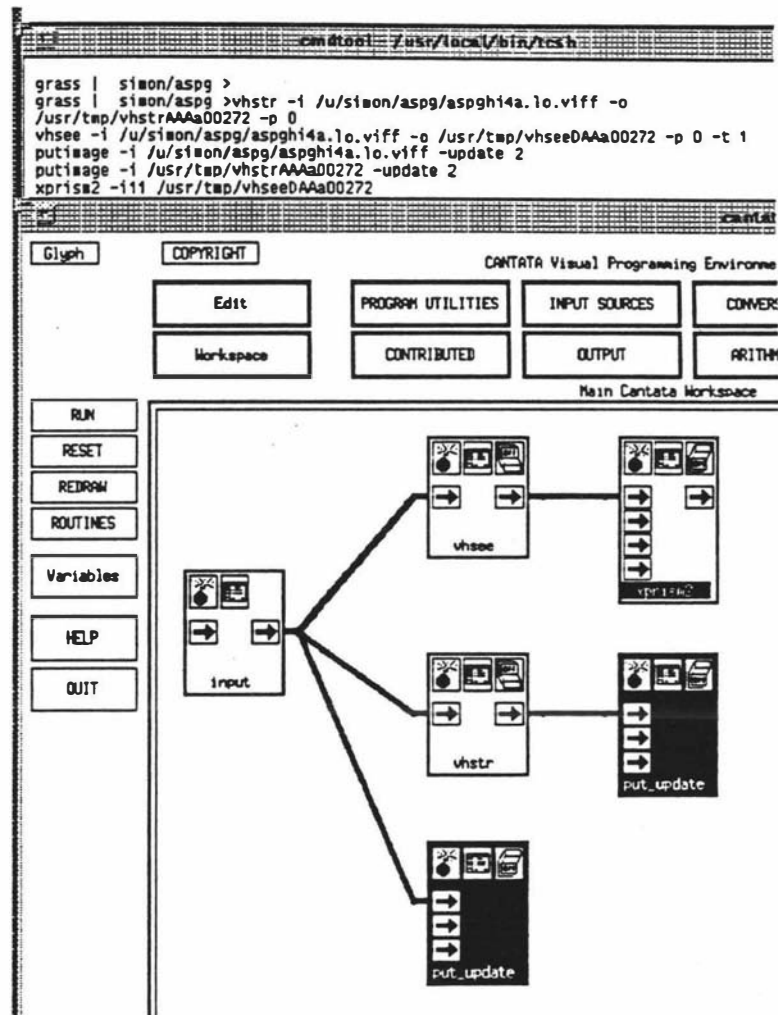


Figure 3.10: Data flow and command line equivalents in Cantata.

Duplicated data streams are difficult to represent in an inherently non-visual language. A limited form of data stream duplication is achieved in the UNIX operating system with the `tee` command. For example, the UNIX command:

```
> ls -l | tee theDir
```


lists the current directory onto the screen and into a file (`theDir`). However, the `tee` command allows the duplicated data stream to flow only into files. What is really required for true data flow processing is a 'real' tee (Davis, 1990), which would allow true duplicate streams, as shown by Figure 3.11.

```

                module_a2 | module_a3 > outfile_a
                |
module_1 < infile | real_tee
                |
                module_b2 | module_b3 > outfile_b

```

Figure 3.11: Possible implementation of a real_tee operator.

The `real_tee` construct, although needed to support duplicated data streams, is difficult to express in an inherently non-visual language. Therefore one of the primary motives for the development of naturally visual languages is to create an interface that supports duplicated data streams. One such language, the `ikp` image processing system (Davis, 1990), was developed to overcome this precise difficulty. This system uses an iconic data flow diagram as a basis for its interface language.

Although Cantata supports a visual representation of an algorithm, it is classed as visually transformed language because its algorithms are ultimately transformed to sequences of single-function programs which are executed from a UNIX command line, as shown in Figure 3.10. The translation of the algorithms into an intermediate language incurs a considerable processing overhead. This overhead is particularly severe in Cantata because all intermediate results are buffered in disk files, which makes processing extremely slow. A desirable alternative is to execute graphical representations directly. Here, a graphical representation serves not merely as a front end to a command line system, but represents the complete specification of the program. Such a language would fall into the class of naturally visual languages because they are inherently visual. The data flow representation of algorithms coupled with fast execution speed would facilitate the heuristic approach to algorithm development.

3.5.3 Program responsiveness

Tanimoto (1990) proposes four levels of "liveness" for visual programming systems. At the first level, the visual representation serves only as visual documentation for a program. At the second level, the visual representation is the specification of the program and can be run. Most visual languages for image processing fall into this category, including the Cantata and `ikp` systems. At the third level, the program responds to any edits made to the program and re-executes immediately to reflect the changes. At the fourth and most "lively" level, the program accepts a stream of data and executes for every new piece of data. These concepts are embodied in the proposed VIVA system (which stands for "Visualization of Vision Algorithms") whose primary objective is to provide a learning environment that supports the construction and understanding of algorithms.

3.5.4 Advantages

The persistent representation of an algorithm as a graphical picture is useful in image processing because it facilitates the incremental development of algorithms. The term incremental means that an algorithm can be built one step at a time, and that each step can be performed without the re-executing the preceding steps. Incremental development allows a user to concentrate on the part of the algorithm that is to be extended. This allows a user to focus solely on the problem solving task.

Visual languages incorporate desirable features of command line systems and direct manipulation systems, which are: a language facility and a direct engagement facility. A language facility provides a graphical means to represent a sequence or network of operations. Furthermore, this representation can be a recorded and re-executed at a later time. A direct engagement facility provides a powerful means to compose and edit the graphically represented language.

The pictorial nature of a visual language makes possible the representation of parallel data flows. Parallel data flows occur frequently in image processing algorithms. For example, the independent processing of the red-green-blue channels of a colour image could be represented as three parallel data flows. Such a representation would not only show the independent, but also the parallel, structure of an algorithm. More importantly, it would emphasise the fact that each flow contributes to the processing of a final composite RGB image.

3.5.5 Disadvantages

Control structures, such as loops and conditional statements, are generally difficult to incorporate into data flow languages without compromising the simplicity of the data flow form. However, these structures are routinely used in image processing, especially in algorithms that perform analysis procedures for every extracted object in an image. The Prograph™ system addresses the issue of control structures by offering special classes of operations for handling iteration and lists. For example, Figure 3.12(a) depicts an operation that sounds the system beep ten times; Figure 3.12(b) depicts a sequence that accumulates the values in a list and displays the running total.

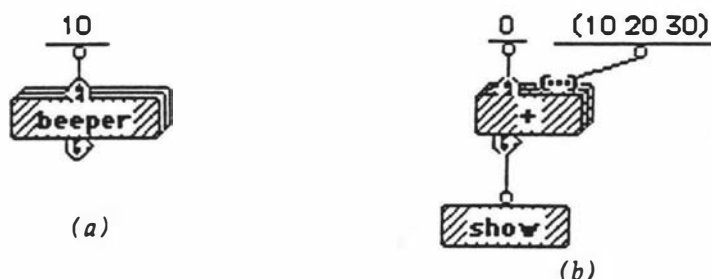


Figure 3.12: Examples of control operations from Prograph: (a) beeps 10 times, and (b) accumulates values in a list and displays the running sums.

A second disadvantage is that visual languages require a large screen space in which to represent algorithms. In textually represented algorithms, each command occupies a single line and these lines are packed tightly together on a display screen. By contrast, in a visually represented algorithm, each graphical element occupies a portion of screen space and these elements are typically distributed sparsely over the display screen to prevent the program from appearing cluttered.

Although graphical languages have the potential to better utilise a two-dimensional display than a textual language, graphical programs are typically longer in one direction. Visual programs often appear elongated because they are often composed of many stages but few parallel flows.

The requirement for a large screen is seen as a disadvantage primarily because of its high cost relative to the other components of an imaging system. Large displays, especially large colour displays, and their accompanying high-bandwidth video driver circuitry tend to be expensive.

3.6 Summary and conclusions

The human-computer interface mediates the interaction between a human designer and an imaging algorithm. Inevitably, the interface changes the perceived nature of the task, but a well-designed interface should minimise this imposition. An interface should ideally be transparent to the user.

No single approach meets all the needs of an environment for the heuristic development of imaging algorithms. It would appear that an interface that combined the features of menu, direct manipulation, and visual language styles could achieve the desired transparency. Menus allow rapid operations to be searched rapidly and entices the user to try new operations, which is a feature conducive to heuristic development. Direct manipulation style provides direct engagement facilities for the rapid execution of operations, and continuous and reversible actions that invite the user to explore the use of available operations. A key aspect of direct manipulation is the use of metaphor as a device to consistently and systematically organise the elements of the interface. The data flow is a widely used metaphor for image processing because it allows multi-threaded data paths to be represented explicitly. Upon examining a visually transformed data flow languages for image processing (Cantata), it would seem that an inherently visual language would provide the desired data flow representation without compromising execution efficiency. The following table summarises the advantages and disadvantages for each interface style with respect to image processing algorithm development.

Interface Style	Advantages	Disadvantages
Command Line	<ul style="list-style-type: none"> Algorithms can be represented in a textual language. Appearance of the interface does not change when new commands are added. Commands can be executed quickly provided the syntax is known. 	<ul style="list-style-type: none"> Users are required to remember associations between the names and contents of variables. Linear text does not give a 'big picture' of the current state of the solution. Linear text does not explicitly represent multi-threaded data paths. New commands must be learnt away from the command line parser: usually through the use of on-line help or hardcopy documentation.
Menu	<ul style="list-style-type: none"> Entices the user to try new commands because the user actions to search and execute a command are integrated. The use of a system does not require a comprehensive knowledge of the command set. 	<ul style="list-style-type: none"> Difficult to represent algorithms due to lack of a high-level language. Frame-buffer based systems restrict the number and type of images that can be processed.
Direct Manipulation	<ul style="list-style-type: none"> Consistent and systematic organisation of interface through the use of metaphor. Direct engagement enables quick and direct actions. Exploration is encouraged because mistakes can be readily reversed. 	<ul style="list-style-type: none"> Lack of a high-level language to represent algorithms. Difficult to specify multiple inputs or outputs or both for an operation.
Visual Language	<ul style="list-style-type: none"> Inherits all the advantages listed above for a direct manipulation interface. Persistent representation of program facilitates incremental development of algorithms. Algorithms can be represented visually. 	<ul style="list-style-type: none"> Awkward to incorporate control structures, such as loops and conditional facilities. Representation of complex algorithms requires large display screens.

Table 3.1: Comparison of the advantages and disadvantages of the interface styles discussed in this chapter.

This chapter has evaluated current human computer interface styles when applied to the application of heuristic development of image processing algorithms. The insights brought to light in this chapter are drawn together in a single coherent design, which is presented in the following chapter.

Chapter 4

An Interface Design

This chapter presents a design of an interface that facilitates the development of image processing algorithms. The first section looks at the philosophy adopted for the design. This is followed by a brief discussion on two modes of problem solving: stepwise refinement and dynamic exploration. The final sections discuss how these modes of investigation can be incorporated into a common interface using a visual language.

4.1 Design philosophy

As mentioned in the previous chapter, an ideal interface is one that transparently mediates the interactions between the user and the algorithm. Norman's (1990) philosophy of *user-centred design* aspires to this ideal. User-centred design, as the name suggests, focuses on the needs and interests of the user, which are taken into account in the design of the interface. The objective of this approach is to produce a useable and understandable product. This thesis adopts Norman's approach as its underlying design philosophy.

User-centred design is explained with the diagram of Figure 4.1. This diagram shows the relationship between: the *user* of the system, the *designer* of the system, and the *program*. User centred-design prescribes that the user's

perception of the task should be taken into account by the designer when a program is written. The arrows of Figure 4.1 indicate that the designer should understand the user's perception of a task domain, and that the designer should write the program in such a way to make the user's task easier to perform. In an ideal user-centred design, the user's model, the design model, and the appearance of the program are equivalent, moreover the design should originate at the user.

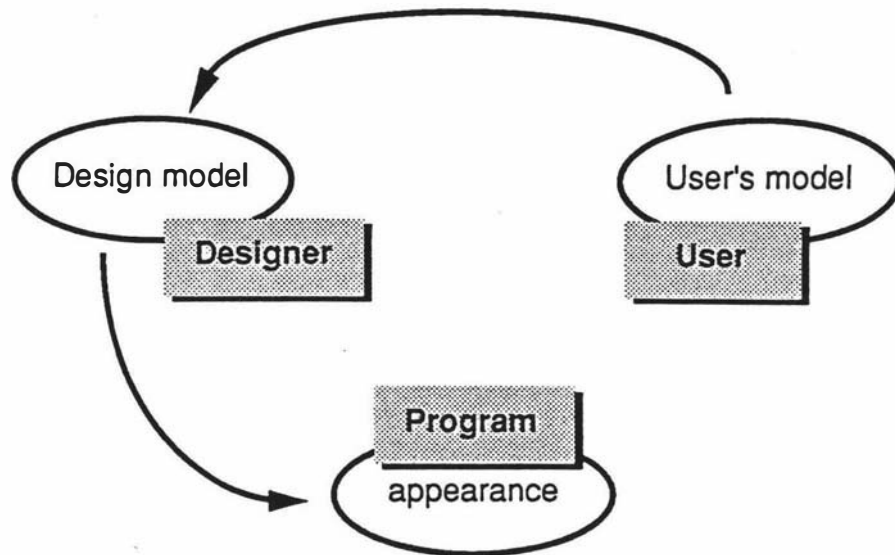


Figure 4.1: Three aspects of user-oriented design (Norman, 1990).

The user's model adopted in this chapter is based on the solution graph representation of an algorithm discussed in Section 2.5 (reproduced in Figure 4.2). It is recognised that different users may have different perceptions of a task. However, the solution graph model has been adopted because it is easy to understand and is appropriate for the task, as discussed in Section 2.3. With this user's model, an algorithm is defined by the topology of the solution graph.

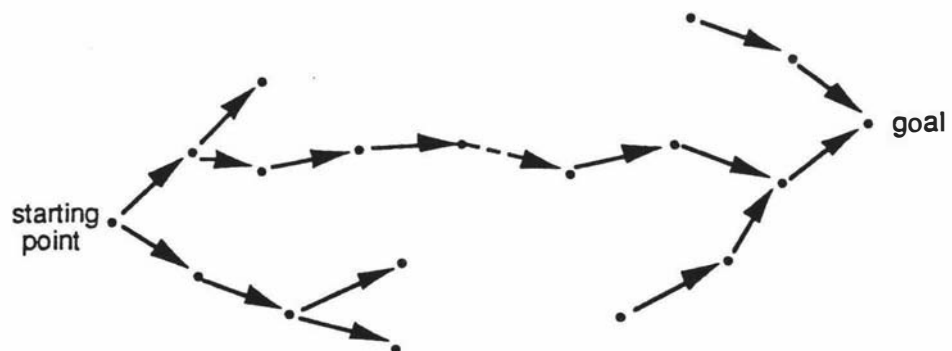


Figure 4.2: A solution graph.

Other problem solving methods use or have used diagrammatic representation of problems to aid solution development; notably those of Polya (1957, 1968) and Newell and Simon (1972). Polya's problem solving methods, which he applied mainly to solid geometry problems, uses solution graphs extensively to represent

the knowns and unknowns of a problem. Solution graphs promoted by Polya presented the entire current state of the solution at a glance. Newell and Simon used a solution tree diagram to represent problem solving sessions performed by the General Problem Solver (GPS) program. These diagrams, called *Problem Behaviour Graphs*, were a trace of steps taken by the GPS program through problem space in its search for a solution.

The visual aspect of the solution graph relates to the static property of the user's model; the dynamic aspect of the user's model is also important. This concerns the actions carried out by a user to extend the solution graph. A user generally employs two heuristics in the development of a solution: *stepwise refinement* (see Section 2.6) and *dynamic exploration* (Spence & Apperley, 1977). Stepwise refinement and dynamic exploration are complementary modes of investigation. Both are used during the development of an algorithm, so it is desirable to integrate both in a common user interface.

4.1.1 Stepwise refinement

The stepwise refinement heuristic for building a solution has been discussed earlier in Section 2.6. A solution graph is built up by the step-by-step addition of new branches. Each step follows a cycle that comprises three stages: (i) the *generation* of an interim goal; (ii) the *execution* of operations that could potentially attain the interim goal; and (iii) the *evaluation* of the extent to which the goal has been achieved. Stated tersely, the three stages involved are: generation, execution, and evaluation. The solution graph is incrementally extended by a discrete series of goal-setting and goal-achievement episodes. The discrete goal-oriented nature of stepwise refinement is highlighted in the diagram of Figure 4.3.

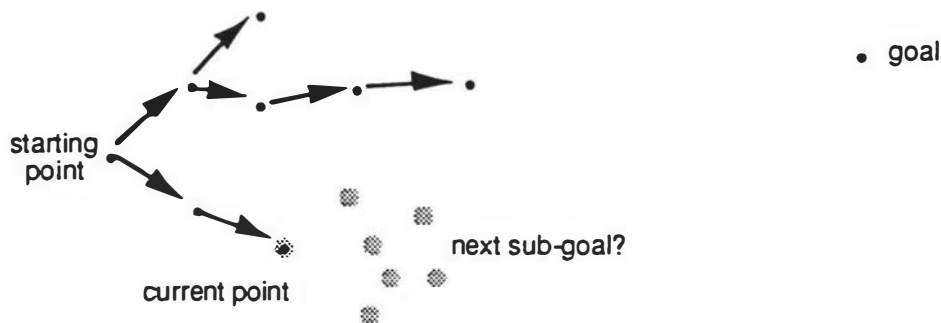


Figure 4.3: The solution graph is built step by step.

4.1.2 Dynamic exploration

To complement the *directed* approach of stepwise refinement, a more *exploratory* approach is sometimes needed, particularly in situations where the designer is at a loss to formulate a sub-goal. A user explores the solution space by deliberately varying the topology of a solution graph or varying the parameters of constituent operations, and observing changes in the generated results.

Notably, these variations are performed without first establishing a specific sub-goal. This is equivalent to sweeping a continuous region of the solution space, as shown in Figure 4.4.

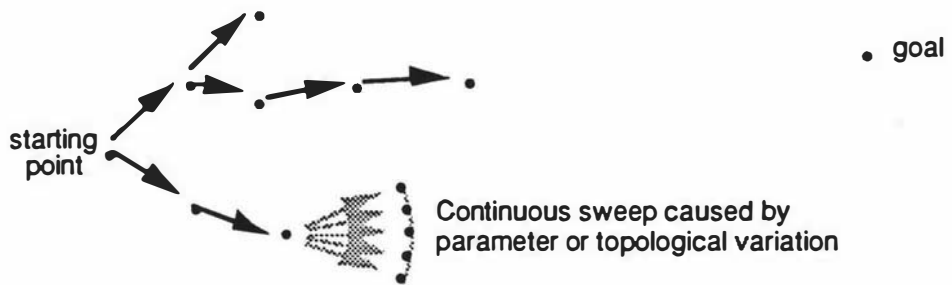


Figure 4.4: The solution graph representation of the effect of dynamic exploration.

Spence and Apperley (1977) used the term *dynamic exploration* to describe such a facility in the circuit design process. In the MINNIE circuit analysis package which Spence and Apperley described, designers could make continuous variations in component values and simultaneously and instantly observe the effect of these changes on a predetermined circuit behaviour. This enabled designers (i) to gain a "feel" for the effect of each component on the properties of interest, (ii) to smoothly adjust circuit parameters to an optimum response, and (iii) to easily and quickly test speculative ideas. Further, Spence and Apperley suggested that the continuous interaction did "... not simply speed up the design process ..." but that "... there are some ideas that would just not be tested with a longer response time ...". They drew a distinction between a conscious *trial-wait-observe error-decide-trial* activity and the dynamic exploration characterised by continuous interaction.

4.2 Generation

The extension of the solution graph, or the process of *generation*, involves finding a "suitable operation". For stepwise refinement, a "suitable operation" is one that achieves the prescribed sub-goal, while for dynamic exploration, it is one that will potentially yield results that advances the solution towards the ultimate goal. The interface can help a user find a suitable operation in two ways: by providing a complete picture of the solution graph and by providing a rapid means to search for and select an operation.

4.2.1 An iconic data flow language for image processing

An iconic data flow representation of an algorithm could provide the needed overview of the current state of the solution. Such a representation could be readily implemented as part of a graphical user interface in the form of a solution graph. For this reason, an iconic data flow language seems the

appropriate choice as the graphical metaphor (see Section 3.4.1) in a direct manipulation interface for the heuristic design of imaging algorithms.

Data flow diagrams have gained widespread acceptance in the field of software engineering as a visualisation tool to aid the planning and design of large software systems. De Marco (1978) and Gane and Sarson (1979), who are major proponents of data flow methods, use data flow diagrams to aid communication between customers and software developers in the early stages. De Marco claims that the diagrams provide a "big picture" of the system in a non-technical manner. In the design of a software system, the data flow diagram is only one of several techniques used to specify requirements. Detailed information, such as control and loop constructs, are specified in flow-charts and textual design documents. These additional documents are used by systems analysts and programmers and are important in the advanced stages of the development process.

Data flow diagrams serve as a visualisation tool in software engineering, but for an image processing system, they could serve as a complete specification for an executable algorithm. Therefore, data flow diagrams for an image processing system would differ in three principal ways to the diagrams used by De Marco: (i) they must have the ability to be interactively edited, (ii) they must be executable, and (iii) they must allow the specification of control and loop constructs.

The task of algorithm generation is considered to be one of extending a solution graph by the addition of new operations. An interface that assists a user in this generation phase should allow a user to see and understand - at a glance - the definition of the algorithm and to assist in the selection of new operations for the algorithm. These two facilities are discussed in the following sections.

Icons

It is desirable for a representation of an algorithm to document its function. That is, a user should be able to "see" the function of an algorithm at a glance. Self-documentation benefits all users, from the person designing a new system to the novice presented with an algorithm for the first time. The most highly visual elements in an iconic data flow language are the operation icons, hence close attention should be given to the design of clear and unambiguous icons.

In visual languages, icons are often used as graphical symbols to represent data objects or operations. An icon should indicate its referent clearly. This clarity is difficult to achieve in image processing applications because icons are often used to symbolise operations that perform abstract actions; such actions are generally difficult to show with pictures. For example, the Sobel filter would be difficult to represent with an icon, especially if the icon set must also meaningfully distinguish the Robert's product and other edge detection operations.

Apperley (1990) suggests the effectiveness with which an icon conveys its underlying meaning can be subjectively ranked into three classes: *transparent*, *translucent*, and *opaque*. These names draw on the analogy that icons are like a port through which the underlying meaning is viewed. If the association is made immediately and without explanation, it is classed as transparent. If the association comes easily after having been once explained, it is classed as translucent. If the association cannot be made without difficulty, despite repeated explanations, it is classed as opaque.

Rogers (1989) classifies icons on the way the form of an icon represents its underlying concept. An example for each class is shown in Figure 4.5. *Resemblance icons* depict the underlying meaning through an analogous image. *Exemplar icons* depict a typical example for the underlying object. *Symbolic icons* convey an abstract meaning with a simplified image. *Arbitrary icons* bear no resemblance to the intended meaning and so the association must be explained.

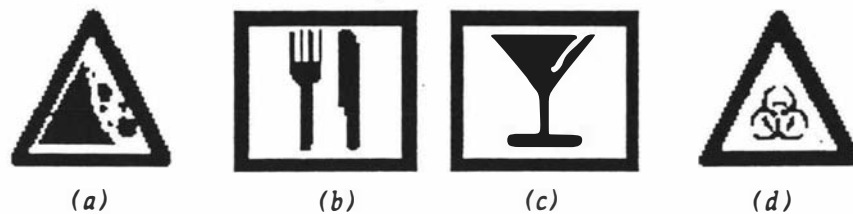


Figure 4.5: Different forms an icon can take: (a) resemblance - 'falling rocks', (b) exemplar - 'restaurant services', (c) symbolic - 'fragility', (d) arbitrary - 'biohazard'.

The choice regarding whether icons should refer to its underlying meaning by the use of graphics, text, or a combination of both, is important to the overall ease of use of the interface. Rasure and Williams (1991), two of the authors of the Cantata visual language, argue that to assign unique and well-suited icons to all operations would be a formidable undertaking owing to the sheer number of operations (over 250) that exist in their system. Cantata icons (called *glyphs*), shown in Figure 4.6, do not use graphical elements to assist the recognition of operations. Their identification rests solely on the textual label. Hirakawa *et al.*, authors of the HI-VISUAL system, argue that icons that represent actions are difficult to design. They avoid such difficulty in their interface by specifying operations through mouse movements rather than icons. However, a possible difficulty with such a scheme is the definition of mouse movements to distinguish between subtly different operations; for example, the Sobel and Laplacian filters.

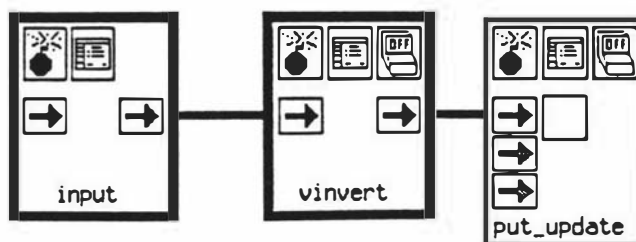


Figure 4.6: Cantata icons are identified solely on the basis of textual labels.

4.3 Execution

In a typical visual language, such as Prograph and LabVIEW, the selection of an operation from a menu creates an instance of that operation. In such an interface, a selection deposits an icon onto the graphical workspace. An operation, represented as an icon, has the properties of: a unique visual identity, a temporal existence, and a spatial existence. The execution of the operation involves the specification of its input and output data flows, and its parameters.

4.3.1 Specification of input and output data

Since an operation defines a process that transforms data, it must have means to accept input data and pass on output data. Most operations have single input and output channels. For example, for a VIPS box `average filter`, the input and output data are specified as symbolic references to image variables (`im_in` and `im_out`):

```
box average im_in im_out
```

Other operations accept more than one item of input data and return a single result, such as the operation to find the difference (`diff`) between two images (`im_in1` and `im_in2`) in Serendip (Wilson, 1987):

```
let diff = subtract( im_in1, im_in2 )
```

A generic imaging operation has multiple input and output data channels - where the exact number is determined by the specific operation. Some operations may have either no inputs or no outputs, such as operations to generate test images and operations to save data to disk. It is unlikely that an operation will have neither input nor output data channels, as such an operation can never be part of an algorithm!

In a visual language, data items flow along data paths. Inputs and outputs for operations are typically specified by joining the paths directly onto the operation icons at special termination points. Such termination points are depicted in Prograph as circles, and in Cantata as boxes with arrows inside as shown in Figure 4.8.

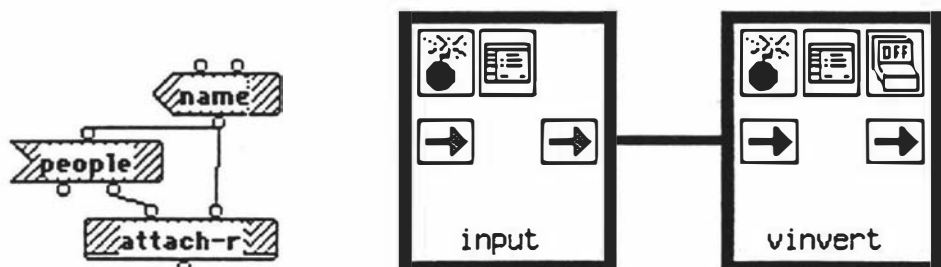


Figure 4.8: Termination points for (a) Prograph operations (circles), and (b) Cantata operations (arrows).

4.3.2 Specification of parameter values

Most operations require parameter values in the calculation of a result. These values influence the calculation of the result in some way. Some parameters are specified by a user, while others are specified by another part of the algorithm (Sakaue, & Tamura, 1985). For instance, the size of a window for a box average filter could be supplied by a user and the cut-off intensity values for a thresholding operation could be supplied by the algorithm. Examples of these cases are shown in the VIPS command lines:

```
box average im_in im_out ( 3, 3 )
threshold im_in lo_cut 255
```

where the value of `lo_cut`, which denotes the lower threshold intensity, is derived by the algorithm.

Calculation parameters can be specified implicitly via default parameter values. For example the following command lines are equivalent to the ones above.

```
box average im_in im_out
threshold im_in lo_cut
```

Command line systems offer little flexibility in the way a command can be invoked. Each command must be specified in accordance to a strictly enforced syntax. In contrast, direct manipulation interfaces allow calculation parameters to be specified in many ways. Graphical objects, called *controls* or *widgets*, allow values to be specified either as incremental or absolute values. A facility to specify incremental adjustments to parameter values is helpful in situations where the absolute value is not as important as knowing that the value should be "increased" or "decreased". However, when a specific absolute value is desired, it is appropriate to specify this directly rather than having to adjust an incremental control until the desired value is reached.

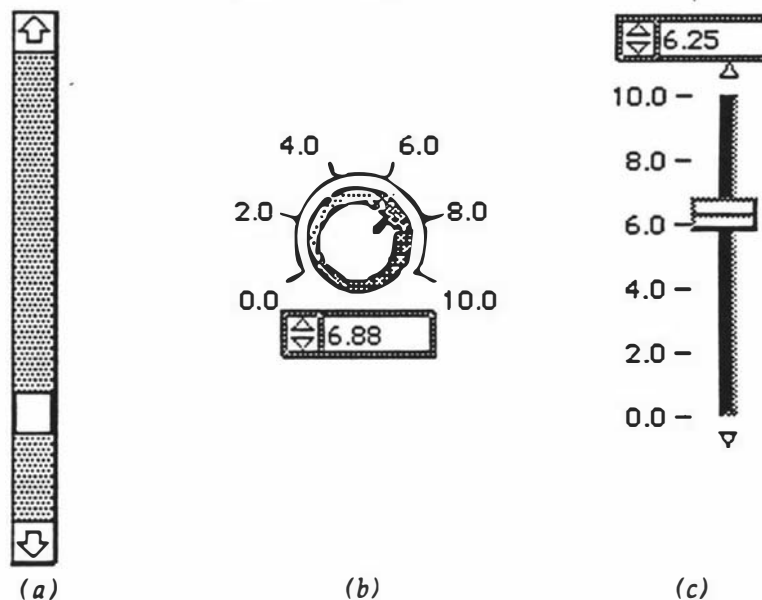


Figure 4.9: Controls objects for the setting of parameter values.

Figure 4.9 shows representative direct manipulation widgets. A familiar control is the slider bar, shown in Figure 4.9(a). This allows values to be specified in coarse and fine steps when the cursor is clicked on the grey background and arrowed regions, respectively. Elaborate variations of the scroll bar include the circular knob and slider shown in Figure 4.9(b) and (c); these are from the interface of the LabVIEW™ system. These widgets indicate the absolute value of the parameter setting as a decimal number. This number, displayed in an editable text field, can be changed by typing a new number over the top of the old. Coarse incremental control can be performed by engaging the knob or slider button. Fine incremental control - in single steps of the smallest increment - can be performed by clicking in the triangular button next to the decimal number field.

A characteristic common to all the controls shown in Figure 4.9, is representation of the adjustable value as a proportion of the dynamic range. Proportional, rather than absolute, representation of a value is appropriate in situations where the controlled parameter has no natural units; for example, the brightness and contrast settings for a cathode ray tube.

The controls shown in Figure 4.9 adjust parameter values in a continuous fashion. However, not all parameters are represented as a continuum, some are represented as discrete values. For example, discrete switches are used to specify the parameters of the Subtract operation, as shown Figure 4.10. Often the difference between two images results in negative values; however negative values are unsupported for unsigned byte images, which have an intensity range of 0 to 255. Therefore, switches are required for the Subtract operation to specify how negative difference values should be treated. The first switch in Figure 4.10, "Offset", is a simple on-off switch. It specifies whether or not an offset of 128 should be added to the difference values before they are written to the difference image. Such an offset will raise negative pixel values in the range -1 to -127 to positive values. The second switch specifies one of two mutually exclusive choices of "Wrap" or "Saturate". These functions specify two different ways to transform negative values to positive values. The first "wraps" the value around the dynamic range of 0 to 255. Hence the value -20 will be converted to the value 236 (256-20). The second option, "Saturate", assigns all negative values to zero.

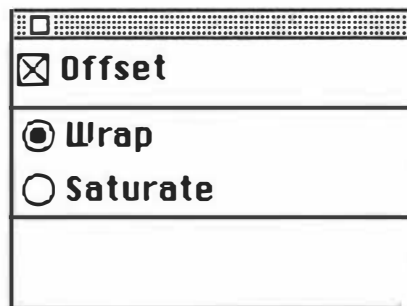


Figure 4.10: A control to enter discrete parameter values.

4.3.3 Operation invocation

Two obvious ways to execute a data flow algorithm are (i) to select an "execution" item from a menu and (ii) to double click an icon of one of the imaging operations. A program can be executed in the Prograph system by the selection of the "Execute Method" menu item, as shown in Figure 4.11. For image processing, the drawback of this method is in the granularity of what can be selected for execution. Prograph allows entire programs and methods (sub-programs) to execute, but in image processing it is often useful to specify execution of an algorithm beginning at a particular operation. This leads to the second way to run an algorithm: double-click on an operation. The execution of an operation should lead to the execution of all operations that follow it in the algorithm. With this method, the execution of the entire algorithm can be performed by invoking the first operation in the algorithm data flow. A system need not provide both methods of invoking an execution. For an image processing system, the more useful method would be the double clicking method because it offers finer selectivity of what can be executed.

Execution	
Run	⌘R
Execute Method	⌘E
Abort	⌘Z
Edit Application	⌘`
Set Program	
Clear Program	
Step/Show Level...	
Step/Show On	⌘[
Step/Show Off	⌘]
Breakpoint On	⌘⇧[
Breakpoint Off	⌘⇧]
Clear Steps & Breaks	⌘⇧⌘]

Figure 4.11: Menu item for the execution of a Prograph program.

Once an operation has been invoked, its execution will typically initiate the execution of many other operations. The order in which the subsequent operations are automatically executed is determined by the *execution scheduler*. In a typical iconic data flow language, like the Prograph example shown in Figure 4.12, the output of one operation feeds into the input of another operation. Therefore if either of the '*' operations execute, then the '+' operation must update its result to reflect its new inputs. The updated results for the '+' operation, will in turn, initiate the recalculation of the 'sqrt' operation. In this way, the execution of either '*' operations will cause the execution of the '+' operation followed by 'sqrt' operation. This style of execution, known as *data-driven* execution (Ackerman, 1982), is appropriate for image processing because the designer is interested how a new parameter will effect the entire algorithm.

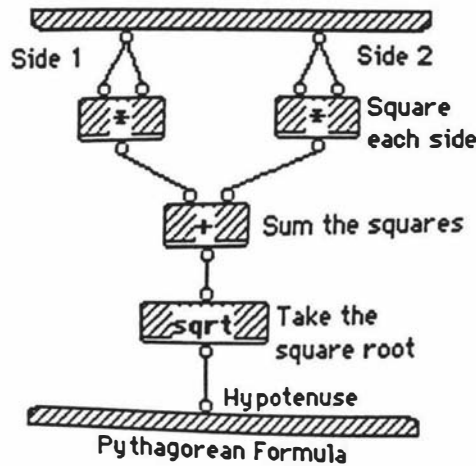


Figure 4.12: Prograph uses the classic 'box' and 'stick' representation for an algorithm.

An investigation of parameter values could be hastened if the adjustment of a parameter value could automatically trigger re-execution of an algorithm. This facility could more than quicken the investigation, it could potentially induce dynamic exploration. For this to happen, the system must respond to adjustments in parameter value very rapidly. Ideally, the system should respond so rapidly that the generation of a series of parameter values would cause the changes in the displayed result to appear animated.

4.4 Evaluation of the results

After the selection and execution of an operation, the user must evaluate whether the results satisfy the goal of the action. If the results are satisfactory, then the development moves to the next step. However, if the results are unsatisfactory, then the user must assess whether the deficiency was caused by a bad choice of operation or parameter setting. This decision may lead to the use of new parameter values or a new operation altogether.

The goals for stepwise refinement and dynamic exploration differ, hence the respective evaluations, also differ. For stepwise refinement, the basic question is "has the sub-goal been attained?". For dynamic exploration, the equivalent question is "does this result bring me closer to the ultimate goal?". Regardless of the approach, the user is helped greatly if the new result may be easily related back to the operation that generated it. Such assistance can be provided by displaying the new result near the operation that generated it, and displaying it without delay.

Visibility of results and the promptness of delivery are important attributes relating to the assessment of results. Obviously the result must be visible in order for it to be assessed. A natural way to display an image result in a graphical interface is to place it in a window. The default position for the window should be near its associated operation because this helps a user to link the related items. The Cantata system works in this way, as shown in Figure 4.13.

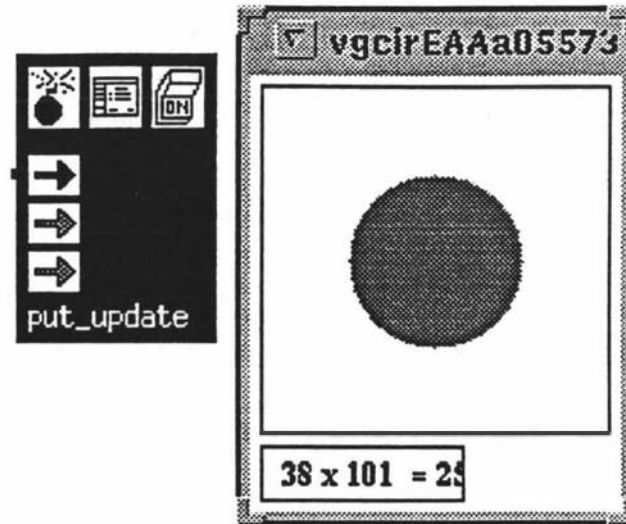


Figure 4.13: A Cantata operation and its result which is displayed nearby in a movable window.

In Cantata, the result of an operation cannot be displayed without first attaching an operation called `put_update` to it. The need to use a specialised operation to perform a routine action - such as displaying an image - is an indirect interaction that could be avoided if each operation includes a facility to display its own results.

4.5 Summary and Conclusions

The design philosophy adopted in this chapter is one of user-oriented design. This advocates that a design should focus on the needs of the user. For imaging applications, a representation of the algorithm based on the solution graph would achieve this aim. A solution graph visually represents the operations used in the solution and the data-relationships between them. An iconic data flow language would explicitly represent the solution graph in an interface.

The problem solving strategies, or heuristics, commonly used to build the solution graph, are stepwise refinement and dynamic exploration. Both strategies involve acts of generation, execution, and evaluation in the extension of a solution. The major distinction between the two methods exists in the degree to which the three activities are integrated. In stepwise refinement, each step is distinct from the other. A deliberate sub-goal is decided upon; then an operation is chosen to specifically attain the sub-goal; and finally the extent to which the sub-goal is attained is evaluated. In dynamic exploration, the three steps are integrated into one continuous activity. An operation is selected according to how likely it is to produce an "interesting" result. The operation is executed many times in quick succession, each time with different parameter settings. Results are generated with sufficient rapidity to induce a sensation that the parameter controls are directly connected to the image display. Ideally the system responds so rapidly that continuous changes in parameter values appear to animate the

change of results. This high degree of interaction promotes free investigation which is a primary characteristic of dynamic exploration.

Interspersed amongst the philosophical discussion of design in this chapter, concrete recommendations for the implementation of an image processing system have been suggested. In effect, these recommendations constitute a design framework based on a user-centred approach to heuristic imaging algorithm development. To uphold the principles of user-centred design, these guidelines should be incorporated into an implementation of the design. The recommendations of this chapter are as follows. This chapter recommends that:

- facilities should be provided to perform generation, execution, and evaluation as discrete actions (for step-wise refinement) and as a continuous action (for dynamic exploration);
- an iconic data flow language be used (so that the user can quickly ascertain the state of the solution and hence make a reasoned decision for the next plan of action);
- the aforementioned language be modelled after the solution graph (because the solution graph has been adopted as the user's model);
- the data flow language be editable and executable;
- the data flow language should incorporate control structures;
- the icons of the proposed language contain pictures and text (to improve the documentation of algorithms);
- operations be selectable from a menu structure (because a search for operations is necessary in both stepwise refinement and dynamic exploration approaches);
- input and output data for an operation be specified by graphical terminations placed somewhere on an operation's icon;
- specification of parameters be done through direct manipulation widgets (to allow numeric values to be specified in incremental and absolute styles and to allow choices to be specified through discrete switches);
- operations be invoked by the action of double clicking on its icon or by changing a parameter value;
- the execution of one operation should in turn initiate the execution of all operations that depend, directly or indirectly, on its result; such execution can be achieved through the use of a data-driven scheduler;
- results should be displayed in windows, which are movable, and by default position²⁴ close to its associated operation.

These design guidelines have been incorporated into a visual language for image processing algorithm development, which is presented in the following chapter.

Chapter 5

OpShop: An Implementation

This chapter presents an overview of the software implementation of the design presented in the Chapter 4. The description is in four sections: (i) a description of the elements of the user interface, (ii) a description of the user interactions, (iii) an overview of the key aspects of the software design, (iv) and an evaluation of the speed of computation.

5.1 An overview of OpShop

The purpose of OpShop⁵ is to provide an environment for the heuristic development of image processing algorithms. This is achieved by providing a

⁵ In New Zealand, an opportunity shop (colloquially called an opshop) is a store where one goes to buy second hand articles at bargain prices. This name has been chosen for the software presented in this chapter to reinforce the idea of inexpensive yet delightful discovery - in this case, of solutions to algorithms, rather than of *haute couture*.

visual language that serves as both a programming language and a program development environment. The two functions of the system are inseparable. This is often the case for visual language programming environments (Ambler & Burnett, 1989). OpShop provides an integrated environment for both the development and execution of an algorithm. This integration leads to a high degree of interactivity in the development process because it minimises the time taken to turn an idea into code, perform a trial, and to make modifications.

The principle features of any visual programming environment are the representation of the program under construction and the interaction it supports to allow the user to edit programs. A *program visualization* system (Myers, 1990) graphically represents data structures and control flow in a program, but does not provide facilities to allow programs to be visually edited. A *visual programming* system provides for both. According to this classification, OpShop is a visual programming system: it depicts algorithms by a visual data flow language and supports the direct manipulation style of user interaction.

In OpShop, an algorithm is represented as a data flow network of imaging operations. An example of its iconic visual language is given in Figure 5.1. Operations are depicted by icons, while data flows between operations are represented by interconnecting lines. The points at which data flows connect to operations are called *terminals*, which are depicted by triangular symbols. Two types of terminals exist; one to support input flows and the other to output flows. Input terminals are situated on the left side of an operation icon, while output terminals are situated on the right side. This arrangement of terminals leads to a predominantly left-to-right flow of data through an algorithm; algorithms are generally read from left to right. Figure 5.1 represents a sample algorithm that can be directly executed.

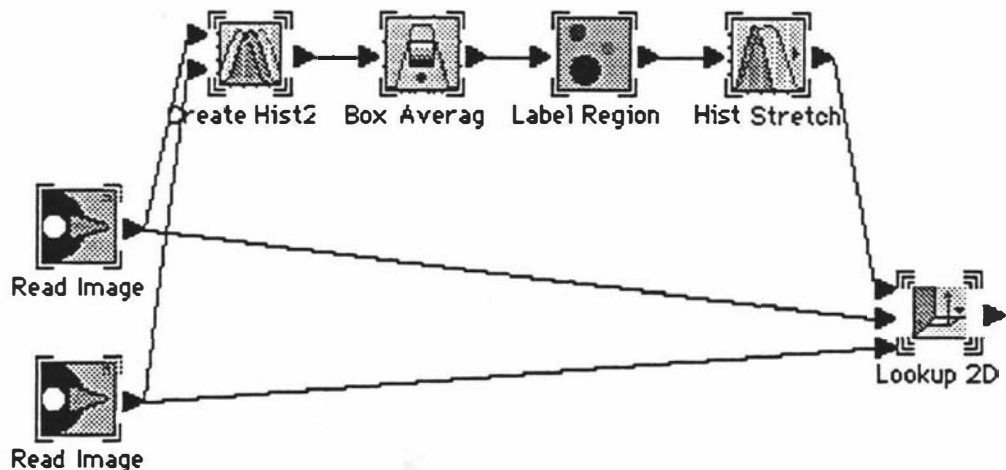


Figure 5.1: An OpShop algorithm to perform feature classification based on colour.

OpShop algorithms, such as the example shown in Figure 5.1, are directly edited by the user through actions specified by a pointing device. Typically, a mouse is used to directly engage the iconic elements of the OpShop language through

actions such as clicking and dragging. The algorithm topology is modified through such mouse actions.

The OpShop language provides a complete specification of an algorithm. It is not compiled or translated into an intermediate language in order for it to be executed. Rather, the visual representation is executed directly. Therefore, not only is the editing of an algorithm specified by mouse actions, but also the execution of an algorithm. OpShop integrates into a single environment, facilities for the graphical display, the visual editing, and the execution of algorithms. This feature, coupled with rapid calculation of results, provides a highly interactive environment that is conducive to the heuristic development of imaging algorithms.

The OpShop software environment was developed on a Macintosh LC computer, which was configured with 10M Bytes of RAM, a 40M Byte hard disk and version 7.0.1 of the Macintosh operating system software. The *Info Box* for the OpShop application is shown in Figure 5.2; the OpShop icon portrays the data flow nature of the OpShop language. The programming language used, THINK C 5.0.2 (Symantec Corporation, 1991), is an object-oriented language. The OpShop software extensively uses the object-oriented features of THINK C and the THINK class library (TCL 1.1). To run, the OpShop application requires a Macintosh configured with 8-bit colour display, system software version 6.0.7⁶ or later, and at least 3 MBytes of RAM free after the system software is loaded.

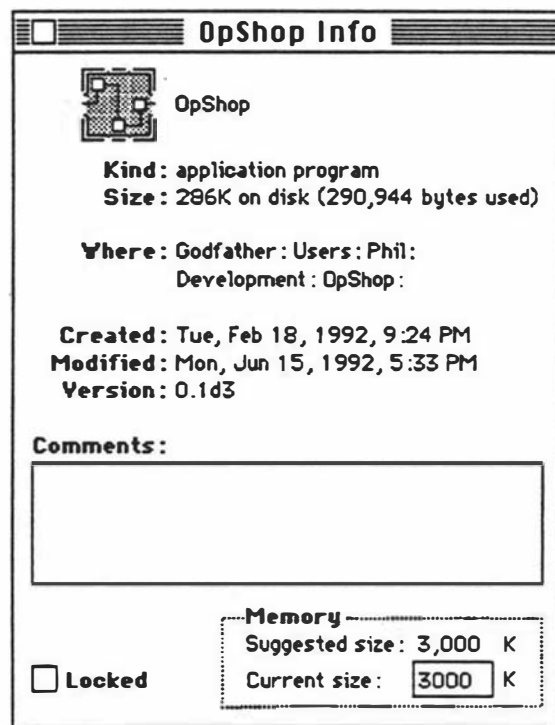


Figure 5.2: The "Info Box" for the OpShop application.

⁶ The 32-bit Quickdraw INIT is required when OpShop is run under System 6.

OpShop comprises a collection of 67 operations that span a range of imaging functions including arithmetic calculations, preprocessing, segmentation, and image measurement. The functional description for each operation is given in Appendix 1.

5.2 Elements of the visual language environment

This section presents an overview of the graphical elements of the visual language environment. The elements are discussed in the same order that they would be met by a user in the construction of a new algorithm. Therefore, the *whiteboard* is described first, because development activity cannot begin without a work area. Presented next are *operations*. These are deposited onto the whiteboard when selected from the OpShop menu bar. Combinations of operations, *algorithms*, and a description of how they are created, are presented next. Finally, *subflows* are described. A subflow simplifies a cluttered whiteboard by reducing the number of icons on a whiteboard without altering the functionality of the algorithm.

5.2.1 Whiteboard

Algorithms are built and executed in a work area called a *whiteboard*. As shown in Figure 5.3, a whiteboard looks like a typical Macintosh window with the exception of an additional text box in its bottom left corner that indicates the amount of memory available for allocation to new operations. Two operations (Read Image and Threshold) appear in the whiteboard shown in Figure 5.3. The whiteboard supports typical Macintosh window functions: it can be closed by clicking in the close box (which is the left box in the title bar), it can be enlarged to fill the screen by clicking in the zoom box (which is the right box in the title bar), and it can be hidden behind or exposed from behind other windows. Several whiteboards can exist concurrently during an OpShop session, but only one can be active at any time.

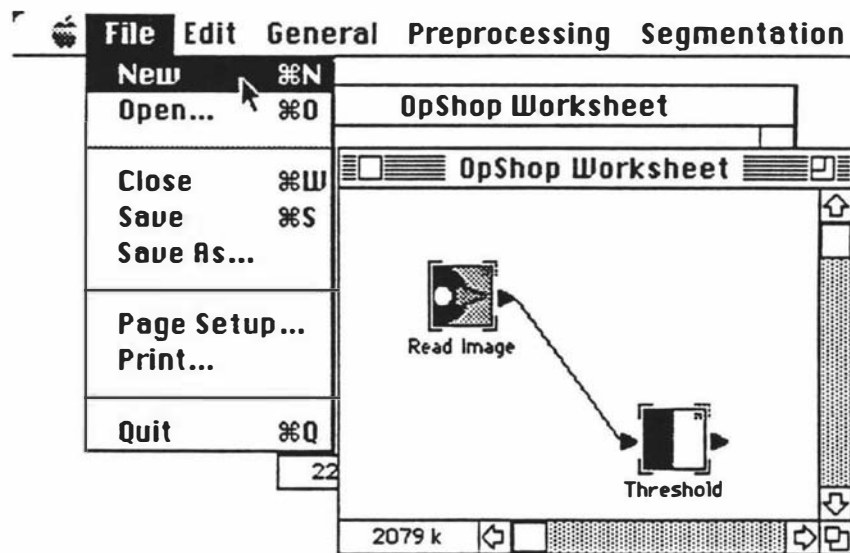


Figure 5.3: An OpShop whiteboard containing two data flow linked operations.

Whiteboards are created by a user selection on the New menu item. An existing whiteboard and its contents can be written and read to file using the Open... and Save... menu items. Algorithm topology and parameter settings are saved with the file, but image data is not saved. Whiteboard files exclude image data to avoid storage problems. Without saving image data, the whiteboard data files are typically three kilobytes in size, rather than hundreds of kilobytes if images are included. The image data can readily be reconstructed by simply executing the restored algorithm.

5.2.2 Operations

The basic element of an algorithm is an image processing operation. In OpShop, operations are represented by graphical symbols, called *icons* (see Figure 5.4). Icons can be created, deleted, stored on disk and retrieved, and repositioned on a whiteboard. The direct engagement properties of OpShop icons conform to the behaviour prescribed in 'The Macintosh user interface guidelines' (Apple Computer, 1985). In other imaging systems, operations do not have a temporal existence, but rather exist as functions that are executed when invoked from a command line or a menu selection.



Lookup 2D



Wedge

- (a) Non-selected Lookup 2D operation (b) A selected wedge operation. Its pop up menu has been invoked by option-clicking on its colour icon.

Figure 5.4: Examples of OpShop operations.

The OpShop icons all have a number of visual attributes, which are discussed below:

- Each operation is denoted by a unique 32 x 32 pixel colour image. The unique appearance of each icon not only assists in the identification of each operation, but also indicates key characteristics of the operation. For example, the four non-linear filters shown in Figure 5.5 display a family resemblance whereby each icon has a vertical bar of sorted intensity values to indicate that each is a rank based operation.



Min

(a)



Max

(b)



Range

(c)



Rank

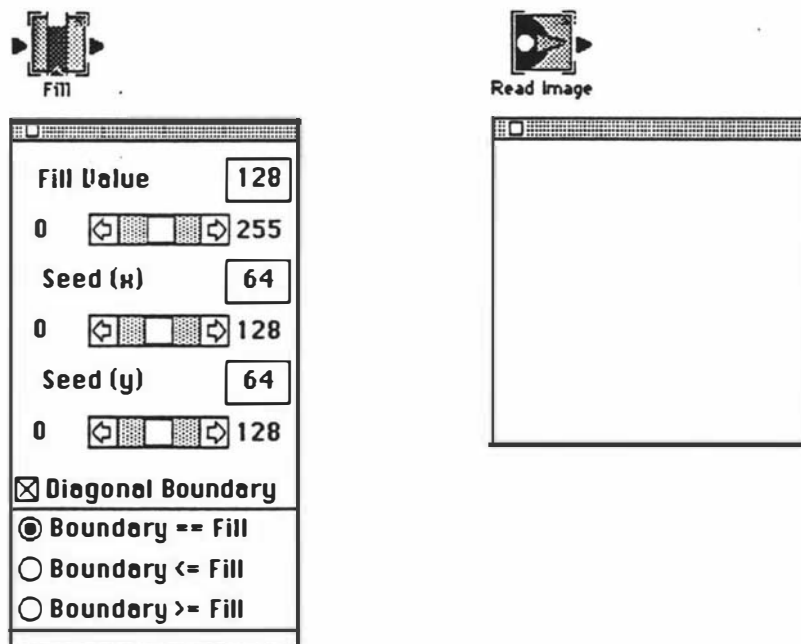
(d)

Figure 5.5: Non-linear filters: (a) Minimum, (b) Maximum, (c) Range, and (d) Rank filters.

- Each operation has a text label indicating the name of the operation. An operation is more readily identifiable when its icon is augmented with a name label (see Section 4.2.1). When an operation is selected, the text label is highlighted. Figure 5.4(a) and (b) show selected and unselected operations.
- Each operation has a set of input and output data terminals to receive data from and pass data to other parts of the algorithm. An operation is limited to five input and five output terminals, as the physical dimensions of an icon cannot accommodate any more. These limits are unlikely to pose a problem as imaging operations with five inputs or outputs are rare. The most that OpShop currently uses is three inputs (Lookup 2D) and two outputs (Distribution). Most operations will have one input and output, such as the filter operations shown in Figure 5.5. Operations that generate image data internally have no input terminals, such as the Wedge operation shown in Figure 5.4(b). The Save Image operation has no outputs because it transfers its data to disk.
- Each operation has its own menu from which actions specific to that operation can be invoked. The menu is accessed by pressing the option key while clicking on the operation's icon. Figure 5.4(b) shows the menu for the Wedge operation; this menu has only a single item, which is to display the window in which the operation's parameters are set.

Associated with each operation are calculation parameter values and results. These can be accessed for each operation by option-clicking the operation icon and option-clicking an output terminal, respectively.

Control panels



(a) The *Fill* operation has both numeric parameters and option switches.

(b) The *Read Image* operation requires no controls to carry out its function.

Figure 5.6: Examples control panels.

Operations have parameters that are used in the calculation of results. These parameters are generally simple numeric values and switches that select the way a calculation is performed. Some operations may have parameters that are a mixture of numeric values and switches, while others may have no parameters. Each OpShop operation has an associated control panel which can hold sliders, text fields, and switches for specification of calculation parameters. Figure 5.6 shows the controls for the operations `Fill` and `Read Image`.

The `Fill` operation has a combination of a slider and a text field to specify scalar parameter values, a check-box switch for choosing one of two options (diagonal or orthogonal fill region), and a set of radio button switches for choosing one of three options. The control panel for the `Read Image` operation is empty because it does not require any preset input from the user to perform its function.

OpShop control panels are implemented as windows that "float" on the whiteboard window. Floating windows remain visible, hence accessible, even when the user clicks in the document window, as shown in Figure 5.7.

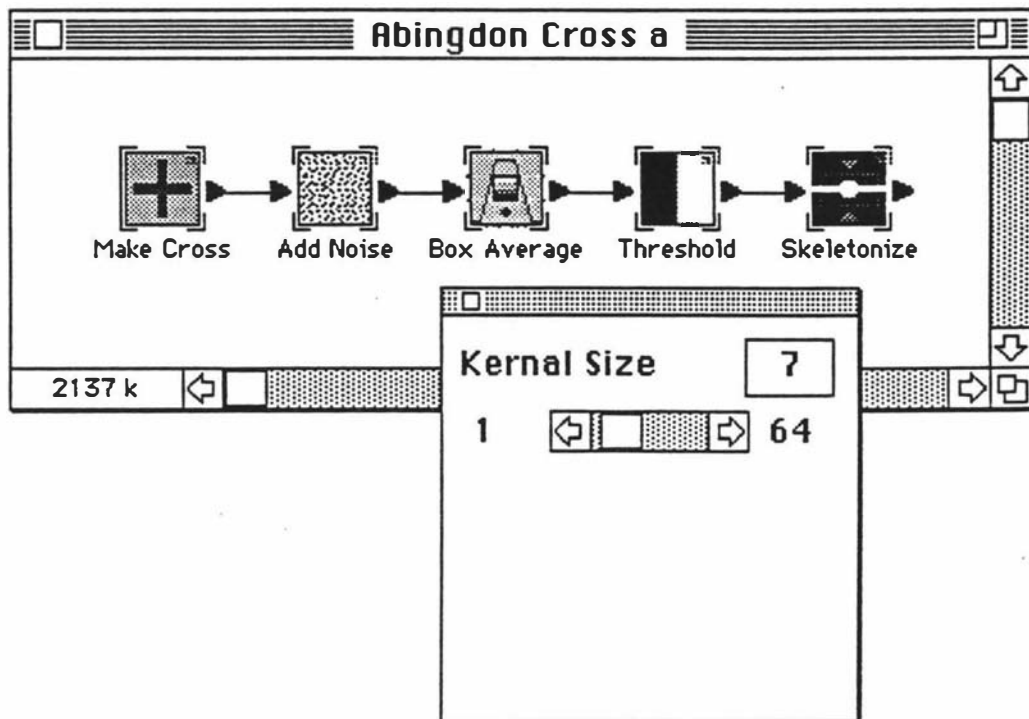


Figure 5.7: Floating control panel for the `Box Average` operation.

Parameter values can be changed easily with floating windows because the controls are always immediately accessible. If the panels were instead implemented with modal dialogue boxes, they would need to be opened and closed each time a parameter was adjusted, thus greatly reducing the overall ease of use. Furthermore, a user could not interact with an algorithm while a modal dialogue box was present. Clearly, floating windows offer more flexibility for interaction than modal dialogues.

Terminals

As mentioned earlier, an algorithm is represented in OpShop as a network of interconnected imaging operations. Data flows between operations along paths indicated by graphical lines. The points where these paths attach to the operation icons are called *terminals*. These are depicted as triangular symbols along the left and right edge of an operation icon, as shown in Figure 5.8.

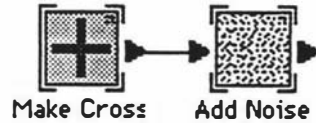


Figure 5.8: Data terminals for operation icons are indicated by triangular icons on the vertical sides of operation icons.

In an earlier version of OpShop (Ngan *et al.*, 1990), data paths were not terminated at specific data terminals, but instead at the operation themselves, as shown in Figure 5.9. The major drawback of this scheme was that operations could have only one input and one output. This was regarded as severe limitation as many imaging operations demand multiple inputs or outputs or both.

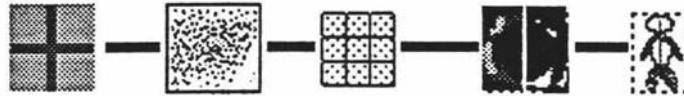


Figure 5.9: Data terminations used in an earlier version of OpShop.

Data terminals also serve as visible symbols to indicate the direction of data flow through an algorithm. The combination of data flow lines and terminal symbols look like arrows, which point in the direction of data flow.

The OpShop language includes five different types of data terminals to support four dimensionalities of data (scalar, 1D, 2D, and 3D) and an *unknown* type. Each type is distinguished by a different colour. A terminal will form a connection only with another terminal of the same dimensionality. The exception is that any terminal can connect to a terminal of the *unknown* type, in which case the unknown terminal will assume the dimensionality of the first terminal to which it connects. When all interconnections are removed from an unknown terminal, it reverts to the unknown state.

A data terminal does not distinguish the type of the data it handles; types meaning byte, integer, real, etc. Operations are assumed to accept any type of data for processing. In object-oriented terms, the operations are said to be *polymorphic*. At present, OpShop supports the following data types: byte (1-byte), short (2-bytes), long (4-bytes), float (4-bytes), double (12-bytes), complex-float (8-bytes), and complex-double (24-bytes).

All terminals are associated with data. Data can be valid and invalid. When an algorithm is in a stable state - that is, when no calculations are in progress or pending - all data in the algorithm are valid. However, while calculations are taking place, certain data buffers may await recalculation. In the time pending their recalculation, the data stored in these buffers are functionally invalid. To

indicate this invalid state, a terminal displays a black mark on the tip of its triangular icon. Figure 5.10 illustrates the visual distinction between the two states.

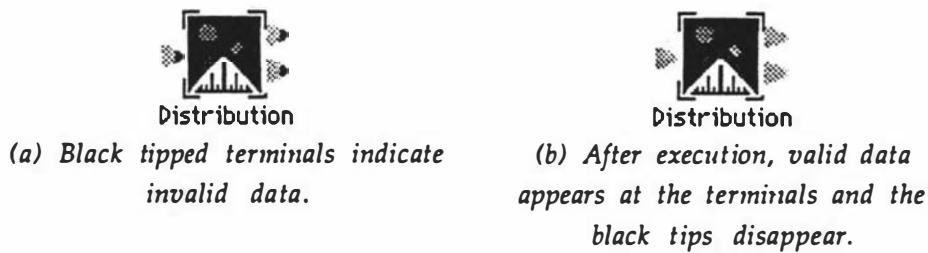
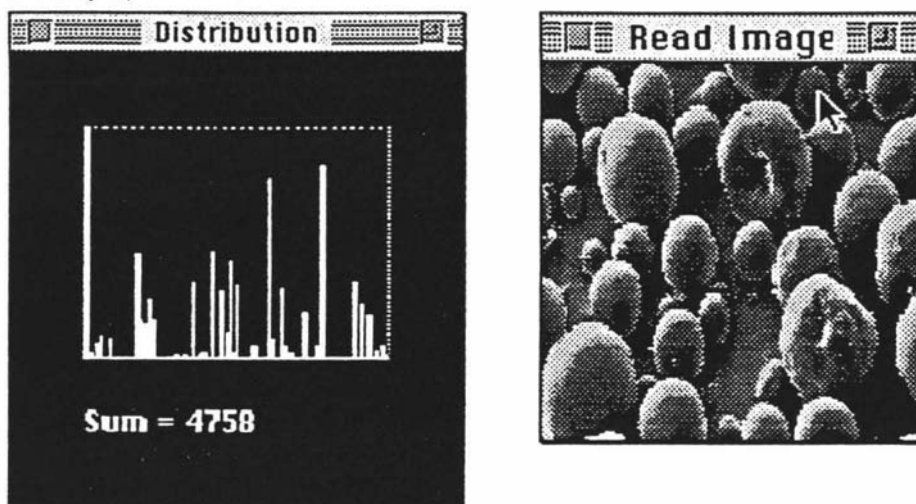


Figure 5.10: Invalid and valid data terminals.

Each output terminal has an associated local result buffer. Although the dimensionality of the buffer is fixed, the size of the dimensions and the data type of the buffer can vary at run-time. The contents of a local result buffer are displayed in result windows which are associated with each output terminal. The user opens a display window by pressing the 'option' key while clicking the mouse button on the output terminal. Figure 5.11 shows the display windows for one- and two-dimensional results. An example of a one-dimensional result is an intensity histogram. An example of a two-dimensional result is a grey-scale image. The title of the result window indicates the operation which relates to the displayed data.



(a) A one dimensional result

(b) A two dimensional result

Figure 5.11: Display windows.

5.2.3 Algorithms

Individual operations discussed in the previous section rarely constitute a complete processing strategy. More commonly, groups of operations are used to perform an imaging task. In OpShop, groups of operations arranged in a data flow configuration are called *algorithms*. Although the term *algorithm* normally implies a sequential procedure, it is used in this thesis to refer to an OpShop data flow in order to remain consistent with conventional image processing terminology.

User interaction in the construction of algorithms

Algorithms are formed in the OpShop system by linking the data terminals of separate operations with a data flow. The user creates a data flow by the action of pressing the mouse button on the terminal of one operation and while keeping the mouse button held down, moving the cursor until it is above the target terminal, then releasing the mouse button. Visual feedback during the linking process is given in the form of various cursor shapes. The most useful is the rubber band cursor. Figure 5.12(a) and (b) illustrates two of the steps involved in the creation of a data flow. The cross- (in (a)) and links- (in (b)) shaped cursors indicate *keep-dragging* and *ok-to-release* actions available to the user.

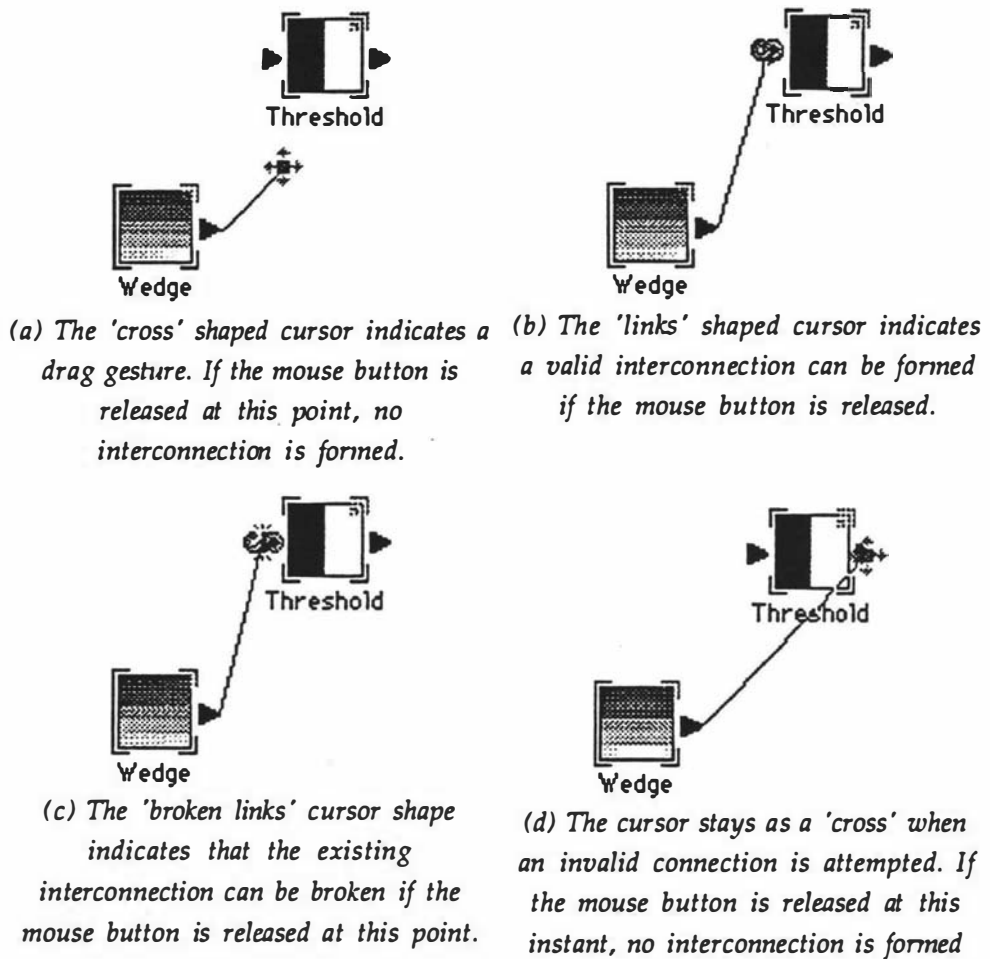


Figure 5.12: Action sequences involved in the creation and deletion of data flows.

To disconnect a data flow, the user clicks on the input terminal end of the flow, as shown in Figure 5.12(c). The 'broken links' cursor indicates that the data flows will be disconnected if the mouse button is released.

During the sequence of user action to create a data flow, the shape of the cursor indicates whether an attempted connection is permissible. Figure 5.12(d) illustrates that the shape of the cursor stays as a 'cross' rather than changing to 'links' when illegal connection is attempted; in this case, an output terminal cannot be connected to another output terminal. The cursor is designed to remain unchanged for illegal actions so that the user can visually perceive that the

action is illegal and will know not to try it (Gaver, 1991). Attempts at illegal connections occur in other situations besides the one shown. The following connections are not allowed:

- (i) output terminal to output terminal;
- (ii) input terminal to input terminal;
- (iii) to input terminal that already has an established connection;
- (iv) between input and output terminals of dissimilar dimensionality (except if one of the terminals is of the *unknown* dimensionality).

Types of algorithm topology

Multiple interconnections can emanate from a single output terminal, as shown in Figure 5.13, which indicates that the output data of the Read Image operation is accessible by both the Box Average and the Max operations. However, an input terminal cannot have multiple sources converging into it. Such a connection would not make sense because an input can only have one source. If an operation requires multiple input sources, such as the subtraction of two images, then this operation would require multiple input terminals, as shown in the Subtract operation of Figure 5.15.

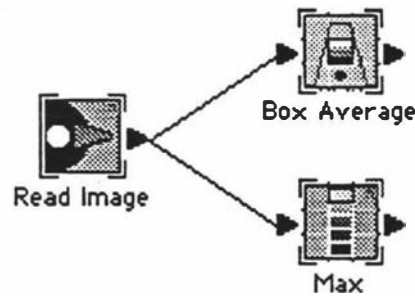


Figure 5.13: OpShop supports multiple connections to an output terminal.

Algorithms commonly consist of operations that do not have multiple inputs or outputs. Such algorithms can be constructed as linear sequences of operations, as shown in Figure 5.14. A linear sequence is formed when every operation has a fan-out of one; in this respect, the linear sequence is a special case of the parallel topology.



Figure 5.14: Linear sequence of operations.

Parallel data paths can merge at an operation that has multiple input terminals. Here the two dimensional expressiveness of a visual language is well suited to show the parallel nature of the data-flows as illustrated in Figure 5.15.

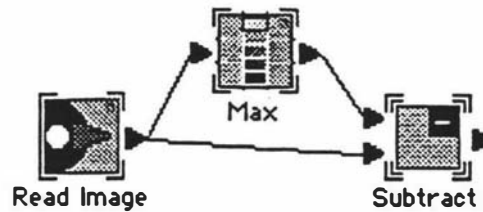


Figure 5.15: Parallel sequence of operations.

The number of operations that can be included in an algorithm is limited only by the memory space pre-allocated to the program at start-up; by default OpShop requires 3 MBytes. This figure can be changed in the Macintosh™ Finder through the *Information Box* for the application (see Figure 5.2). A user should exercise caution when the memory allocation is decreased to a value less than 3MBytes, to ensure that the available memory is never entirely depleted. Each operation deposited onto the whiteboard uses a certain amount of the pre-allocated program memory; for example, a Read Image operation that supports a 128 x 128 pixel result buffer occupies almost 80 kBytes of program memory. The amount of memory consumed by each operation is largely dependent on the size of the result buffer.

5.2.4 Subflows

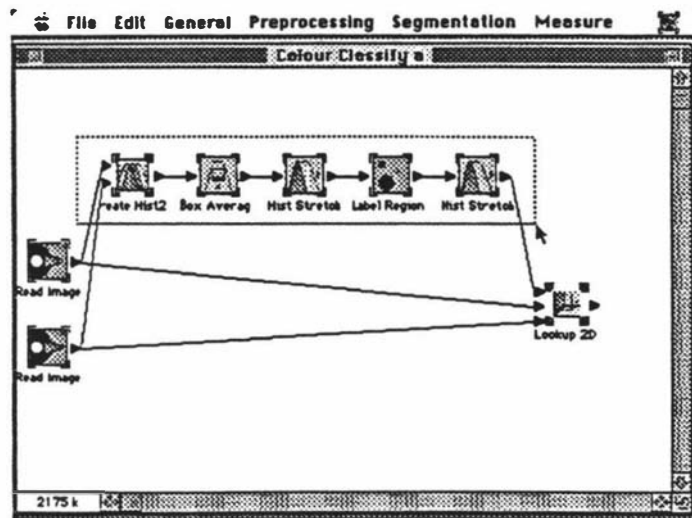
A problem common to all visual language environments is the lack of screen space to display programs (see Section 3.5.5). Graphical representations of a program or algorithm typically occupy large expanses of the screen. OpShop addresses the shortage by providing two mechanisms: scrollable whiteboards (as described in Section 5.2.1) and visual procedures called *subflows*.

Scrolling the document windows is not an optimal way to overcome the problem of a screen overcrowding. Consider the situation where an algorithm has two operations which need to be repeatedly accessed, but are separated by a distance greater than a screen width. The user is forced to repeatedly scroll the window to access the alternate operation. It would not take long for this activity to become tedious. The need to scroll could be eliminated if the separated operations were brought closer together. This can be achieved by combining all operations between them into a single operation called a *subflow*. A subflow is the visual equivalent to a procedure in text based programming languages such as FORTRAN and C.

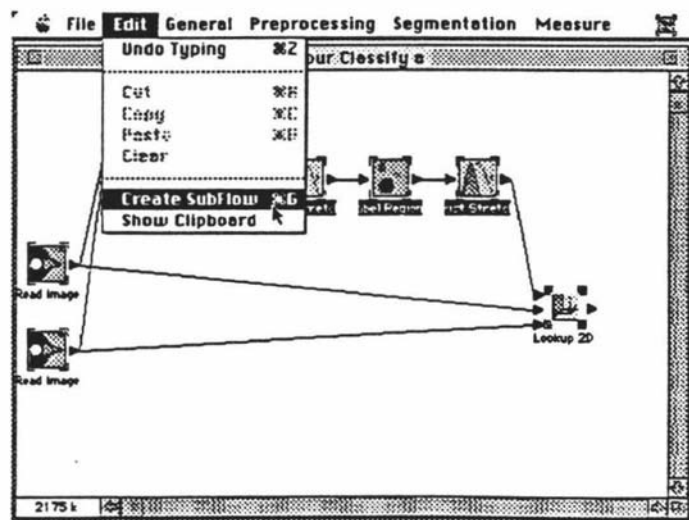
Subflows do not merely conserve screen space, but also provide a way to organise an algorithm into logical units, where each unit would perform a well defined task. The organisation of source code into modules and libraries is a common practice in the programming of textual languages.

Figure 5.16 illustrates the steps involved in the creation of a subflow. First, all operations to be encapsulated in a subflow are highlighted. Figure 5.16(a) indicates a way that a group of operations may be selected: by dragging a

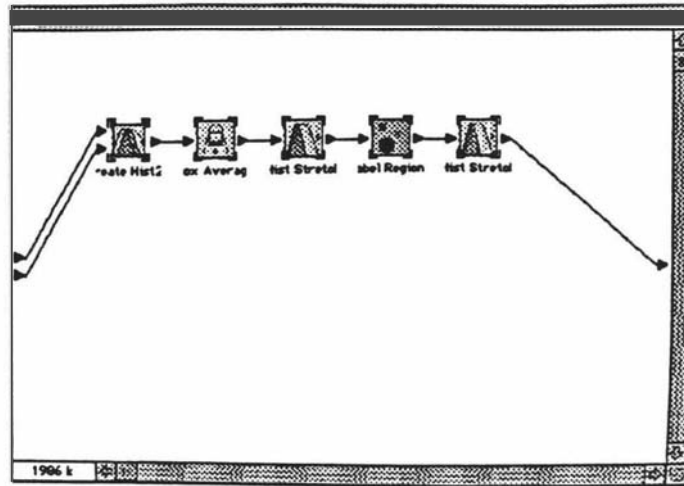
selection rectangle across the required icons; shift-clicking is another action that can add or remove icons from a selected set. Once the set of operations to be transferred has been selected, the **Create SubFlow** item is chosen from the **Edit** menu (see Figure 5.16(b)). The system responds by creating a new whiteboard and placing within it the interconnected algorithm fragment, as shown in Figure 5.16(c). The selected set of operations is replaced in the parent flow with a single operation called **SubFlow**, as shown in Figure 5.16(d); all necessary connections to the subflow are automatically created.



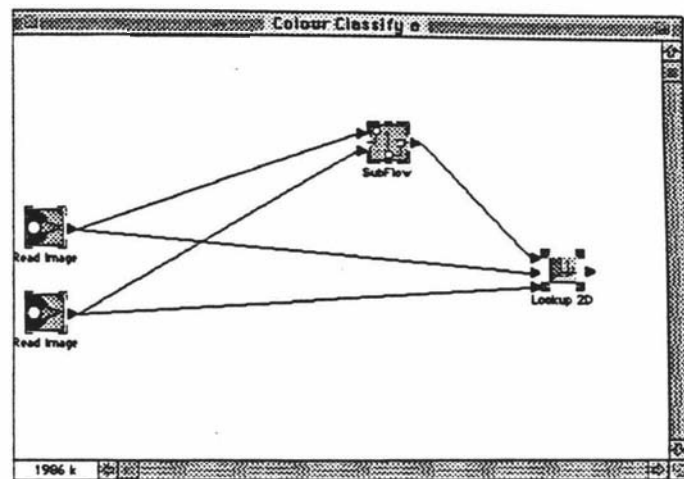
(a) The operations to include into the subflow are selected by a dragging motion.



(b) The selected operations are grouped into a sub-flow by choosing the **Create SubFlow** command.



(c) The whiteboard for the newly created subflow.



(d) The modified whiteboard for the parent flow.

Figure 5.16: The sequence of user actions involved in the creation of a subflow.

The subflow mechanism introduces a new type of terminal whose specific function is to pass data between a subflow and its parent flow. These terminals for subflow operations occur in pairs: one is situated on the side of the subflow icon while the other is situated on a side of the subflow whiteboard. Examples of such terminals are indicated in Figure 5.16(c) and (d). The two terminals on the left edge and the single terminal on the right edge of the subflow whiteboard in Figure 5.16(c) correspond to the two terminals on the left edge and single terminal on the right edge of the SubFlow icon of Figure 5.16(d).

As with sub-program constructions in any language, a sub-program can be called from within a sub-program and so a hierarchy of nested sub-programs can be created. The hierarchical organisation of a program can enhance the readability and comprehensibility of a program. OpShop has no limit to which subflows can be nested, save the limitation of allocated memory space.

5.3 User Interaction

This section discusses the dynamic properties of algorithms and the user interaction required to execute them.

5.3.1 Execution

Execution of an algorithm is performed by simply double clicking on an operation. In response, this operation recalculates its result which is deposited into its local result buffer. This result may then be used as input for other operations, in which case this second operation must also update its result because its output is no longer consistent with its input.

The data-driven execution scheme in OpShop ensures that the necessary calculations are performed to maintain a consistency between results and the algorithm definition. Therefore, the execution of a single operation by double-clicking may initiate the execution of many other operations. A functional description of the execution scheduler is given later in section 5.4.3.

Double clicking on an operation is just one of three ways to initiate algorithm execution. Algorithms also execute when a change is made to a parameter value on a parameter control panel, and when a change is made to the algorithm topology.

5.3.2 Parameter exploration

In Chapter 4, it was established that a system that supports heuristic development should offer facilities for *parameter exploration*. An environment is conducive to parameter exploration when (i) it provides a convenient way for the designer to adjust parameter values, when (ii) the system immediately responds to change in values by initiating the recalculation of results, and when (iii) the results are calculated and displayed rapidly.

Control Panels are a convenient way of specifying parameter values because they are readily accessible and easy to interact with. Figure 5.6(a) shows the range of controls available in the OpShop operations. Numeric scalar parameters can be specified as absolute values by typing the numbers into a text field box; they can also be specified in increments by engaging a slider bar.

Parameter exploration is possible in OpShop because a change to a parameter value directly initiates the execution of the operation to which it relates. The length of delay before the result is displayed depends on the complexity and number of operations that execute, and the computational power of the computer

used. The optimal processing speed would allow an update of all results within 0.25 seconds⁷ after a parameter adjustment is made.

A feature of parameter exploration is the animated display of changing results. When a user holds the mouse button on the arrowed part of a slider bar, a series of parameter values is generated in quick succession. Each of these new parameter values causes new results to be calculated. If successive results are updated quicker than the eye can perceive, the changing result appears to be animated. The effectiveness of the animation is clearly dependent on the speed of the host computer. With the Mac LC on which this development was carried out, true animation is only possible for simple operations such as *Threshold*. Future computers offer the possibility of true animation for more computationally intensive operations and even sequences of operations. Parameter exploration could absorb any available amount of computational power!

5.3.3 Topology exploration

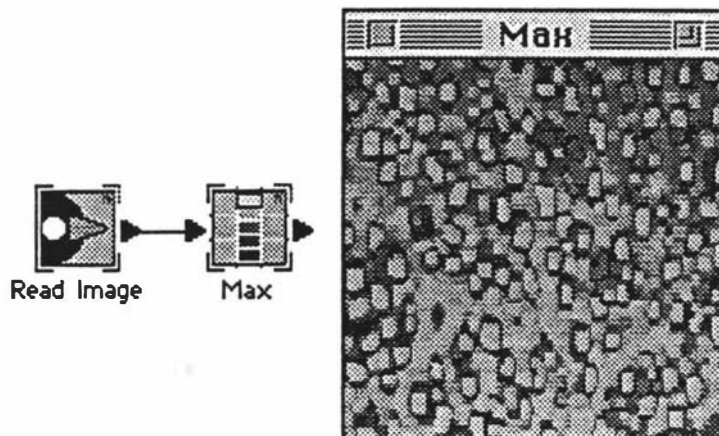
The dynamic properties of OpShop not only facilitate interactive investigation of parameter values but also of algorithm topology. An OpShop algorithm is represented by a data flow graph; the topology of that graph specifies the network of data paths interconnecting constituent operations. Topology exploration refers to a facility that allows a designer to readily adjust an algorithm topology and to rapidly view the effect of that adjustment. The purpose of the exploration is to find a data flow network that performs a prescribed imaging function.

OpShop has two interface facilities that makes topology exploration possible; direct engagement style of interaction and an execution scheduler that is responsive to user generated changes to algorithm topology. Direct engagement allows data flows to be made and broken rapidly. In immediate response to a change in topology, the OpShop execution scheduler causes the affected operations to recalculate and display results that are consistent with the adjusted topology.

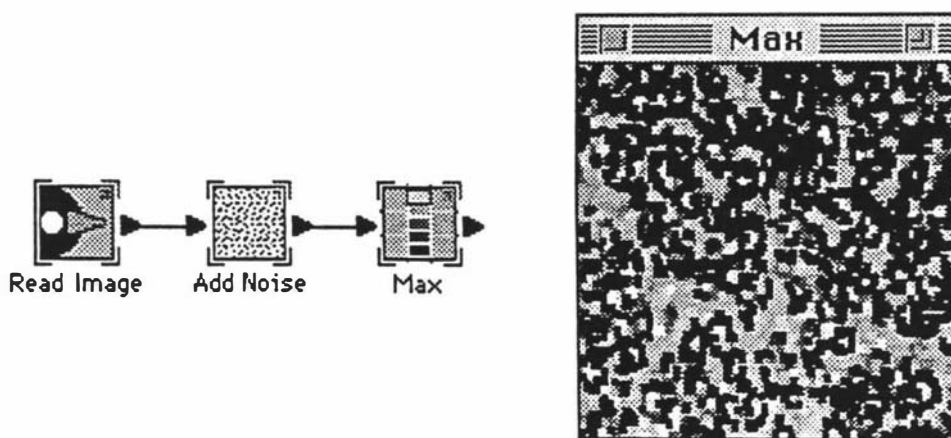
The sequence of networks illustrated in Figure 5.17 illustrates topology exploration. The aim of the sequence is to investigate the performance of the Max

⁷ Goodman and Spence (1978) performed tests to investigate the effect of system response time on the ability of a human user to point the cursor to a randomly located target on the screen. They observed the user had little difficulty performing the positioning task for system response times less than 0.25 seconds. However, the user's ability steadily worsened for response times greater than this quarter second threshold. This observation suggests that the user starts to lose the ability to judge the effect of hand movement on the cursor when the cursor response is more than 0.25 seconds, but that the user perceives continuous cursor response for delays less than 0.25 seconds.

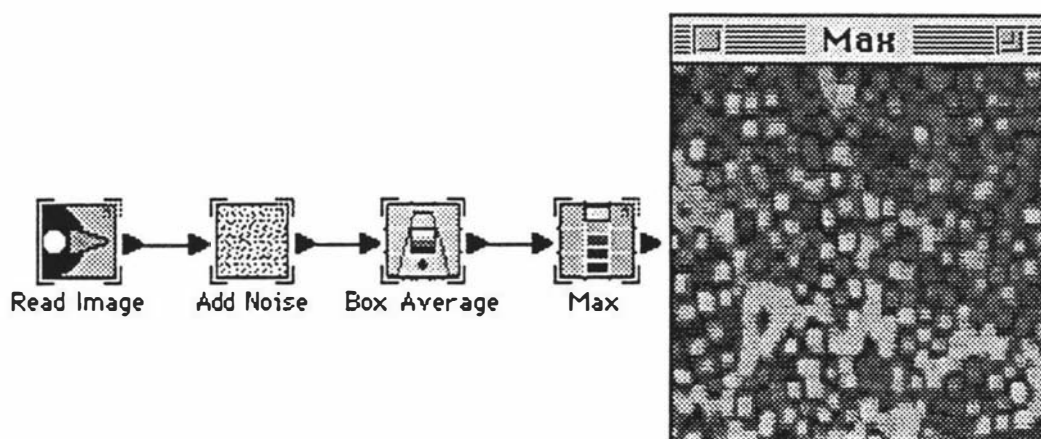
filter in the presence of noise. In the three frames of the sequence, (a) shows the Max filter applied to the test image, (b) shows the response of the Max filter to noise superimposed on the test image, and (c) illustrates the same response when the noise is first smoothed before the application of the Max filter.



(a) A Max filter is applied to a test image.



(b) The noise sensitivity of the Max filter is investigated.



(c) The introduction of a low-pass filter reduces the adverse effect of noise on the Max filter.

Figure 5.17: The successive refinement of an algorithm is an exercise in topology exploration.

The sequence of experiments can be considered a succession of refinements to the algorithm. A key task in the refinement of an algorithm is the comparison of results of successive iterations in order to see if an improvement is actually gained. A more effective way to perform a comparative study is to construct a composite algorithm where each parallel branch represents a successively refined topology (see Figure 5.18).

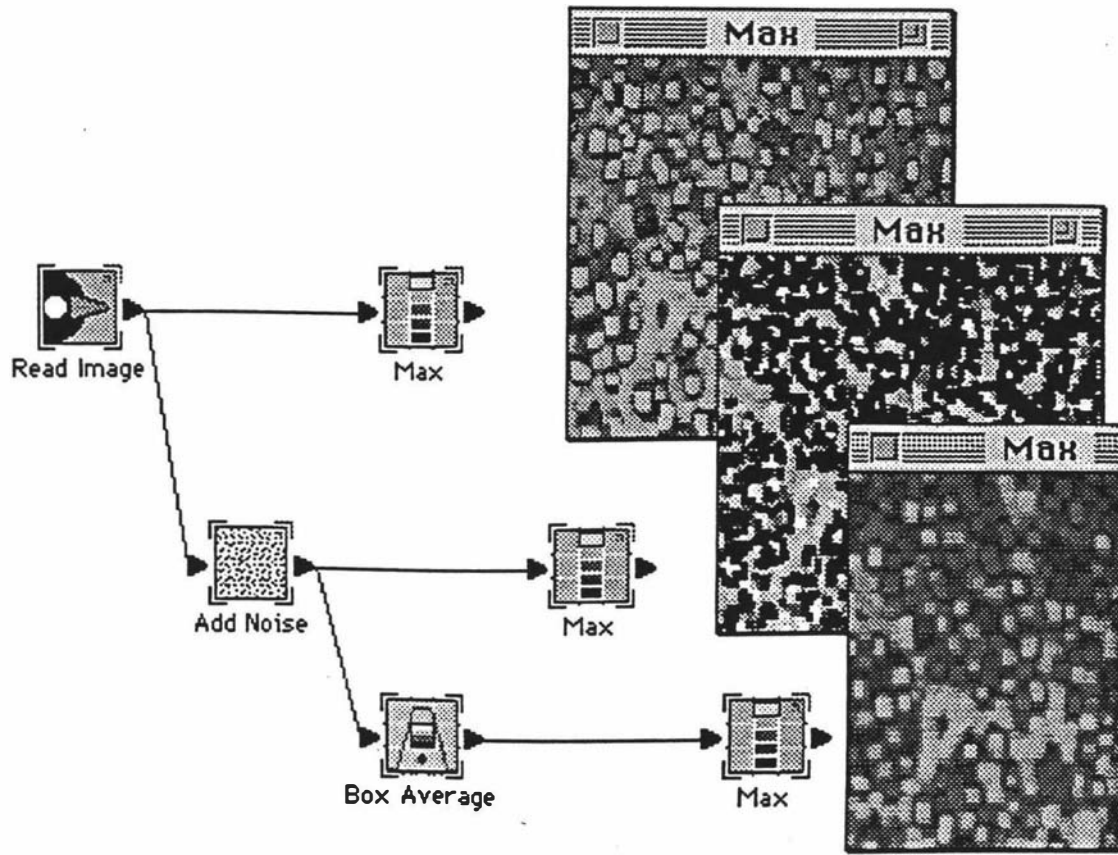


Figure 5.18: The construction of an algorithm with parallel data paths is equivalent to the three experiments of Figure 5.17.

5.4 The OpShop software design

This section discusses the implementation details of the OpShop software. The first sub-section outlines the reasons for choosing THINK C as the programming language. The following subsections detail key parts of the implementation, namely (i) the data structures for the operations, algorithms, and subflows; and (ii) the execution scheduler.

5.4.1 Why THINK C?

The decision to use THINK C as the programming language for the OpShop system resulted from the consideration of a series of three questions.

A high-level programming language or an "end-user" programming tool, such as Hypercard? From the beginning of the implementation, it was decided that (i) the program was to start as a clean slate and that (ii) the programming environment chosen should not lock the development into an unnecessarily rigid framework or user interface style. In essence, the programming environment had to offer sufficient generality to allow the coding of different interface approaches. *Hypercard* (Apple Computer, 1987; Goodman, 1990) was considered too restrictive because it required a program be organised as a linked set of cards. The *Prograph* (Pietrzykowski & Matwin, 1984) visual programming language was another programming environment considered for the implementation. *Prograph* was not chosen because it was a new and untried environment that posed a risk of not being sufficiently comprehensive. At the time, *Prograph* was still an interpreted language, and its compiler had not been released. This was seen as a undesirable limitation because efficient performance would be needed to properly demonstrate parameter and topology exploration. At the time the implementation of OpShop was started, only a high-level general purpose programming language was able to produce code for the implementation of innovative interfaces that could run efficiently; so the decision was made to use such a language.

An object oriented or procedural language? This choice hinged on whether an object-oriented or a functional decomposition approach was more appropriate for the system (Booch, 1986). An object-oriented design is generally easier to implement with an object-oriented language and similarly a functional design with a procedural language. The reasons for adopting an object-oriented approach were compelling. Firstly, graphical user interfaces are composed of graphical objects (sometimes called widgets) and so naturally suit an object-oriented design. Secondly, in OpShop, image processing operations are not just functions that return a result when provided with the required input parameters: they are objects composed of many entities including: an icon, a display window, a parameter window, a context sensitive menu, and a collection of input and output terminals. A software object is able to combine all the associated parts of an operation and its calculation functions into a single cohesive package (Cox, 1986); high cohesion within software modules is generally considered a hallmark of well designed software (Sommerville, 1989).

To C or not to C...? It is not enough to choose an object-oriented programming language solely based on language features; it is also important to examine the total programming environment that comes with the language. The four environments available to the author at the time of embarking on the software implementation were: THINK C, THINK Pascal, MPW C++, and MPW Object Pascal. The MPW environments were disregarded because they required much longer to compile and link code than the THINK counterparts. The main strengths of the MPW environments were that they provided powerful tools for version control and included an extensible editor. However, the lengthy time which these environments needed to compile and link software projects were strong disincentives for choosing them despite their support of powerful

programming tools. Both THINK environments offered a comprehensive object-oriented class library, fast compile and link times, and good interactive source-level debuggers. The C version was preferred because facilities for low-level data manipulation, which are often required in imaging software, were provided as standard features of the language. THINK C implements a competent subset of the C++ language, including the main object-oriented features of objects, classes, and inheritance (Wegner, 1987).

5.4.2 Data structures

Operations, algorithms, and subflows are implemented in software as run-time data structures called *objects*. Objects are defined in C source code by templates called *classes*. A `class` structure is very much like a `struct` data structure in the C programming language, but the key difference is that a class includes *methods*⁸ (functions) and *instance variables*. Instance variables define the state of an object, while methods provide an interface through which the instance variables can be accessed and modified. The class of an operation includes variables that define its icon, display and control parameter windows, context-sensitive menu, and lists of input and output terminals. Class methods of an operation include those to redraw its icon at a specified position, save and read its parameters to a data file, toggle the selection state, and to perform the image processing calculation.

Operation structures

An OpShop operation can be described with the use of two complementary views. The visual parts of an operation are best described by a graphical representation augmented with textual annotation to indicate the associated instance variable names, as shown in Figure 5.19. The non-visual parts of the operation are best viewed in pseudo-code, as shown in Figure 5.20.

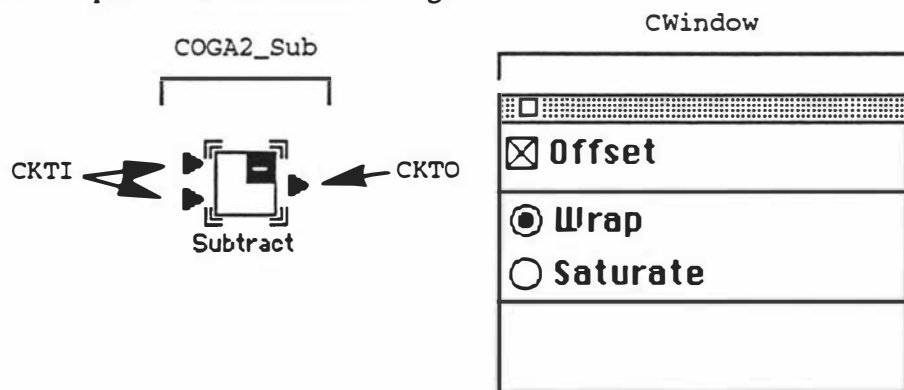


Figure 5.19: Object names for the components of a typical OpShop operation.

⁸ Though the implementation of objects in THINK C is based on C++, most of the terminology comes from Smalltalk. The THINK C terms, *method* and *instance variable* are equivalent to the C++ terms *member function* and *data member*.

```

COGA2_Sub
  Instance Variables
    CList InputTerminalList
    CKTI T1
    CKTI T2
    CList OutputTerminalList
    CKTO T1
    CWindow controlWindow
  Methods
    ExecuteMainOp()

```

Figure 5.20: The structure of the *COGA2_Sub* operation.

Figure 5.19 and Figure 5.20 together describe the function of the Subtract operation. The graphical representation indicates that the Subtract operation is an instance of the *COGA2_Sub* class and that it has two input terminals (*CKTI*) and a single output terminal (*CKTO*). The pseudo-code reveals that the input terminals are named T1 and T2 and the output terminal is named T1. The pseudo-code also indicates that the Subtract operation has a method called *ExecuteMainOp()*, which returns the difference of two input images. The control window is itself an object, called *CWindow*. It should be noted that class names are indicated in bold type and are prefixed with the letter 'C' while method names are followed by 'O'.

The display window and result buffer objects are absent from the above classes because they are part of an output terminal object (*CKTO_2D_Byte*) rather than an operation object (e.g. *COGA2_Sub*). The pseudo-code for an output terminal for a 2D array indicates the presence of a result buffer object. However, the display window object is still absent.

```

CKTO_2D_Byte
  Instance Variables
    CPixmap resultBuffer

```

The declaration of the display window object occurs in the superclass of the *CKTO_2D_Byte* class, namely the generic 2D output terminal object:

```

CKTO_2D
  Instance Variables
    CWindow displayWindow

```

The *CKTO_2D_Byte* class inherits the declaration for the display window from its superclass using the object-oriented mechanism of *inheritance*. Inheritance is used extensively in the OpShop source code to maximise code reuse and hence minimise the amount of original code. Figure 5.21 shows the inheritance tree for terminal objects the OpShop software. A generic terminal (*CKTG*), which exists at the top level, defines properties common to all

terminals, such as, a triangular icon and a pointer to its related operation. The two specific cases of the generic icon are defined, an input (CKTI) and an output (CKTO) operation. These icons in turn have special cases to accommodate different data types.

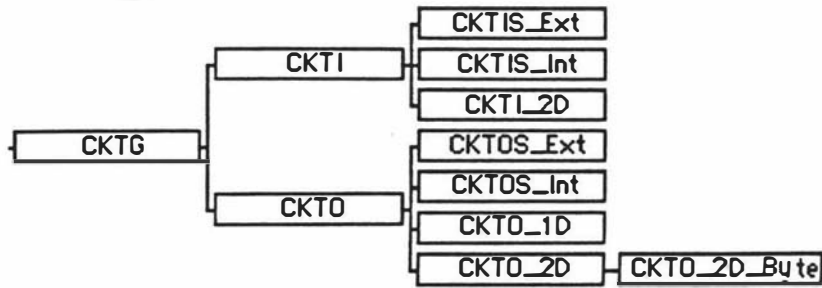


Figure 5.21: The inheritance tree for terminal objects in the OpShop software.

Algorithm structures

An OpShop algorithm is represented internally as a doubly linked graph of terminal objects. This data structure is equivalent to the visual representation of the algorithm. Figure 5.22 compares the visual representation of a simple algorithm with the corresponding pseudo-code for the algorithm. The links between terminal objects are indicated in both views.

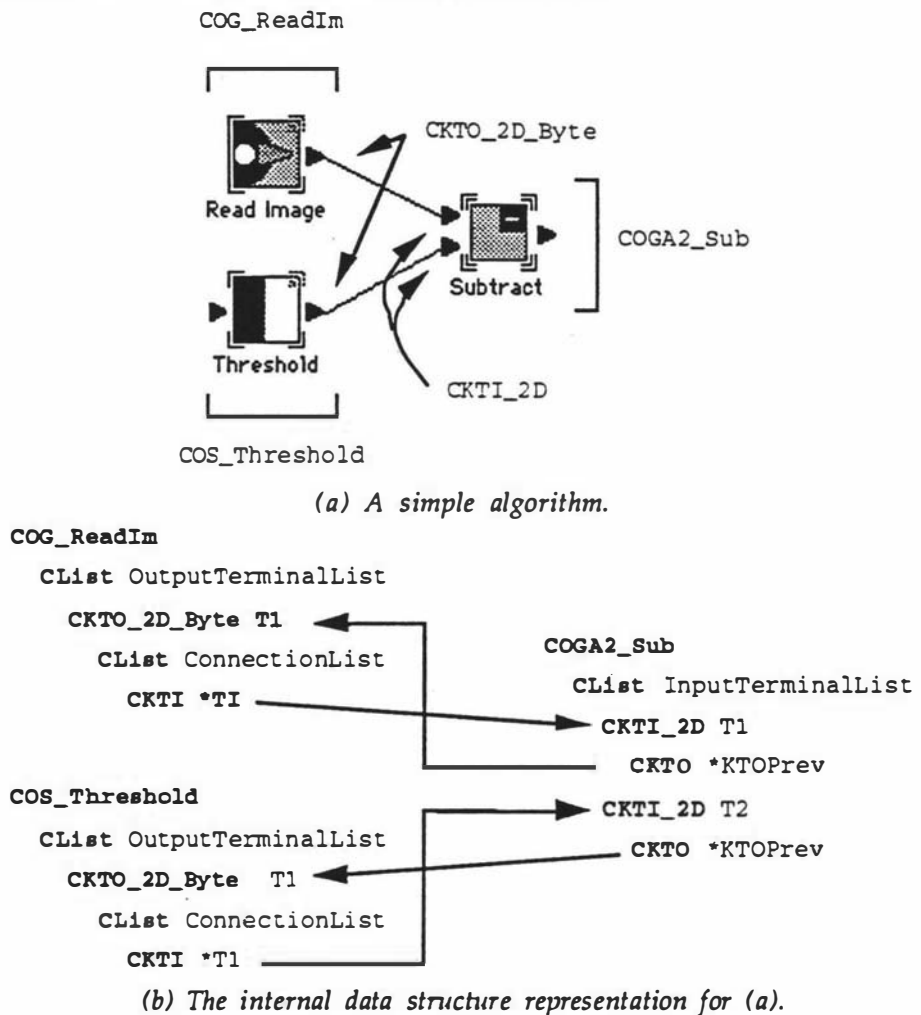
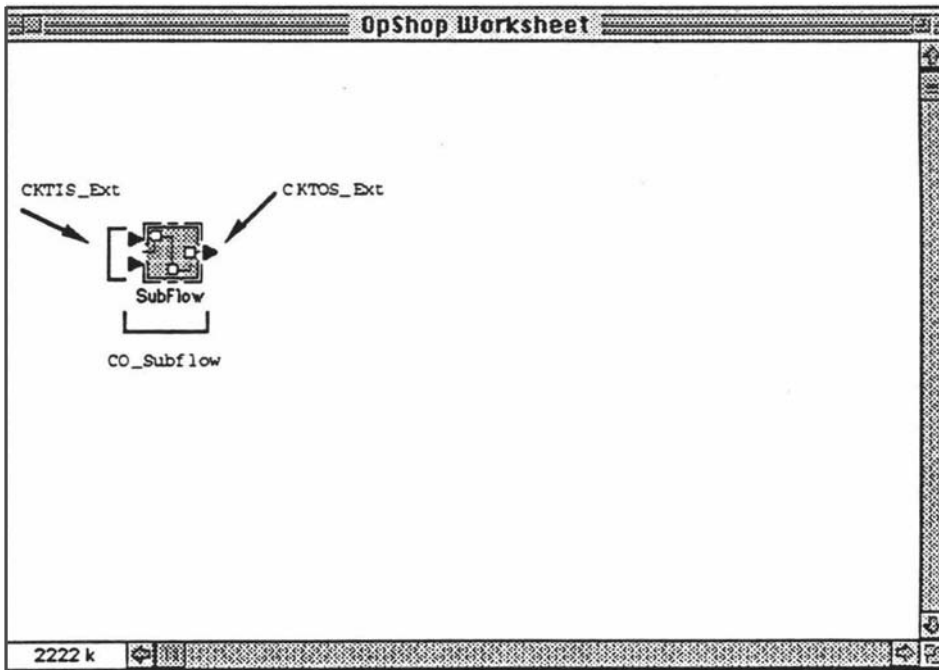


Figure 5.22: The internal representation for the topology of an algorithm is implemented by pointers of the data terminal objects.

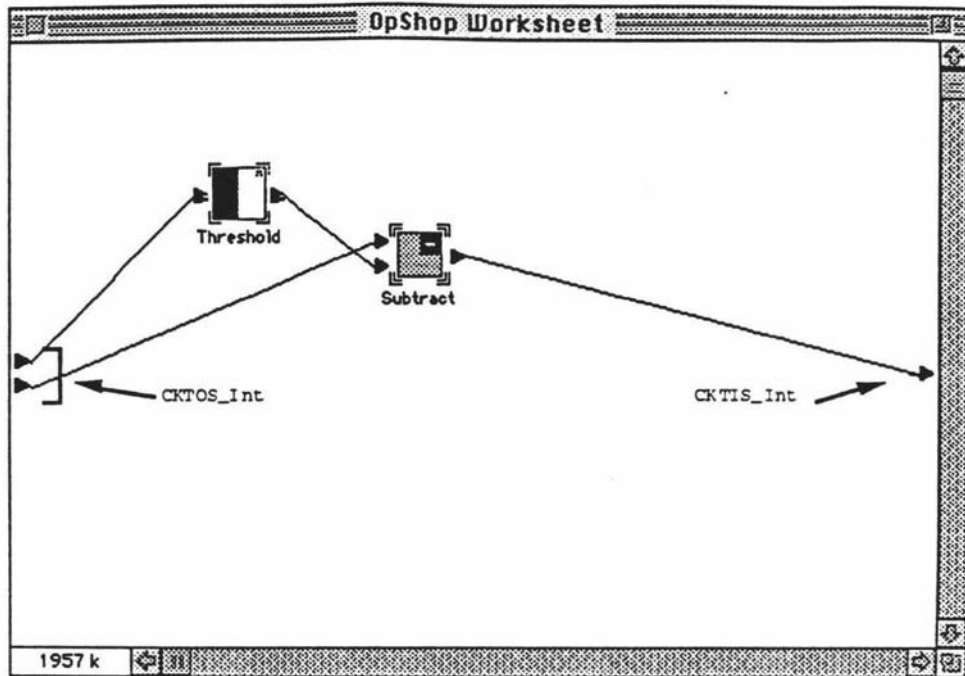
A doubly linked structure is used instead of a simpler singly linked structure because communication between operations travels in both directions. Messages to recalculate are propagated forward, hence the need for forward references. When an operation does recalculate, input data is obtained by reading the data located in the previous operation's result buffer, hence the need for a reverse reference. Doubly linked lists are also needed because messages are passed in both directions as part of the graphical update procedure; when an operation is dragged to a new position, the terminals must also redraw the data flow paths to reflect the new positions of the icons.

Subflow structures

As mentioned in section 5.2.4, a collection of operations can be grouped into a single logical unit called a subflow. Subflows appear in the parent flow as a single operation, although its behaviour is functionally identical to the collection of operation it represents. The key data structures in a subflow are the two pairs of terminals that import and export data to and from the parent algorithm. In Figure 5.23, the external terminals of these pairs (`CKTIS_Ext` and `CKTOS_Ext`) appear on the vertical sides of the SubFlow icon, while the complementary internal terminals of these pairs (`CKTOS_Int` and `CKTIS_Int`) appear on the left and right internal edges of the subflow document.



(a) Names for external subflow terminals



(b) Names for internal subflow terminals

CO_Subflow

CList InputTerminalList

CKTIS_Ext anExtITerm1

CKTOS_Int *itsCKTOS_Int ↔ CKTOS_Int

CKTIS_Ext anExtITerm2

CKTOS_Int *itsCKTOS_Int ↔ CKTOS_Int

CList OutputTerminalList

CKTOS_Ext anIntOTerm1

CKTIS_Int *itsCKTOS_Ext ↔ CKTIS_Int

(c) Object structure for the subflow shown in (a) and (b).

Figure 5.23: The structure of a subflow operation.

The pair of terminals used to import data into a subflow are: **CKTIS_Ext** and **CKTOS_Int**, and similarly the pair of terminals to export data are: **CKTIS_Int** and **CKTOS_Ext**. The terminals in a complementary pair communicate with each other through an internal bi-directional link as indicated in the pseudo-code shown in Figure 5.23(c). The two types of output terminals associated with subflows, **CKTOS_Int** and **CKTOS_Ext**, do not buffer data like other output terminals, so option-clicking on them does not bring up a display window.

5.4.3 A data-driven execution scheme

An execution scheme determines the order in which operations execute. The development of an execution scheme is a non-trivial task when an algorithm is

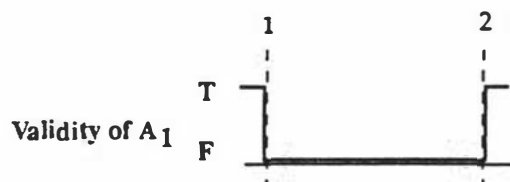
expressed in data flow form. Two schemes exist for the execution of data flow structures, which are broadly classed as *data-driven* and *demand-driven* (Docker, 1989). Of the two, the data-driven scheme better facilitates the interaction required in the exploratory approach to algorithm development. When a user makes a parameter change to an operation or a topological change to an algorithm, the new results should be rapidly displayed. Furthermore, if the change is calculated for the entire algorithm, then the user is able to construct local episodes of *what-if* investigations involving the whole algorithm. In contrast, the central answer given by a demand-driven execution scheme is for the question *what-is-the-result-here?* The data-driven execution scheme inherently answers this question because intermediate results are automatically kept up to date. In a demand-driven system, an operation does not execute unless it is explicitly requested by a user or if its result is needed by another operation; even then, it executes only if its result needs updating. The amount of computation that a system must perform is minimised by demand-driven execution. However, this benefit is not sufficient to offset the drawback that such execution does not support the exploratory *what-if* style of investigation.

Scheduling

The data-driven execution of an OpShop algorithm is best illustrated by diagrams that are similar to logic traces that are used to describe the function of digital electronic circuits. These traces indicate the validity of data for each flow during algorithm execution. The most elementary example is the execution of a single operation that has no input terminals; such an operation will always execute immediately upon request. Figure 5.24 shows the logic trace for the execution of the Read Image operation.



(a) Output label for result of the Wedge operation.



(b) Validity trace for the data at A1.

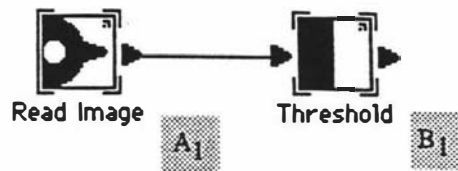
- 1: ResultIsInvalid() message is generated and the validity of A1 is set to False, which is indicated by a black tip on the terminal icon (see Section 5.2.2).
- 1-2: The Read Image operation executes.

- 2: NewResultIsAvailable() message is generated and the validity of A_1 returns to True.

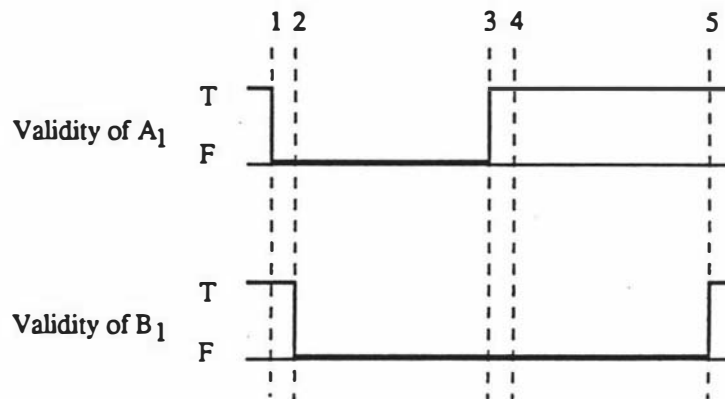
(c) Interpretation for (b)

Figure 5.24: A validity trace for the data-driven execution of a single Read Image operation.

The next example shows a slightly more complicated algorithm involving one data flow (see Figure 5.25). The important difference between this and a single operation is that: (i) Read Image operation sends the ResultIsInvalid() message to the Threshold operation before any calculations are made; this message causes Threshold to set its output validity to False, and that (ii) the Threshold operation executes only upon receipt of the NewResultIsAvailable() message, which the Read Image sends after completing its calculation.



(a) Output labels for a simple algorithm: A_1 and B_1 .



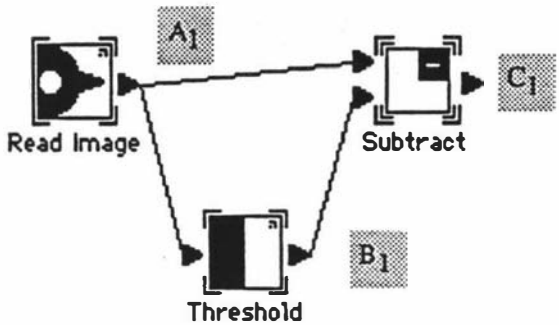
(b) The validity trace for the data at terminals A_1 and B_1 .

- 1-2: ResultIsInvalid() message is sent from Read Image to Threshold.
- 2-3: The Read Image executes.
- 3-4: NewResultIsAvailable() message is sent from Read Image to Threshold.
- 4-5: The Threshold operation executes.

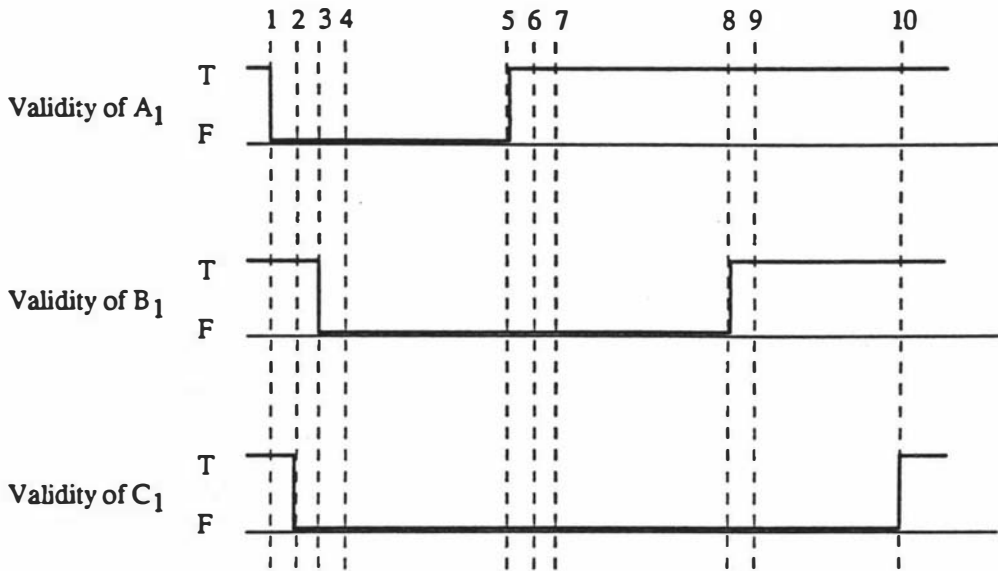
(c) Interpretation for (b).

Figure 5.25: A validity trace for the data-driven execution of a simple algorithm.

The situation where two data flows merge into a single operation demonstrates the enforcement of the data-driven scheduling rule: "an operation can execute only if all its inputs are valid". The validity trace of Figure 5.26 shows that although the Subtract operation receives the `NewResultIsAvailable()` message from Read Image before Threshold does, it does not execute immediately. At the instant Read Image informs Subtract that a new result is available, the second input of Subtract (on B₁) is still invalid. According to the above scheduling rule, it is not until Threshold updates B₁ that Subtract can execute.



(a) Output labels for an algorithm with parallel data paths: A₁, B₁, and C₁.



(b) Validity trace for algorithm with parallel data paths. Note that all outputs are set to an invalid state before any recalculation occurs.

- 1-2: `ResultIsInvalid()` message is sent from Read Image to Subtract.
- 2-3: `ResultIsInvalid()` message is sent from Read Image to Threshold.
- 3-4: `ResultIsInvalid()` message is sent from Threshold to Subtract.
- 4-5: The Read Image operation executes.

- 5-6: `NewResultIsAvailable()` message is sent from `Read Image` to `Subtract`. The `Subtract` operation does not execute because its input data supplied by `B1` is invalid.
- 6-7: `NewResultIsAvailable()` message is sent from `Read Image` to `Threshold`, which begins to execute immediately.
- 7-8: The `Threshold` operation executes and sets its output validity high when it completes.
- 8-9: `NewResultIsAvailable()` message is sent from `Threshold` to `Subtract`. The `Subtract` operation executes immediately because both of its inputs (on `A1` and `B1`) are now valid.
- 9-10: The `Subtract` operation executes.

(c) Interpretation for (b).

Figure 5.26: A validity trace for the data-driven execution of an algorithm with parallel data paths.

In the example given in Figure 5.26, `Read Image` has two outputs connected to it: `Subtract` and `Threshold`. Messages are sent from `Read Image` to its neighbouring operations in the same order that these operations were connected to `Read Image` when the algorithm was constructed. For an algorithm with parallel paths, the order of execution may vary depending on the order that the paths were connected. However, the net effect of the algorithm is determined only by the data-dependency.

Side-effects

Ackerman (1982) claims that an execution schedule can be deduced simply from the data dependencies of a data flow algorithm provided all the operations execute without side-effects. An algorithm with hidden data dependencies is an algorithm with side-effects. Shared access of global variables between functions in C is a common example of a side-effect:

```
void foo( int x )
{
    s = x*2; /* s is a global variable */
}
```

When the function `foo()` changes the value of the global variable `s`, any other function that also uses `s` is affected. Implicit data dependency exists between any two functions that can modify a common global variable. Absence of global variables and careful control of the scope of variables eliminate this type of side-effect. Another common side-effect is changing a parameter passed into a function by reference:

```
void foo( int *x )  
{  
    *x = 3.141; /* The modification of *x is a side-effect */  
}
```

A function calling `foo()` may be unaware that the value of `*x` has changed and so be unaware of the implicit data dependency that exists between `foo()` and itself.

The OpShop software avoids side-effects by providing a local result buffer for every operation. Contents of this buffer can be changed only by using its related operation and so discounting any possibility of side-effects.

5.5 Feasibility of continuous interaction

This section reports an assessment of the feasibility of achieving *continuous interaction* for OpShop running on current and future computing hardware. The assessment involved a comparison of the execution times of benchmark tests to the 0.25 second criterion that marks the attainment of continuous interaction. It should be noted that the criterion for continuous interaction is less demanding than the criterion for true visual animation, which requires algorithms to complete execution in less than 0.10 seconds (Card *et al.*, 1983).

Commands representative of those found in typical algorithms were chosen for the benchmark tests. `Invert` and `Threshold` are point operations that process a single image to yield a single image result; `Subtract` is a point operation that processes two images to give a single image result; the `Max` filter and `Box Average` filter are local area operations; `Skeletonize` is a complex multipass operation. Results of the benchmark tests are graphed in Figure 5.27. In this graph, the operations are ranked in order of computational complexity. The timings are averaged over five trials. The top and centre graphs show results of OpShop operations running on computers that are considered to be very and moderately powerful by 1992 standards: a 25 Mhz Mac Quadra 900 and a 16 MHz Mac cx. On the Quadra, all the point operations returned results within 0.25 seconds, even for the larger 256 x 256 images. The local area operations also calculated within 0.25 seconds except for the 256x256 images. For `Skeletonize`, only the smallest image was calculated within the desired response time, while the largest image required more than one order of magnitude longer than the optimal time. On the Mac cx, only point operations calculated within the quarter second guideline and then for only the two smaller images. As a control experiment, the equivalent VIPS operations were run on the Quadra. No significant disparities were observed between OpShop and VIPS timings. Differences in times are mainly due to auxiliary pixel intensity conversions performed by OpShop, which are required to ensure that images are displayed with conventional grey scale mappings (0 = black and 255 = white).

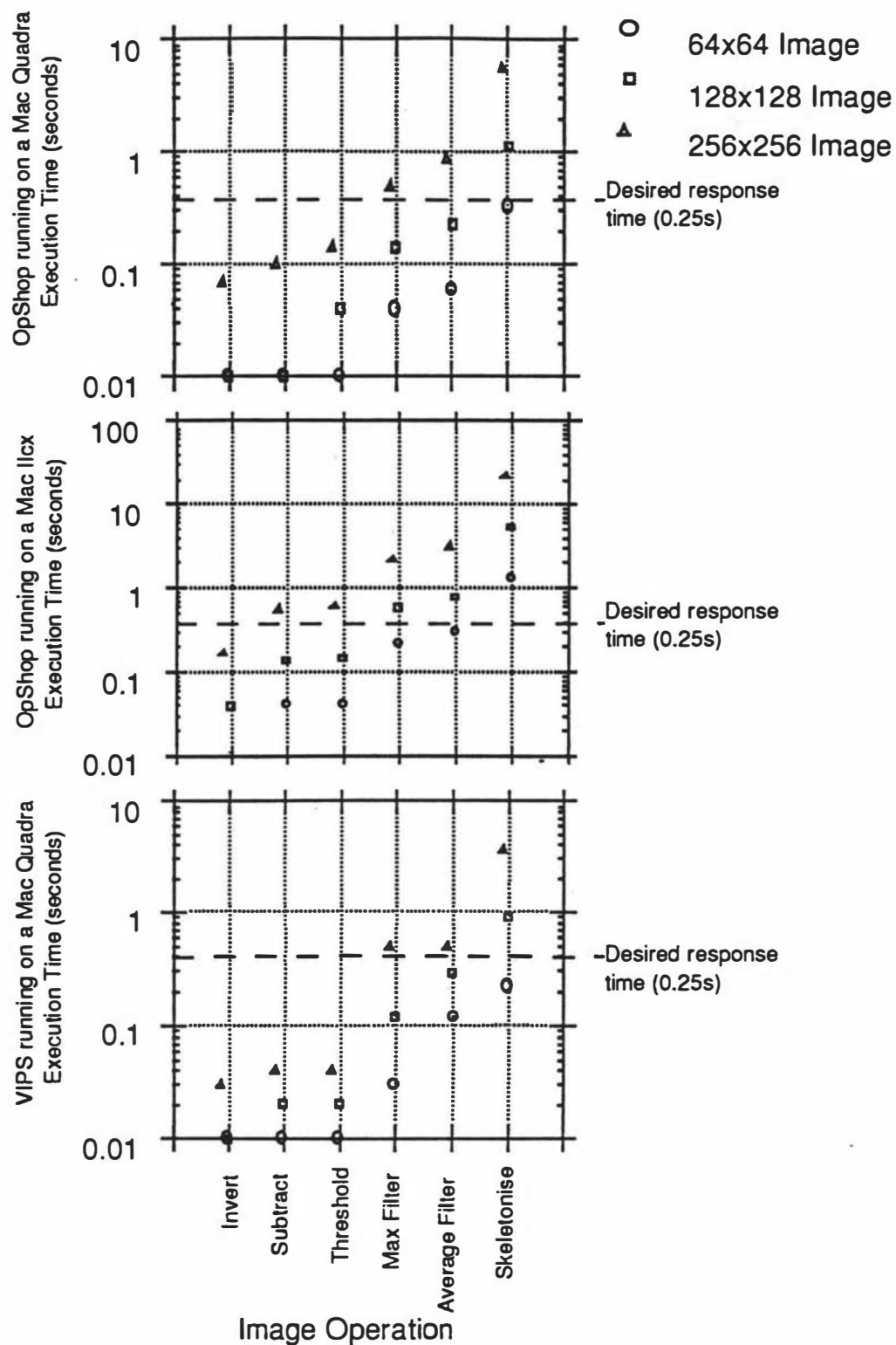


Figure 5.27: Comparison of response times for OpShop and VIPs.

This feasibility study shows that continuous interaction is attainable for point operations processing moderate sized images on current hardware. The Mac Quadra performs point operations on smaller images one order of magnitude more quickly than is needed for continuous interaction. Very rapid processing was attained on the Quadra because the entire contents of an image can be stored in the CPU's high speed data cache. Continuous interaction was attained by the local area operations for all but the largest test image.

In practice, continuous interaction is required for the execution of whole - or at least partial - algorithms rather than single operations. It can be inferred from the benchmark results that continuous interaction is achievable for simple algorithms composed of point operations, or even local area operations, provided image sizes are kept small. Workstations today use 25 MHz processors, but it is expected that speeds of 500 MHz or more will be common place by the end of the decade (Prince & Salters, 1992); these machines should provide sufficient performance to satisfy the computational requirements of the dynamic exploration techniques discussed in this thesis.

5.6 Summary and conclusions

The OpShop software facilitates the development of imaging algorithms by providing an environment that encourages the heuristic investigation of algorithm variations. In this chapter, the key interactive features of the OpShop package were described in three parts:

- a description of the graphical elements that denote operations and algorithms in the OpShop visual language;
- a demonstration of the user actions needed to interactively develop algorithms;
- a description of the software data structures and the execution scheduler.

The graphical elements representing operations, algorithms, and subflows of an iconic visual language have been described. Colour icons to depict operations, interconnected sets of icons to depict algorithms, and grouped subsets of icons to depict subflows were used to form an iconic language. An important aspect of OpShop is direct engagement interaction, because it provides easy variation of calculation parameters and algorithm topology, and rapid calculation and display of results; these characteristics are conducive to an exploratory approach to algorithm development.

The data structures of the operations, algorithms, and subflows in OpShop have also been described. The software for OpShop is written in an object-oriented programming language because a graphical interface iconic visual language is well suited to an object-oriented design decomposition. The fundamental element of an algorithm is a single operation, and so these are coded as object classes. An operation object is composed of many other objects, including an icon, a text label,

and a list of input and output terminals. The input and output terminals contain links to terminals of other operations, and it is this network that defines the algorithm topology. Communication between operations inside and outside a subflow is interfaced by two special pairs of terminals whose specific function is to pass data into and out of a subflow.

Benchmark tests running on current computing hardware demonstrate that continuous interaction can be achieved for simple algorithms - that is, algorithms composed of point operations - provided that the processed images are sized 256x256 or smaller. Continuous interaction can also be achieved for algorithms that include local area operations, provided that the processed images are 128x128 or smaller. The one or two orders of magnitude of improvement in speed required for continuous execution of general algorithms are expected to be well within the capabilities of the hardware offerings of the present decade.

The OpShop package represents a tangible implementation of the concepts of algorithm development and user interaction discussed in Chapters 2, 3 and 4. The key features of the system combine to provide an interactive environment that supports a user in the task of heuristic algorithm development. These key features are: the representation of an algorithm as an iconic data flow network, the adjustment of algorithm parameters and topology in a direct manipulation interaction style, and the rapid update of results. Rapid updates are possible through the use of a responsive execution scheme. Demonstrations of how the OpShop package can be used to simplify algorithm development are given in the following chapter.

Chapter 6

OpShop Examples

This chapter demonstrates by example how the OpShop visual language facilitates the development of image processing algorithms. The development of four algorithms using both a typical command line system and using OpShop are described. The contrast between the two approaches highlights how the features of OpShop work towards the simplification of the development task.

6.1 Introduction

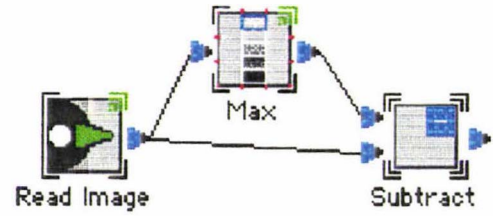
OpShop is a visual language designed to facilitate the development of imaging algorithms. The way OpShop achieves this objective is best demonstrated by example. The four examples presented in this chapter are representative of common interaction tasks in the development of an imaging algorithm. They demonstrate how OpShop helps the user to (i) visualise and understand multi-threaded algorithm topologies, (ii) explore parameter values, (iii) refine algorithm topologies, and (iv) choose between alternative algorithms. It is suggested that these demonstrations, by implication, validate the design philosophy of OpShop presented in Chapter 4.

6.2 Data flow view of an algorithm

A simple example of a multi-threaded algorithm is one which calculates local maxima. The equivalent algorithms in the VIPS and OpShop languages are shown in Figure 6.1.

```
load/raw anImage.img im
box max im im_max
let im_out = im
subtract im_out im_max
```

(a)



(b)

Figure 6.1: Algorithms to calculate local maxima of an image: (a) in VIPS and (b) in OpShop.

These algorithms contain two data threads: one for the original image and another for the local maximum of the original image. The pixels in the two images will only have the same intensity values at points of the local maxima in the original image. These points can be found by taking the difference between the two images; a zero pixel in this difference image will denote a local maximum in the original image. Note how the parallel threads are shown explicitly in the OpShop language but just implicitly in the VIPS listing.

6.2.1 Colour classification

The algorithm demonstrated in this section classifies the pixels in an image on the basis of colour. The colour image used for this example, Figure 6.2, is derived from an aerial photograph of a part of the Massey University campus in autumn. Infra-red film was used, so the trees, grass fields, and bush appear a red colour, and the tar-sealed roads and roofs of buildings appear a grey colour. The pixels in this image fall into four classes: (i) grass paddock, (ii) trees and bush, (iii) concrete or tar-seal structures, and (iv) unknown.

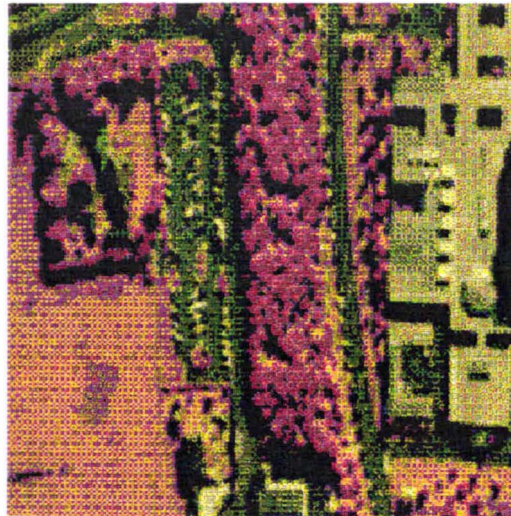


Figure 6.2: Near infra-red image of the Massey University campus in autumn.

The technique of colour classification assumes that the colour values of pixels within a class are similar, and pixels between classes are dissimilar. Figure 6.2 exhibits this property of colour distinction, and so the pixels can indeed be classified on the basis of colour.

The technique used in this example is a two dimensional extension of the technique of finding an optimal threshold by analysing a histogram (see Section 2.2.3). In the case of colour, a histogram indicates the frequency of occurrence of colour values. Pixels corresponding to the different classes in the image, map to distinct clusters in the histogram space. Typically a histogram of colour values is three dimensional, but for the image shown in Figure 6.2, the histogram needs only to be two dimensional because the green and blue channels contain essentially the same information. Therefore, the blue channel has been omitted, and the resultant two-dimensional histogram is shown in Figure 6.3(a). The clusters in this histogram are identified by a peak detection algorithm and labels are assigned to each cluster as shown in Figure 6.3(b). These labels can then be reassigned to the original image to give a classified image. The original histogram, labelled histogram, and classified images are shown in Figure 6.4.

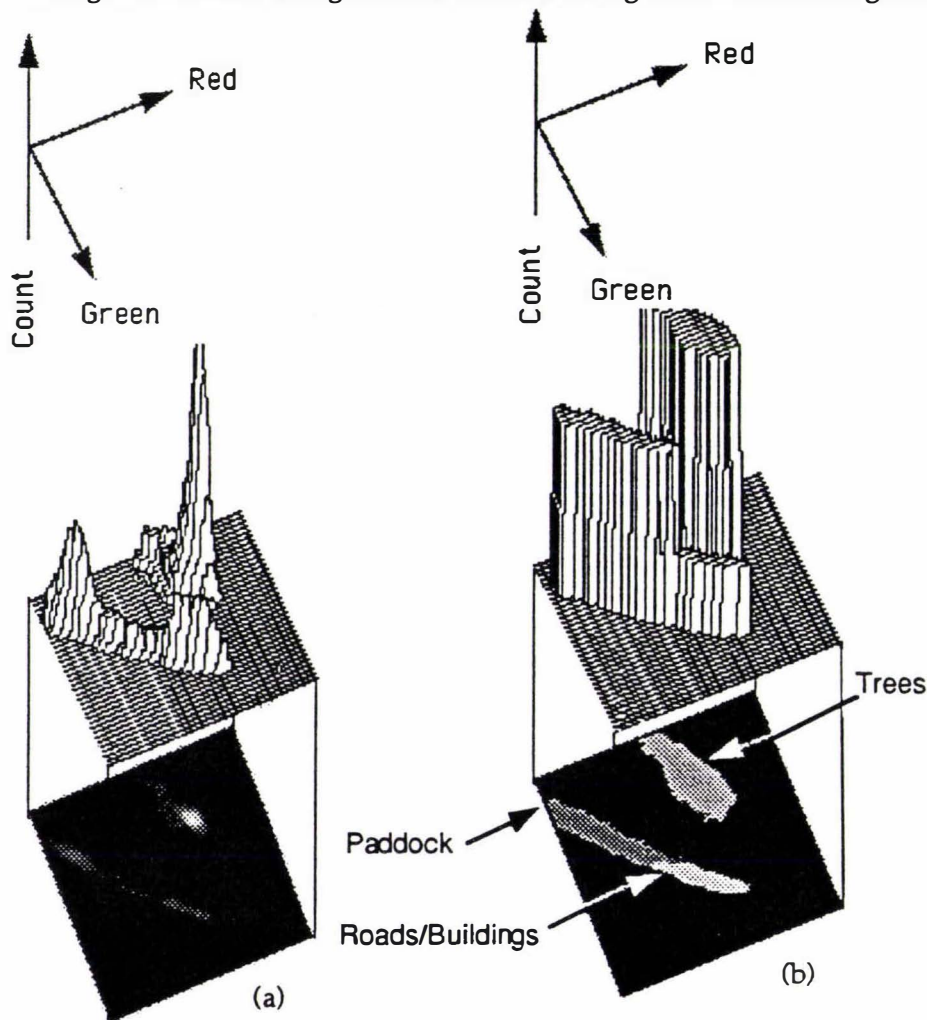


Figure 6.3: Colour histograms for the image shown in Figure 6.2: (a) measured histogram, (b) labelled histogram. The wire-frame and 'image' representations of the histograms are equivalent.

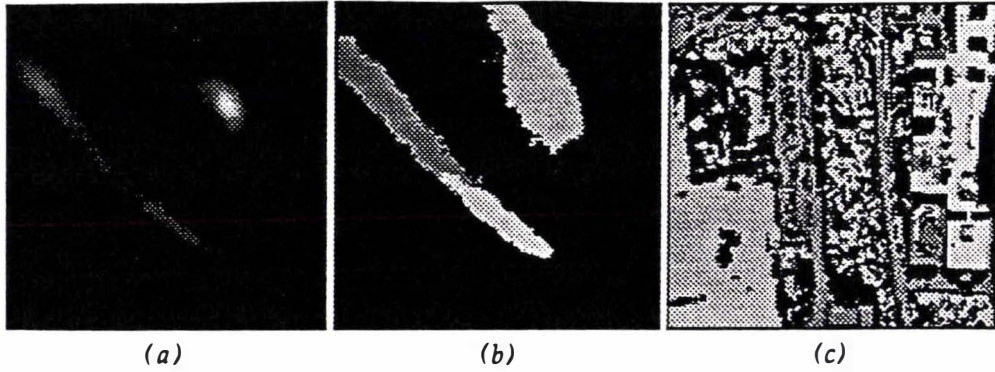


Figure 6.4: Intermediate and final results of a colour classification algorithm: (a) histogram of input image, (b) labelled histogram, and (c) classified image.

6.2.2 A command line implementation

A VIPS algorithm to implement the colour classification procedure just described is shown below:

```
! Preliminaries
!
load/program label.vip label ! Read VIPS label program.
load/raw massey_r.img red    ! Get red channel of aerial
                                ! image.
load/raw massey_g.img green  ! Get green channel of aerial
                                ! image.
!
! Perform colour classification
!
h2 red green t_lut2d          ! Create 2d histogram.
convert t_lut2d lut2d         ! Long to byte image conversion.
box average lut2d t           ! Smooth noise.
let lut2d = t
expand lut2d                  ! Linear stretch the contrast.
                                ! to make function visible.
run label lut2d               ! Label the clusters.
lookup red green lut2d classified ! Reassign labels to
                                ! original image.
```

Figure 6.5: A VIPS algorithm for colour segmentation, which is based on the detection of clusters in a colour histogram.

The preliminary operations prepare the variables and data that are used in the actual processing. The VIPS variables `red` and `green` store the red and green components of the colour image. A two dimensional histogram (`t_lut2d`) of the red and green components is calculated by the `h2` operation. This histogram is converted to a byte image (`lut2d`) by the `convert` operation. The operations `box average` and `expand` prepare the histogram for the labelling procedure which is carried out by the VIPS program `label`. Finally, the classified image (`classified`) is created by mapping the original red and green values (in `red` and `green`) to the labelled values (in `lut2d`) with the `lookup` operation.

The linear textual representation of the algorithm shown in Figure 6.5 obscures the perception of parallel data paths, which are indicated in Figure 6.6. A user would find it difficult to gain an understanding of how this algorithm works without actually reading the algorithm and following the progress of each variable. Obscurity of parallel data paths often hinders comprehension of an algorithm whose logic is inherently parallel, such as the one presented.

```
! Perform colour classification
!
h2 red green t_lut2d
convert t_lut2d lut2d
box average lut2d t
let lut2d = t
expand lut2d
run label lut2d
lookup red green lut2d classified
```

Figure 6.6: The parallel data paths of the images: red, green, (shown) and lut2d (follows the text).

6.2.3 OpShop implementation

In contrast to the command line representation, multi-threaded data paths are represented explicitly in the iconic data flow language of OpShop. Figure 6.7 shows clearly the flow of the original red-green colour pixels to the Create Hist2D operation, which creates the two-dimensional histogram, and also to the Lookup 2D operation, which maps the original values to the extracted class labels. Not only are the paths shown clearly, but also the operations involved in each path. In contrast to the command line representation, the graphical representation of an algorithm used in OpShop is highly visual and explicitly shows the logic of data transformations.

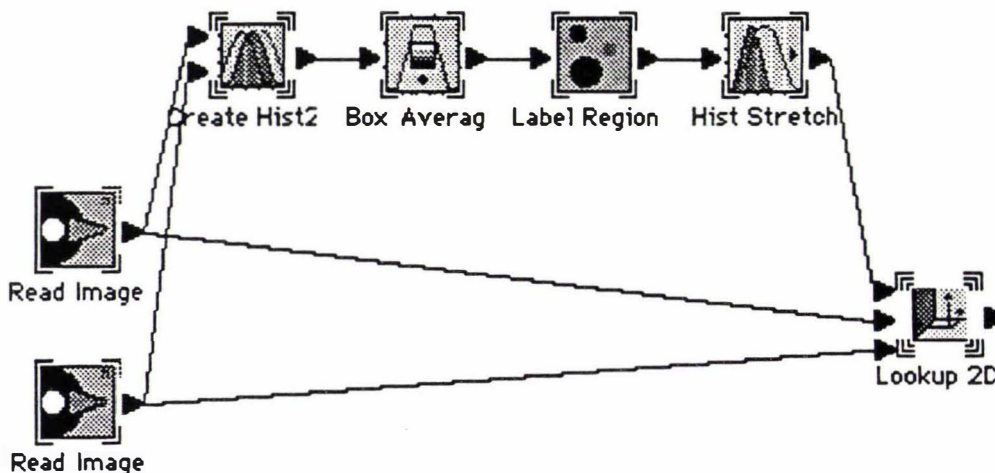


Figure 6.7: Parallel data paths of the colour classification algorithm are shown explicitly in the OpShop representation.

6.3 Parameter exploration

Most operations have associated parameter values that control the calculation of the result. The values of these parameters are often critical to the effectiveness of the operation. For example, when a threshold operation is used to segment an object from its background, the upper and lower thresholds must be chosen to include the entire intensity range of the object, but none of the background. If the two threshold values do not bound the intensities of the object, but instead include intensities of the background, then a threshold operation will yield a segmented region that includes regions of the background. However, if the threshold range is too narrow so as to exclude some of the object intensities, then the segmented region will correspondingly exclude regions of the object. For segmentation and many other operations, the choice of parameter values can be critical to the effectiveness of the operation. The actual parameter values are not only critical, but are often different for each new set of input data. For these reasons, a facility for the exploration of parameter values is essential to the heuristic development of algorithms.

Another aspect of parameter exploration is that a parameter does not affect just the operation to which it belongs, but all subsequent operations that depend on the result of the adjusted operation. Therefore, a parameter value can be considered to be a parameter of an algorithm, and not just a local operation. Therefore a change in a parameter value in one operation may often make necessary the re-execution of a string of subsequent operations. A facility that supports parameter exploration will automatically perform the necessary recalculations, as discussed in Section 5.4.3.

Yet another aspect of parameter exploration is that several parameters in an algorithm may be interdependent, so that a change in the value of one parameter may necessitate changes in other parameters. Such an interdependency occurs between the brightness and contrast controls for a cathode ray tube. When interdependencies exist in an algorithm, the search for a satisfactory balance of parameter values may require repeated execution.

The following example demonstrates the rapidity and ease with which parameter values can be specified in OpShop, and the way in which the system brings all the results up-to-date in response to any parameter adjustments.

6.3.1 The Abingdon Cross benchmark

The imaging task used for this example is based on the Abingdon Cross benchmark, which was used by Preston (1989) to compare relative processing performances of image processing systems. The benchmark transformation is shown in Figure 6.8. The input data for the benchmark algorithm consists of a test image that contains a cross figure that is superimposed with Gaussian distributed intensity noise. The aim of the benchmark is to extract the medial axis, or skeleton, of the cross figure. The critical parameter for generation of the cross is the signal to noise ratio.

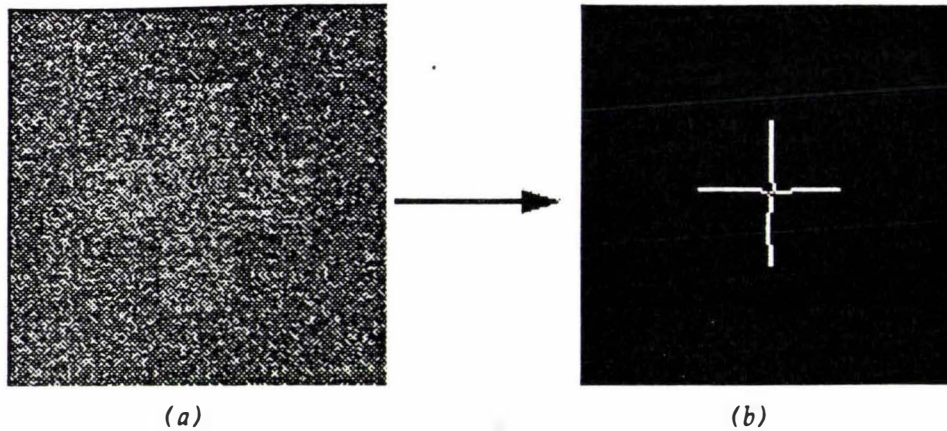


Figure 6.8: Abingdon Cross benchmark transformation: (a) noisy cross image and (b) medial axis of the cross.

6.3.2 A command line implementation

A VIPS algorithm to perform the Abingdon Cross benchmark is:

```
! Preliminaries
!
load/program makecross.vip mc ! Puts cross into 'a'
!
! Do benchmark
!
run mc                      ! Generate cross image
                             !   background = 0, cross = 32.
noise/gaussian b 112 32    ! Generate gaussian noise with
                             !   mean= 128 and std dev = 32.
                             !   i.e. SNR = 1
add a b                    ! Superimpose noise into cross
box average a across 9     ! Average filter, box kernel= 9
threshold across 128       ! Create binary image
thin across                ! Create skeleton
```

Figure 6.9: VIPS implementation of the Abingdon Cross benchmark.

The `mc` program creates a cross figure a signal magnitude of 32. The standard deviation of the Gaussian noise superimposed on the cross image is also 32, hence yielding a test image that has a signal to noise ratio (SNR) of one. Three operations are now applied to the test image to extract the medial axis. First, a `box average` filter with a box size of 9×9 is used to smooth the high-frequency intensity variations. This smoothed image is then thresholded at an intensity of 128, which is a value half way between the average background and the average cross signal. Finally, the `thin` operation calculates the skeleton of the binary image. A typical sequence of image data is shown in Figure 6.10.

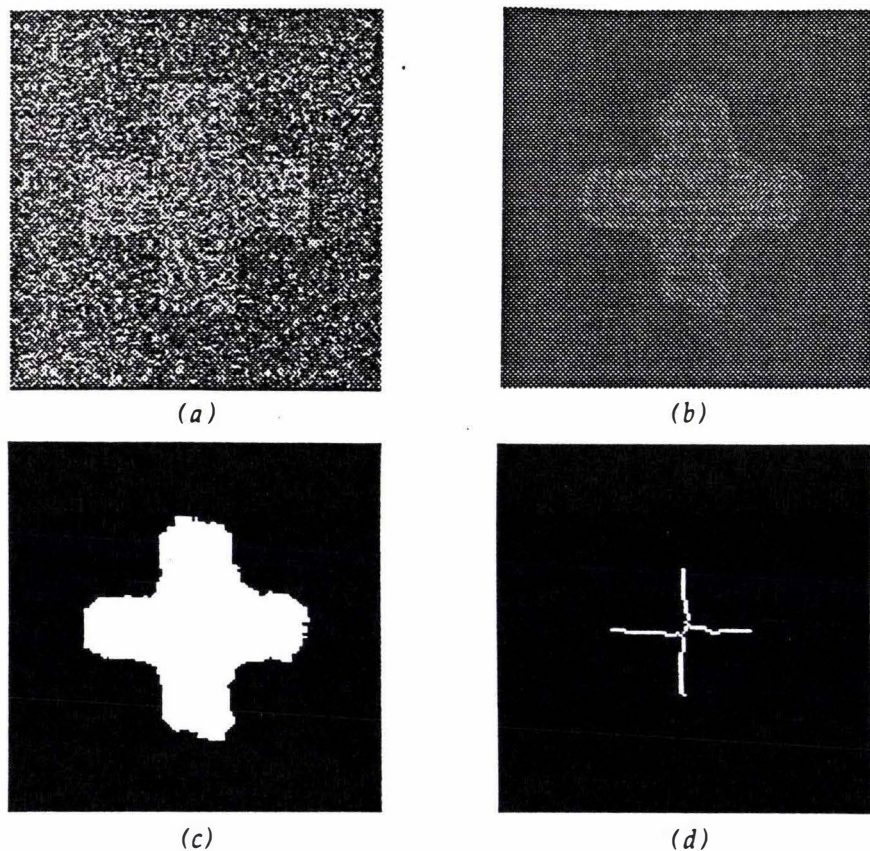


Figure 6.10: Snapshots of image data as it passes through the Abingdon Cross benchmark algorithm: (a) input image, (b) smoothed image, (c) thresholded image, and (d) the skeleton image.

The algorithm shown in Figure 6.9 illustrates two aspects relevant to the selection of parameter values. Firstly, the two parameters involved, the signal to noise ratio and the box size for the average filter, are inter-dependent: the lower the signal to noise ratio, the more severe the required smoothing. Secondly, a poor choice for the size of the box filter typically results in dramatic errors in the skeleton image, as shown in Figure 6.11. In this example, the smoothing is inadequate and consequently leads to a segmented cross with a hole. Since the skeletonising operation must accommodate the hole, the resultant skeleton contains a highly visible loop in its left arm.

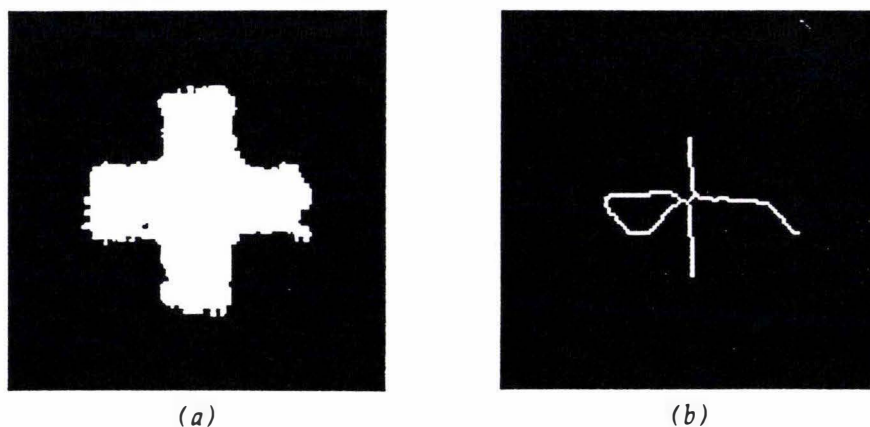


Figure 6.11: Extraction of a deformed skeleton due to insufficient noise suppression: the (a) binary image with a hole and (b) the deformed skeleton.

A facility that will allow rapid experimentation with parameter values is clearly helpful in the development of an algorithm that has inter-dependent parameter values and critical parameter settings. However, command line interfaces do not lend themselves to repeated execution of algorithms because of the amount of retyping involved. Experimentation with the box size parameter requires the re-execution of three operations for each trial, while experimentation with the noise value increases this number to five. Command line recall facilities significantly relieve the amount of retyping; but even then, a small amount of re-petition would still cause experimentation to be tedious. Retyping can be reduced by grouping the repeated operations into a program file. Re-execution would then require only retyping a single command. However, this practice may lead to a proliferation of small program files that have little meaning outside the context of immediate the experiments. Furthermore, each time the user wants to change a part of the repeated sequence, the relevant program file must be edited. Clearly, parameter exploration with a command line system is a tedious process.

6.3.3 An OpShop implementation

OpShop supports parameter exploration by a providing an easy and convenient method for the specification of parameter values, as shown in Figure 6.12. With these controls, the specification of a new parameter can be performed by a simple mouse click. If the value of the standard deviation of the noise is adjusted, then the Add Noise operation will automatically re-execute, as will the Box Average, Threshold and Skeletonize operations, in order to maintain the integrity of the output data with respect to the current parameters (see Section 5.4.3). The ease with which the standard deviation (hence SNR) can be changed, and the rapidity of execution of the algorithm, is conducive to the exploration of that parameter.

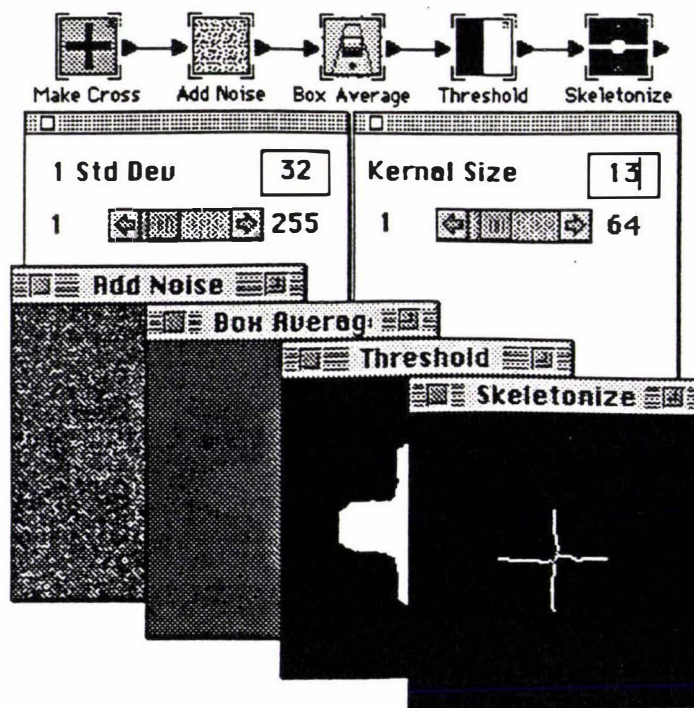


Figure 6.12: The OpShop algorithm for the Abingdon Cross benchmark.

In OpShop, the exploration of combinations of inter-dependent parameter values can also be conveniently performed because the controls for interdependent parameters can be concurrently accessed. For example, the controls for both the standard deviation parameter or the box size parameter shown in Figure 6.12 can be assessed with the same effort. The grid of results shown in Figure 6.13 were generated by accessing the parameters in exactly this way. The generation of these results required the specification of just ten different parameter values; two for the initial parameter values and one for each additional trial thereafter. The ten trials were performed quickly and easily because the control panels provided good accessibility to the parameter values.

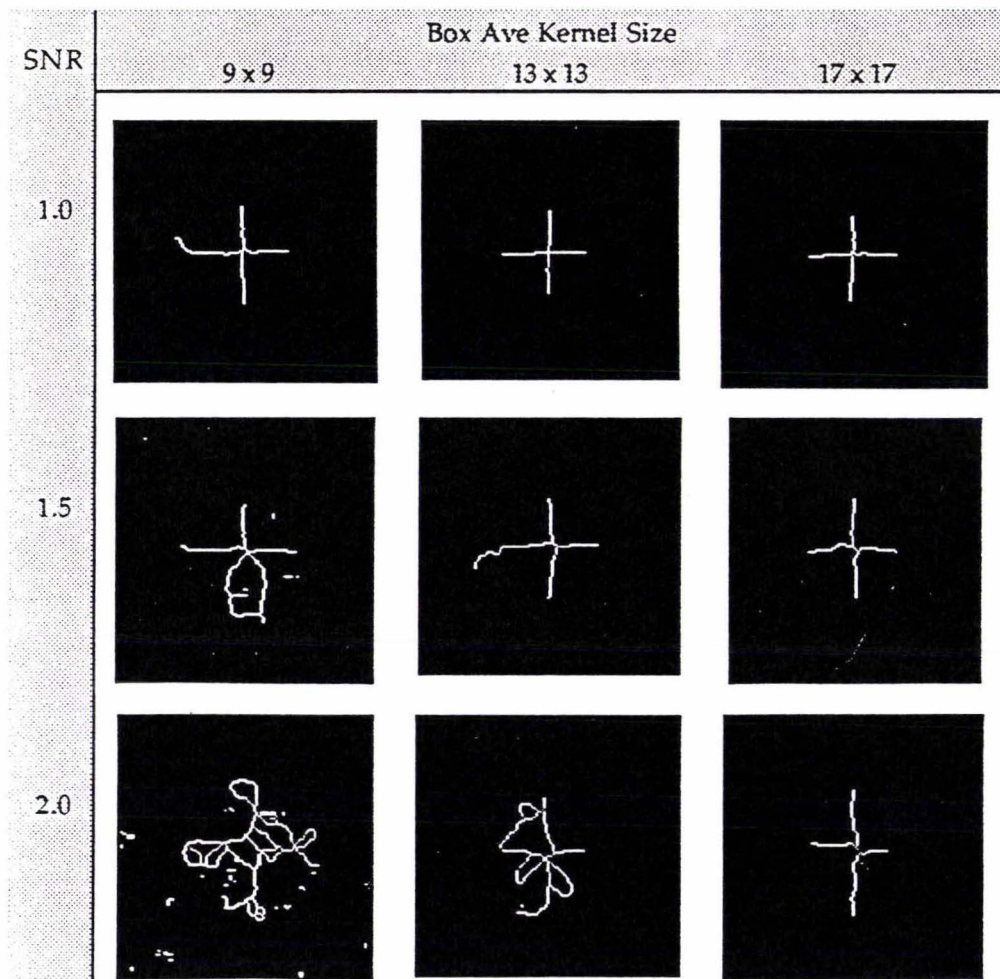


Figure 6.13: Results for noise sensitivity tests.

6.4 Topological exploration

6.4.1 Segmentation of a non-uniformly illuminated scene

This third example demonstrates how OpShop helps the user to define the topology of an algorithm. This definition process involves the addition and removal of operations from an algorithm, the rearrangement of the operations within an algorithm, and the generation results.

The function of the imaging task presented in this section is one of segmentation. The image shown in Figure 6.14 is of a microscope slide that displays a cross-section through axons in a sample of horse muscle tissue. The task involves the measurement of the area distribution of the axons, which are represented by the central regions of the ring structures. The rings are myelin sheaths which provide structural support for the axons. This project carried out by the author was part of a clinical study (Kennegieter, 1989) to test the hypothesis that a certain disease effects the diameters of the axons. To measure the axon area distribution, the individual axons must first be isolated from the rest of the image. This segmentation process is the focus of the example.

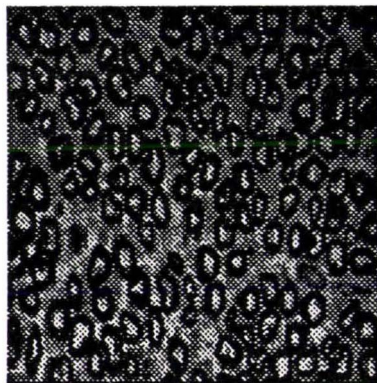


Figure 6.14: Axons in a sample of horse muscle tissue.

The segmentation of regions of axons can be performed by the following operations:

- (i) acquire a grey-scale image,
- (ii) segment rings from the background,
- (iii) clear the pixels around the border of the image to the background colour so that axons truncated at the edge of the image are filled in by the next step,
- (iv) flood the background of the binary image with an intensity equal to the value of the myelin sheath.

For ideal input data, the action of this sequence of operations is illustrated by the progression of images shown in Figure 6.15.

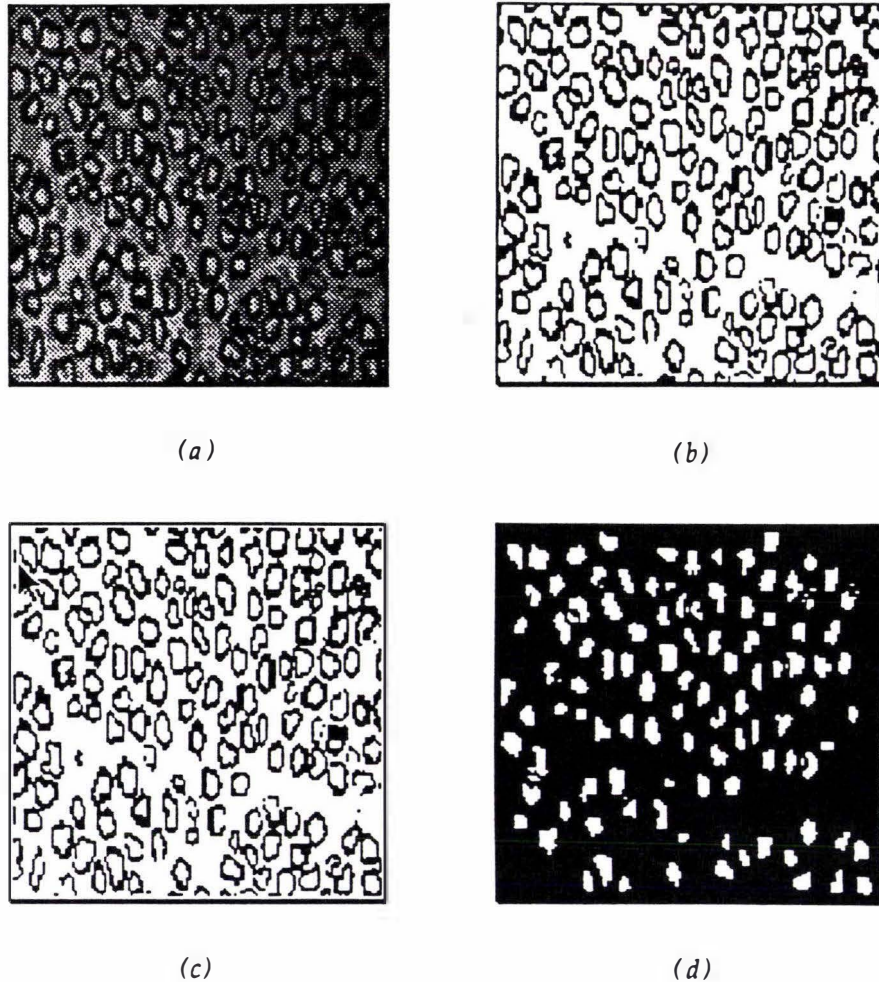


Figure 6.15: Successive transformations of the horse axon image as it passes through the segmentation process: (a) acquired image, (b) segmented image, (c) segmented image with cleared border, and (d) axon image.

In practice, image data often exhibit non-ideal characteristics that disrupt the proper operation of the segmentation process just outlined. One such non-ideal characteristic is non-uniform illumination of the scene at the time of image capture; this causes an intensity gradient to be superimposed across an image. Figure 6.14 is degraded in this way. The degradation is not immediately apparent, but is revealed when the image is segmented at different global threshold values. The sequence of segmented images at different threshold values in Figure 6.16 shows that the superimposed intensity gradient is lowest in the top right hand corner of the image and rises towards the bottom left corner.

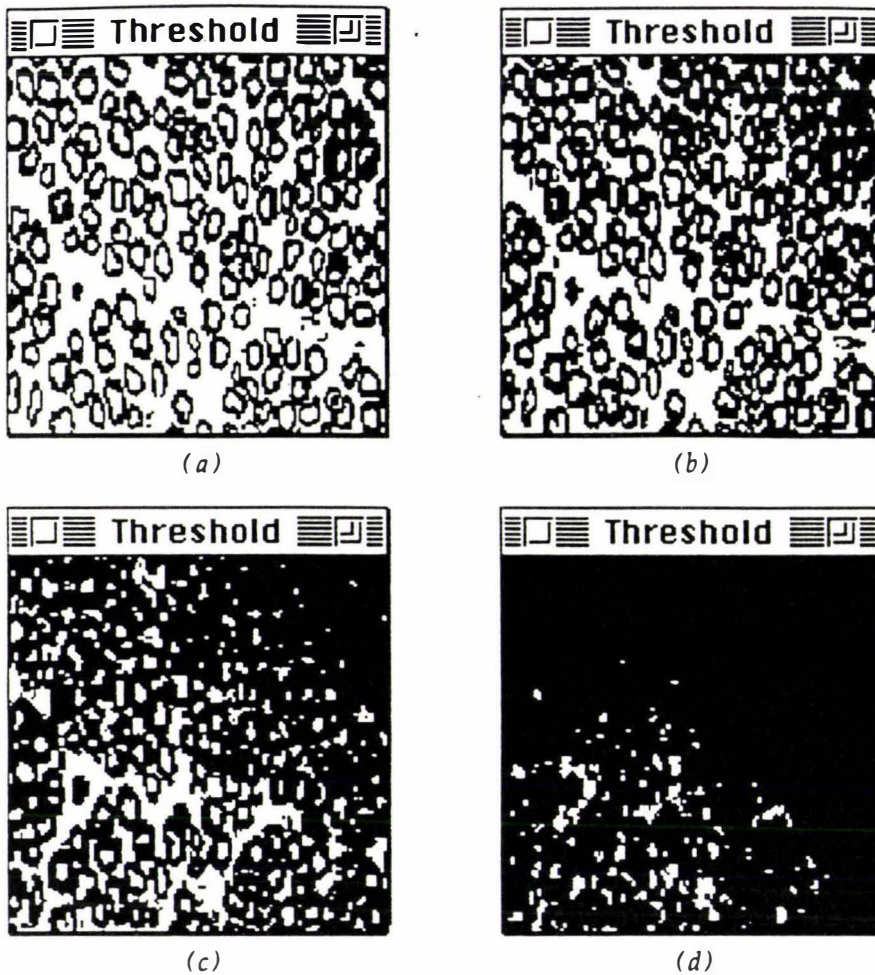


Figure 6.16: A sequence of binary images generated for different lower threshold (t_l) values: (a) $t_l = 60$, (b) $t_l = 90$, (c) $t_l = 120$, (d) $t_l = 150$.

Without compensating for the intensity gradient, the simple global thresholding algorithm leads to the generation of many spurious 'axons', as shown in Figure 6.17. The false axons represent regions where the flood-fill operation could not transform the background to black because these regions are totally enclosed by sheath pixels.

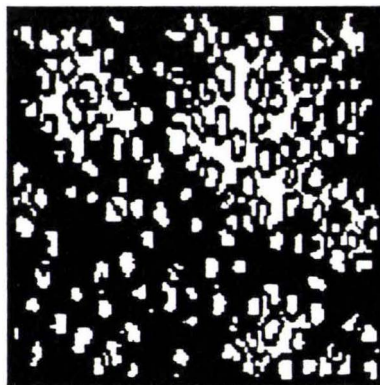


Figure 6.17: Artifacts generated because of the application of the simple segmentation algorithm to non-uniformly illuminated input images.

6.4.2 A command line implementation

For the global threshold operation to yield an accurate segmentation of the mylen sheaths, the intensity gradient must first be removed. Compensation for the intensity gradient can be achieved by the execution of additional operations prior to the application of the thresholding operation. In the VIPS algorithm, compensation is achieved by inserting additional commands before the threshold and fill operations, as shown by:

```
load/raw axons.img ax

{ Operations to compensate for non-uniform illumination }

threshold ax 70
draw box /rectangle (0 0) (255 255) /image 255 ax
fill ax 0 (10 10)
```

Figure 6.18: Proposed modification to the simple segmentation algorithm for the compensation of non-uniform illumination.

It should be noticed that the proposed modification to the algorithm in Figure 6.18 occurs in the body of the algorithm rather than at its end. Therefore any changes involving operations related to the compensation will necessitate the retyping of the threshold, draw and fill operations. The need for constant backtracking and retyping is illustrated in a trace of VIPS commands, shown in Figure 6.19, for a possible development scenario.

```
declare image (255 255) ax
load/raw axons.img ax
slice ax                                     ! Provides an interactive way to
                                           ! investigate threshold values.

threshold ax 70
draw box /rectangle (0 0) (255 255) /image 255 ax
fill ax 0 (10 10)

! At this point the intensity gradient is noticed. Therefore
! try a background subtraction method of compensation.
! Backtrack to retrieve original data I
load/raw axons.img ax
let t1 = ax                                 ! It is much quicker to retrieve the
                                           ! original data from a temporary
                                           ! variable than from disk.

box max t1 ax (16 16)
subtract ax t1 /saturate
let t2 = ax                                 ! threshold operation overwrites
                                           ! input, so it's wise to take a copy
                                           ! of it in case it's needed again.

slice ax
threshold ax 0 95
display ax
draw box /rectangle (0 0) (127 127) /image 255 ax
display ax
! Whoops, drew the box the wrong size.
! Retrieve ax, and backtrack to II
let ax = t2
threshold ax 0 95
draw box /rectangle (0 0) (255 255) /image 255 ax
display ax                                 ! Box size is correct this time.
fill ax 0 (10 10)
display ax
!
! See lots of small spurious binary regions
! This compensation method must be improved or abandoned.
! Now, I can choose between three alternatives:
! 1. Try another threshold value III
! 2. Try another max filter kernel size
! 3. Try new compensation method altogether
```

Figure 6.19: A possible VIPS trace for the development of the background compensation method.

The session progresses in successive episodes of backtracking. At point *I*, the designer must reload the original image data to begin the implementation of a compensation method. At point *II*, the designer made a mistake in drawing the border and so must replicate the `let`, `threshold`, and `corrected draw` operations to regain the same place. The first trial of the compensation method is completed at point *III*, but at this stage the user observes that the method has not worked properly. To improve the current compensation method, the user is presented with three possible alternatives; each involves a different degree of backtracking.

6.4.3 An OpShop implementation

In contrast to command line systems, OpShop is specifically designed to facilitate the experimentation of algorithm topology. The ease with which commands can be inserted into the body of an existing algorithm is illustrated by the OpShop sequence shown in Figure 6.20.

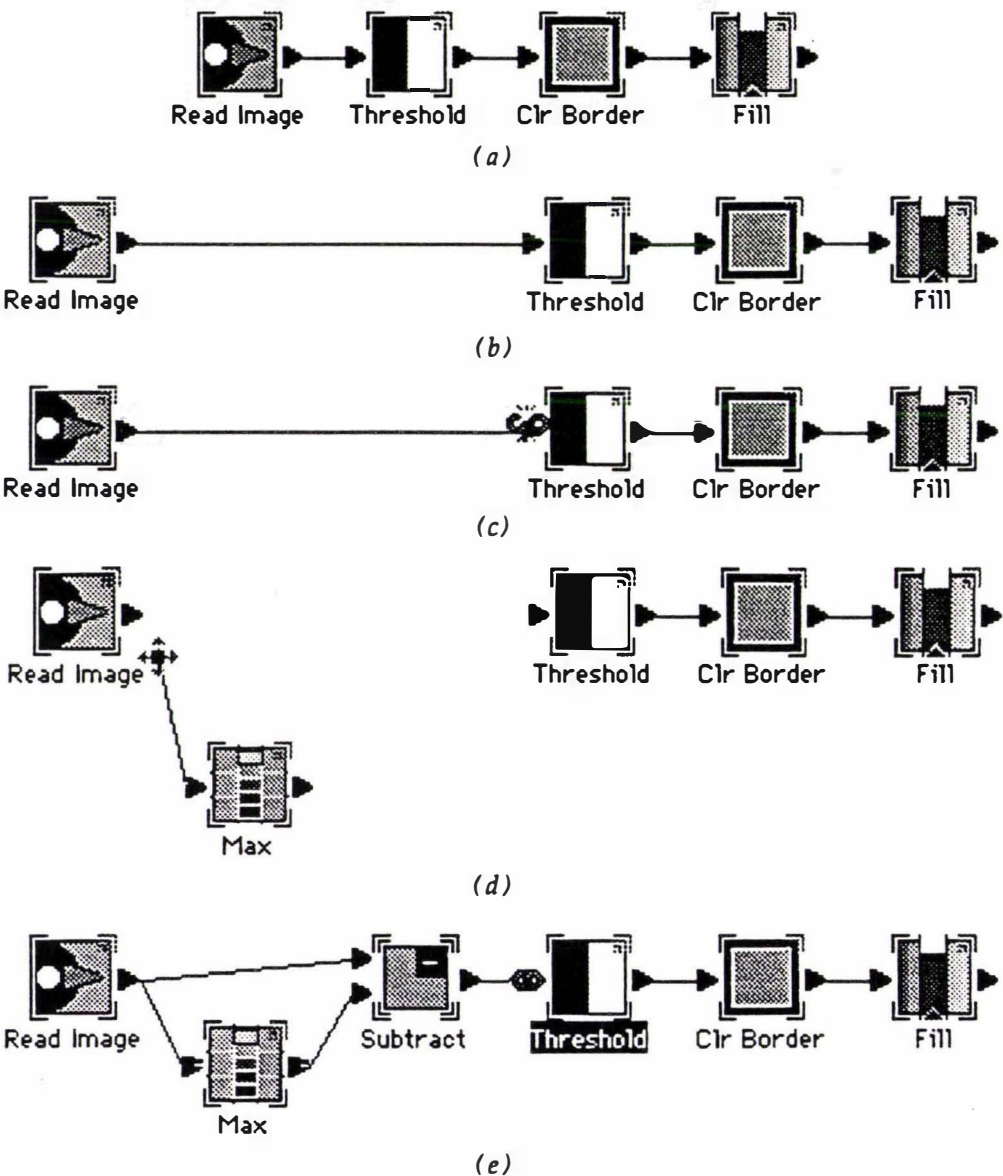


Figure 6.20: The addition of an operation into an existing OpShop algorithm.

Figure 6.20(a) shows the basic segmentation algorithm before the addition of compensation operations. In Figure 6.20(b), the Read Image operation is dragged leftwards to accommodate the operations to be added. The input data flow to the Threshold operation is broken in Figure 6.20(c). Next, a Max filter is deposited onto the whiteboard by a menu selection. This is connected to the algorithm by a dragging a rubber band line from its input to the output of the Read Image operation, as shown in Figure 6.20(d). The Subtract operation is added in Figure 6.20(e) to complete the experiment.

The iconic data flow language of OpShop makes experimentation with algorithms simple. The simplicity is largely due to the temporally persistent nature of operation objects and algorithm definition. When an operation has been deposited onto the whiteboard and incorporated into an algorithm, the topology of the modified algorithm persists until it is deliberately adjusted by a user. Furthermore, when the topology of an algorithm is adjusted, the modifications are restricted to a small part of the algorithm. In contrast, an algorithm is represented in a command language by the order in which commands are presented to the command line prompt. Therefore, to modify this algorithm, the re-ordered sequence of operations must be presented to the command line prompt; hence the need for a high degree of user input.

6.5 Choosing between algorithm alternatives

6.5.1 Generation of alternative solutions

During an algorithm development session, a designer may construct more than one possible solution. In such situations, it is desirable that the user has the capability to compare the candidate algorithms side-by-side.

The example used in this section continues the horse axons example presented in section 6.4, where two segmentation algorithms were developed; one that performed a simple global thresholding and an other that compensated for non-uniform illumination before applying the global threshold. In the development of such algorithms, a user would benefit from a facility to compare the results of the old and improved methods.

6.5.2 Command line implementation

Two aspects of command line systems that hinder the comparison of alternative algorithms are *variable reuse* and the *linear textual representation* of algorithms. A variable that stores a significant result can be overwritten before the result is recognised as being significant and stored for later retrieval. Therefore results - particularly results stored in frequently used variables - cannot be guaranteed to exist because of variable reuse. A result that has been overwritten but not saved elsewhere, can be regained only by repeating some or all the calculations that generated it in the first instance. An example of

variable reuse is shown in the VIPS listing of Figure 6.19. The result `ax`, which is the output of the `fill` operation in the seventh line, is significant because it represents the result of the simple segmentation method. However, `ax` is immediately overwritten in the following line and reused to store the original image data for the second experiment. The variable `ax` should have been saved, instead of being overwritten, so that it could be later compared with the result of the compensated algorithm. Since it was not saved, the original `ax` must be recalculated before any comparisons between the two methods can be made.

A second reason for the difficulty in comparing alternative algorithms in command line systems is that, the boundaries between algorithms, or logical parts of algorithms, are displayed implicitly when represented by text. The listing shown in Figure 6.19 is a homogeneous sequence of text that is devoid of visual cues to indicate logical relationships between commands. For instance, the sequence contains no indication that two algorithms exist, nor that the first algorithm ends on the seventh line at the `fill` operation. It should be noted that the annotation shown in the listing would normally be absent in its screen representation of an interactive development session.

6.5.3 An OpShop implementation

Alternative solutions are easy to compare in OpShop because the alternatives may be represented as parallel data threads. Variable reuse is not an issue because variables are not used. Alternative solutions are visually indicated by branching paths, as shown in Figure 6.21, which shows how judicious positioning of equivalent operations from parallel experiments aids comparative studies. Deliberate juxtapositioning of result windows further reinforces the equivalences and highlights the differences between algorithms. The alternatives can be easily created by a cut-and-paste facility⁹.

The upper thread depicts the uncompensated method for segmentation. Compensation by background subtraction is represented by the middle thread. The lower thread shows a second alternative compensation method, which uses the `Loc Hist Str` operation to normalise the intensity gradient across the image by adaptive contrast enhancement. The three result windows relate to the three threads as indicated by their respective positions.

⁹ The cut-and-paste facility has not yet been implemented, but will probably be included in later versions of OpShop.

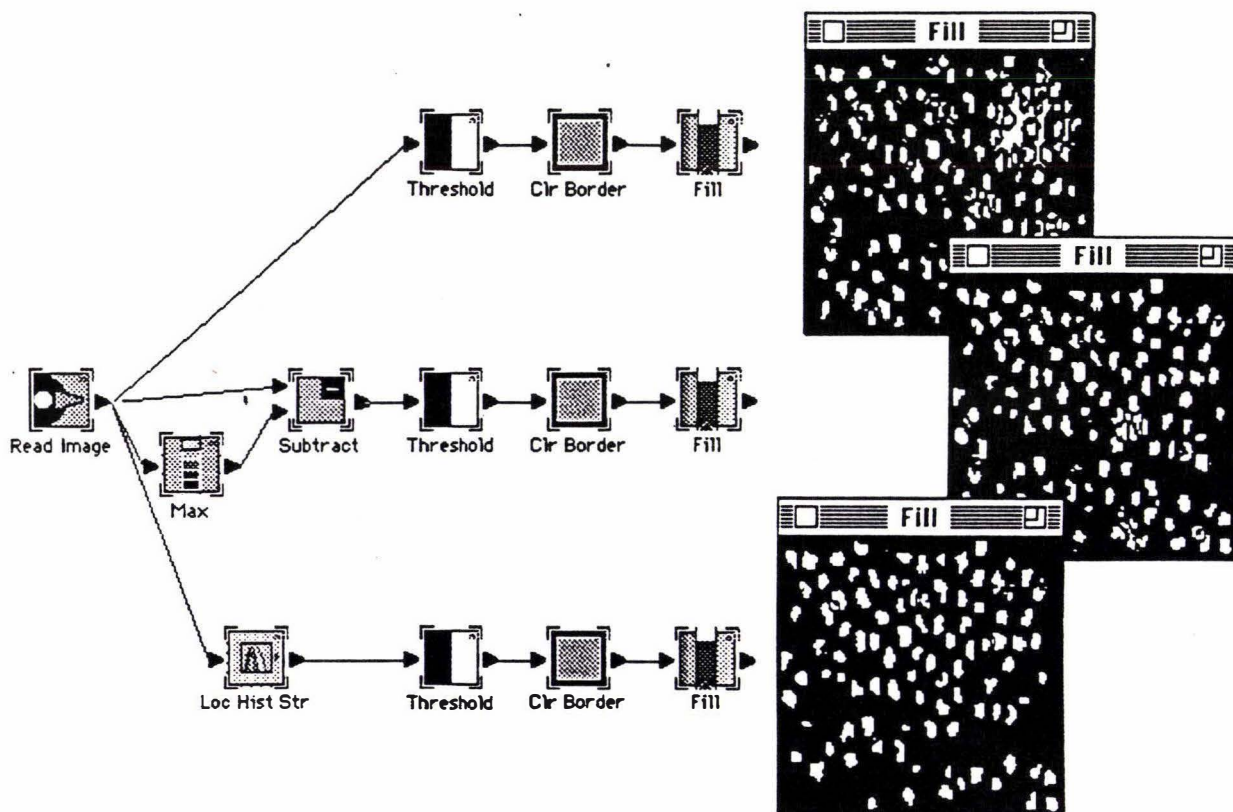


Figure 6.21: A possible scenario for the comparison of three different methods to segment horse axons from a background.

6.6 Summary

This chapter has demonstrated how the OpShop visual language simplifies the development of imaging algorithms. This has been achieved by describing the application of the OpShop language in four situations that commonly occur in the process of algorithm development.

In summary, the strengths of OpShop language highlighted by the four examples are:

- (i) graphical data flow representation simplifies the comprehensibility of algorithms involving multi-threaded data paths;
- (ii) simple and rapid adjustment of operation parameter values simplifies the search for suitable calculation parameter values; this is the facility of parameter exploration;
- (iii) easy addition, insertion, removal and reordering of operations simplifies the search for suitable algorithm topologies; this is the facility of topology exploration;
- (iv) graphical data flow representation allows parallel experiments to be represented explicitly and so simplifies the tasks of choosing between alternative algorithms.

Chapter 7

Summary and Conclusions

In this thesis, the research which lead to the development of a graphical human-computer interface for imaging applications was described. This involved an analysis of the task domain, an evaluation of interface techniques in the context of algorithm development, and the adoption of a user-oriented design approach. The implementation of the design resulted in a visual language software package, which was outlined in Chapters 5 and 6.

Chapter 2 examined in detail the application domain relevant to this thesis: that is, image processing algorithm development. A new perception arose from the discussion of algorithms: that algorithms can be modelled by either a *process-oriented* or a *data-oriented* computational structure. The process-oriented model has been adopted widely in the past, because a sequential algorithm maps directly to the representation required for execution on a typical von Neumann machine. This model is expedient for execution, but not necessarily for development. The example given in Section 2.3 demonstrated that a data-oriented model mapped closely to the conceptual model which a human designer forms to explain the solution of an imaging problem. The data-oriented model was formalised in a graphical representation called the *solution graph*. This

representation was used to illustrate a suite of heuristics commonly used by algorithm designers in the construction of a typical image processing algorithm.

In *Chapter 3*, the applicability of conventional interface techniques for the task of heuristic algorithm development was evaluated. The aim of this evaluation was to provide insight into how algorithm development is assisted or hindered by currently used types of interfaces. This insight was taken into account in the proposal of a new interface design. The evaluation revealed that most conventional techniques hinder algorithm development because they lack either direct forms of interaction or language facilities to describe the algorithm.

- *Command line interfaces* were found to provide indirect mediation between the user and the task because of the requirement for symbolic references to data objects and the need for repetitious typing to repeat command sequences.
- *Menu interfaces* require fewer user actions to select and execute operations than command languages. However, the pointing model for command selection leads to records of command sequences that are difficult to read and interpret.
- A high degree of user interaction is achieved by *direct manipulation interfaces* through the property of direct engagement. This interface style naturally suits imaging applications because both tool and data are inherently spatial in nature. The main drawback of this interface is that macros are represented as a sequences of user actions that are related to the interface, rather than related directly to the imaging task.
- *Visual languages* were found to represent algorithms in an understandable form without sacrificing direct interaction. For this reason, the visual language was favoured as the interaction technique for the proposed design.

Chapter 4 outlined the approach taken for the design of an interface tailored specifically for the interactive development of imaging algorithms. The main insight afforded in this chapter is that a systematic method of interface design involves: (i) a coherent design model, (ii) a user-oriented analysis of the task, and (iii) a thorough understanding of the available interface techniques and the success with which they have been used in the past. The information for the second and third aspects were furnished by Chapters 2 and 3 respectively.

The software was designed according to a *user-centred* model (Norman, 1990). This model was used to explain an interface design in terms of: the user's model, the designer's model, and the system's visual appearance. This approach suggests that in order for a design to be user-centred, these three aspects should be equivalent, and that the specification of the designer's model and the system's visual appearance should be defined by the user's model. Since a user's model explains the essential interaction between the user and the task, it was therefore appropriate to adopt the solution graph as the representation for the user's model. *Stepwise refinement* and *dynamic exploration* were cited as two complementary modes used in building a solution graph. These modes can be

considered as heuristics because they define problem solving strategies that increase the likelihood of attaining a successful solution. Analysis of the strategies revealed that both could be decomposed into cycles of three stages: generation, execution, and evaluation. The two strategies exhibit strong similarities and, in fact, complement each other in the design process. Therefore, it was suggested that both strategies could be accommodated in the same interface, and that such an integration would be desirable. To achieve this integration, the distinctions between the two modes were clarified. Stepwise refinement was observed to be a goal-directed approach, where its three stages were carried out in cycles of discrete steps. On the other hand, dynamic exploration was seen as an exploratory approach, where the distinctions between its three stages were blurred and instead appeared as a single continuous action.

Chapter 5 described the features and discussed the development of a software package. This package, OpShop, was implemented in accordance to the design guidelines set in Chapter 4. The software package was designed to support the stepwise refinement and dynamic exploration approaches to the development of imaging algorithms. The solution graph view of algorithms was incorporated as an executable data-flow diagram. Facilities for the rapid adjustment of operation parameter values and algorithm topology were provided via graphical objects that could be directly manipulated with a mouse pointing device. Rapid calculation of results were provided by an execution scheduler that responded to changes of operation parameters or algorithm topology made by the user. Such an execution scheme maintained strict consistency between the results and the algorithm definition. The rapid delivery of results after changes of parameter or topology was found to strengthen the relationship between the algorithm controls and the displayed result.

Chapter 6 described four situations where the visual and dynamic features of the OpShop package worked to simplify algorithm development. Each of these situations were representative of a typical interaction task in algorithm development. The first example demonstrated that multi-threaded data paths were easier to observe when represented in graphical data flow form than in linear textual form. In the second example, the provision of direct manipulation controls to adjust parameter values, coupled with the rapid update of results, was shown to facilitate experimentation with parameter values. The Abingdon Cross benchmark was deliberately chosen for its interdependent parameter settings. This interdependency places a heavy demand on facilities for parameter experimentation. It was shown that the equivalent experimentation on a command line interface would have demanded considerable retyping to the extent of making the process tedious or at worst, infeasible. In the third example, support for experimentation of algorithm topology is illustrated. The ability to directly add, remove, and rearrange operations in an algorithm was demonstrated and contrasted with equivalent procedures in a command line language. The new interface was shown to be a positive advance because it greatly reduced the user action required to perform an investigation. The fourth example demonstrated the software's capability to conveniently provide a

comparison of alternative solutions. Comparisons were facilitated by the side-by-side arrangement of candidate algorithms. This had the effect of highlighting apparent similarities and differences in alternative algorithms.

Each of the examples in this chapter were designed to demonstrate a single interactive feature; however, imaging tasks would typically demand the combined use of these features. The combined benefit of the interaction features provided in the OpShop package are best experienced.

In summary, this thesis has shown:

- that at the development stage, image processing algorithms are more appropriately represented in a data flow form than in a process-oriented form;
- that an analysis of interaction between the user and the application can provide crucial background information on which to base selection of interface techniques;
- that the user-oriented design philosophy expounded by Norman provides a systematic method for the development of an interface for a given application domain;
- that step-wise refinement and dynamic exploration are two principle heuristics used in algorithm development;
- that facilities of parameter and topology exploration can be ably implemented in a direct manipulation interface.

7.1 Suggestions for future work

The primary concern of this thesis has been the human-computer interaction aspects of algorithm development using image processing systems. The insight gained in this thesis regarding interaction could be extended into other application domains.

The ideas of algorithm development presented here have placed an accent on heuristics. It has been assumed that the sole source of intelligence that furnishes heuristic strategies is a human designer. This source of heuristic strategy is weak where the designer is a novice; novices would profit from being assisted by an expert system that acts as a complementary source of heuristic strategy. It is envisaged that the human designer would be in control of the construction of the algorithm, but that the expert system would provide valuable and timely on-line assistance. In such a system, the OpShop environment could act as a visual communication channel - much in the same ways as a blackboard does - through which the human designer and the expert system communicate. The expert system would monitor the algorithm development activity of the user. When the system recognises a situation that could be performed in a more effective way, it could offer its suggestion directly onto the OpShop whiteboard in the form of an

algorithm fragment. The expert system could have a knowledge base that not only covers aspects of heuristics, but it could also suggest possible operations to include in an algorithm (similar to the system proposed by Bailey (1988)) or it could directly adjust parameter values (like the system reported by Sakaue & Tamura, 1985).

The multi-threaded data oriented view of a processing structure could be used in many applications as a visualisation tool; an obvious example is the field of neural networks. In particular, parameter and topology exploration are relevant to the definition and tuning of experimental neural networks. Parallel processing is another field that stands to gain from the ability to represent processing structures as multi-threaded data flows. A fundamental concern of any parallel processing implementation is the effective partition of program execution between processors. This partitioning is inherent in multi-threaded data flows, where each parallel flow can be computed independently of other flows, and hence each flow can be assigned to a separate hardware processor. The application of data flow diagrams as a vehicle for studying algorithms and architectures for parallel image processing is currently being investigated by Tanimoto (1990); OpShop would provide an ideal presentation medium for such a system.

The OpShop language is likely to be extended in future. One major enhancement is to provide control panels for subflows. At present, to access the parameter settings of an operation within a subflow, a subflow window must first be opened to expose the operation to which the parameter belongs. It is envisaged that future versions of OpShop would incorporate subflow control panels to provide direct access to those parameters critical to the functioning of the subflow. Many parameters within a subflow need only be set once - at the time of development - and require no further adjustment: these parameters would not need to be included in the subflow control panel. Parameters that call for constant attention would be included in the higher level control panel. The user action to designate which parameters of the constituent operations to include in the subflow panel is expected to be a drag operation. Such an action is consistent with the direct manipulation theme of the interface.

A second enhancement likely to be made to the OpShop language is the addition of iteration and conditional control constructs. The OpShop language must offer these features if it to serve as a complete specification language for algorithms. So as to not confuse the data flow view of an algorithm with control information, it would be desirable to embed control information within specialised operations rather than introduce line elements. A possible implementation for an iteration construct is shown in Figure 7.1. The iteration is performed by the combination of three distinct parts: a loop operation, a subflow that contains operations to iterate, and a subflow that contains operations to test for a termination condition. The iterated subflow is invoked by the loop operation. Invocation occurs repeatedly provided the intermediate results of the iterated subflow do not exhibit a termination condition. If a termination condition does exist, it is

detected by the test subflow which in turn informs the loop operation to stop looping and to output the final data.

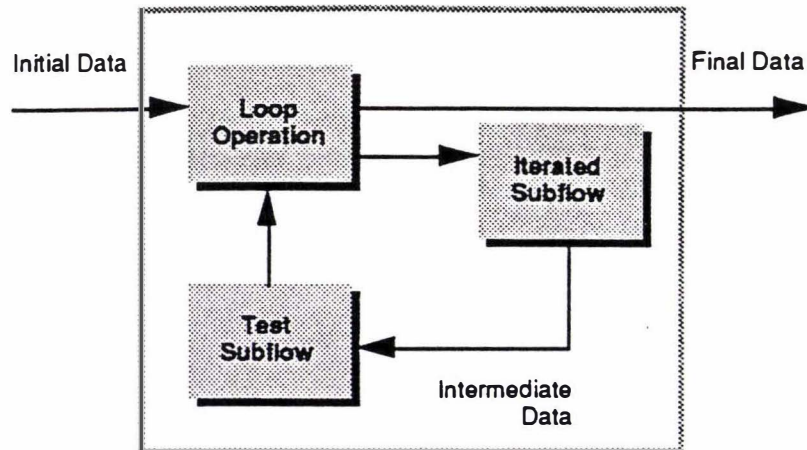


Figure 7.1: Possible implementation for a loop construct in OpShop.

A third addition to the OpShop language that could further enhance its utility is to treat parameter values as data inputs. In the present version of OpShop, parameters can be adjusted only via pre-defined parameter panels (as discussed in Section 5.2.2). However, there are times where the specification of parameter values would be more appropriately performed by the algorithm itself. One situation where this feature might be used is in the specification of optimal threshold values for a threshold operation. For example, Sakaue and Tamura (1985) describe a system where the optimal values are found by a histogram analysis technique. The OpShop language could be modified to allow the option of accepting either values generated by another part of the algorithm or values specified by the user. One possible modification to the language that would achieve this objective is the inclusion of terminal symbols for the input of parameter values, in a similar manner to that currently used for data input. If a parameter is specified by another part of the algorithm, then the corresponding input terminal would be connected to the output of the relevant operation. If instead, parameters are to be supplied by the user, the data terminals could have parameter panels which are, by default, active when no external connection exists.

Execution speed can never be too great for dynamic exploration. Faster computing hardware will naturally benefit the interactivity offered by the OpShop system, especially for the continuous execution of complex algorithms. A possible research avenue to accelerate the processing rate is the development of *incremental* image operations. Small increments in a parameter value may cause some, but not all, pixel values in an image to change. When this occurs, a subsequent operation only needs to recalculate those pixels that depend on the change in its input image. This technique of incremental recalculation is used effectively in electronic circuit design packages and may offer similar speed improvements to the processing of images.

The objective of this study was to create a system that simplifies the interactive design of image processing algorithms. To this end, this thesis has examined an often neglected aspect of the algorithm development task: human-computer interaction. The author and his supervisors hope that the insights reported in this thesis will promote the growth of a new generation of interactive image processing systems that are simple to use, and that facilitate speedy and creative development of image processing algorithms.

References

- Ackerman, W.B. (1982): Data flow languages, *IEEE Computer*, 15(2), 1982, 15-25. [56, 87]
- Adobe (1991): Adobe Photoshop™ user guide, Adobe Systems Incorporated, Mountain View, California, 1991. [32, 35]
- Aloimonos, J. (1988): Shape from texture, *Biological Cybernetics*, 58, 1988, 345-360. [11]
- Aloimonos, J. & Swain, M. (1988): Shape from patterns: Regularization, *International Journal of Computer Vision*, 2, 1988, 171-187. [11]
- Aloimonos, J. & Weiss, I. (1988): Active vision, *International Journal of Computer Vision*, 2, 1988, 333-356. [11]
- Aloimonos, Y. & Rosenfeld, A. (1991): A response to "Ignorance, myopia, and naivete in computer visions systems" by R.C. Jain and T.O. Binford, *CVGIP: Image Understanding*, 53(1), 1991, 120-124. [8]
- Ambler, A.L. & Burnett, M.M. (1989): Influence of visual technology on the evolution of language environments, *IEEE Computer*, 22(10), 1989, 9-22. [41, 61]
- Apperley, M.D. (1990): A private communication. [51]
- Apperley, M.D. & Spence, R. (1989): Lean Cuisine: a low-fat notation for menus, *Interacting with Computers*, 1(1), 1989, 45-68. [30]
- Apple Computer (1985): Inside Macintosh™, Chapter 2, Vol. 1, Addison-Wesley, Reading, Massachusetts, 1985. 64]
- Apple Computer (1987): Hypercard user's guide, Apple Computer Inc., Cupertino, California, 1987. [78]
- Arcelli, C. & Sanniti di Baja, G. (1986): Endoskeleton and exoskeleton of digital figures: an effective procedure, In Cappellini, V & Marconi, R. (Eds.): *Advances in image processing and pattern recognition*, Elsevier Science Publishers B.V. (North-Holland), 1986, 224-228. [13]
- Arcelli, C., Cordella, L. & Levialdi, S. (1975): Parallel thinning of binary pictures, *Electronics Letters*, 11(7), 1975, 148-149. [13]
- Bailey, D.G. & Hodgson, R.M. (1988): VIPS - A Digital image processing algorithm development environment, *Image and Vision Computing*, 6(3), 1988, 176-184. [15, 27]



The numbers in the square brackets indicate the page number, or numbers, at which the reference is cited.

- Bailey, D.G. (1988): Research on computer-assisted generation of image processing algorithms, *IAPR Workshop on Computer Vision - Special Hardware and Industrial Applications*, Oct 12-14, 1988, Tokyo, 294-297. [22, 114]
- Ballard, D.H. & Brown, C.M. (1982): *Computer vision*, Prentice-Hall, Englewood Cliffs, New Jersey, 1982. [8, 12]
- Barraga, N.C. (1986): Sensory perceptual development, in Scholl, G.T. (Ed.): *Foundations of education for blind and visually handicapped children and youth: theory and practice*, American Foundation for the Blind, New York, 1986, 84-98. [1]
- Batchelor, B.G., Hill, D.A. & Hodgson, D.C. (Eds.) (1985): *Automated visual inspection*, Elsevier Science Publishers B.V. (North-Holland), 1985. [8]
- Bates, R.H.T.B. & McDonnell, M.J. (1986): *Image restoration and reconstruction*, Clarendon Press, Oxford, England, 1986. [11]
- Bates, R.H.T.B. & Peters, T.M. (1971): Towards improvements in tomography, *New Zealand Journal of Science*, 14(4), 1971, 883-896. [11]
- Bjorklund, C., Noga, M., Barrett, E. & Kuan, D. (1989): Lockheed imaging technology research for missiles, *Proceedings of the Image Understanding Workshop*, DARPA/ISTO, Palo Alto, California, May 23-26, 1989, 219-231. [8]
- Booch, G. (1986): Object-oriented development, *IEEE Transactions on Software Engineering*, SE-12(12), 1986, 211-221. [78]
- Bowyer, K.W. & Jones, J.P. (1991): Revolutions and experimental computer vision, *CVGIP: Image Understanding*, 53(1), 1991, 127-128. [8]
- Bracewell, R.N. (1986): *The Fourier transform and its applications*, 2nd Edition, McGraw-Hill, New York, 1986. [11]
- Brumfitt, P.J. (1984): Environments for image processing algorithm development, *Image and Vision Computing*, 2(4), 1984, 198-203. [3]
- Budinger, T.F. (1980): Physical attributes of single-photon tomography, *Journal of Nuclear Medicine*, 21(6), 1980, 579-592. [10]
- Budinger, T.F., Gullberg, G.T. & Huesman, R.H. (1979): Emission computed tomography, Chapter 5: Image reconstruction from projections, implementations and applications, Herman, G.T. (Ed.). Springer-Verlag, Berlin, 1979, 147-246. [10]
- Card, S.K., Moran, T.P. & Newell, A. (1983): *The psychology of human-computer interaction*, Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1989. [88]

- Castleman, K.R. (1979): Digital image processing, Prentice-Hall, Englewood Cliffs, New Jersey, 1979. [7, 11, 12, 14]
- Catanzariti, E., di Cerbo, M.G. & Menna, R. (1989): A method for representing and computing immediate texture discrimination in natural images, in Cantoni, V, Cordella, L.P., Levialdi, S. & Sanniti di Baja, G. (Eds.): *Progress in image analysis and processing*, World Scientific Publishing Co., Singapore, 1990, 36-43. [13]
- Corn, A. (1983): Vision function: a model for individuals with low vision, *Journal of Visual Impairment of Blindness*, 77, 1983, 373-377. [2]
- Cox, B. (1986): Object-oriented programming: an evolutionary approach, Addison-Wesley, Reading, Massachusetts, 1986. [78]
- Davis, R. (1990): Iconic interface processing in a scientific environment, *SunExpert*, June 1990, 80-86. [42]
- De Bono, E. (1969): The mechanism of mind, Harmondsworth, Penguin, 1969. [18]
- De Marco, T. (1978): Structured analysis and system specification, Prentice-Hall, Englewood Cliffs, New Jersey, 1978. [50]
- De Menthon, D., Siddalingaiah, T. & Davis, L.S. (1987): Production of Dense Range Images with the CVL Light-Stripe Range Scanner, *Center for Automation Research Report*, University of Maryland, College Park, Maryland 20742, December, 1987. [11]
- Docker, T.W.G. (1989): SAME Structured analysis modelling environment: a prototyping tool, Ph.D. Thesis, Department of Computer Science, Massey University, Palmerston North, New Zealand. [84]
- Dorner, D. (1983): Heuristics and cognition in complex systems, in Groner, R., Groner, M. & Bischof, W.F. (Eds.): *Methods of heuristics*, Lawrence Erlbaum Associates, New Jersey, 1983. [18, 19, 22]
- Fennema, C., Hanson, A.R. & Riseman, E. (1989): Towards autonomous mobile robot navigation, *Proceedings of the Image Understanding Workshop*, DARPA/ISTO, Palo Alto, California, May 23-26, 1989, 219-231. [8]
- Fischler, M.A. & Firschein, O. (Eds.) (1987): Readings in computer vision: issues, problems, principles, and paradigms, Morgan Kaufmann Publishers, California, 1987.[8]
- Foley, J.D. & Van Dam, A. (1982): Fundamentals of interactive computer graphics, Addison-Wesley, Reading, Massachusetts, 1982. [13]
- Foley, J.D., van Dam, A., Feiner, S.K. & Hughes, J.F. (1990): Computer graphics: principles and practice, 2nd Edition, Addison-Wesley, Reading, Massachusetts, 1990. [37]

- Freeman, H. (1961): On the encoding of arbitrary geometric configurations, *IRE Transactions on Electronic Computers*, EC-10(2), 1961, 260-268. [13]
- Freeman, H. (1986): Image processing and pattern recognition: a general overview, In Cappellini, V & Marconi, R. (Eds.): *Advances in Image Processing and Pattern Recognition*, Elsevier Science Publishers B.V. (North-Holland), 1986, 224-228. [7, 10]
- Freeman, H. (1989): Development of a trainable machine-vision inspection system, in Cantoni, V, Cordella, L.P., Levialdi, S. & Sanniti di Baja, G. (Eds.): *Progress in image analysis and processing*, World Scientific Publishing Co., Singapore, 1990, 375-388. [8]
- Fu, K.S. & Mui, J.K. (1981): A survey on image segmentation, *Pattern Recognition*, 13, 1981, 3-16. [12]
- Gane, C. & Sarson, T. (1979): *Structure systems analysis: tools and techniques*, Prentice-Hall, New Jersey, 1979. [50]
- Garden, K.L. (1984): *An Overview of Computed Tomography*, Ph.D. Thesis, Department of Electrical and Electronic Engineering, University of Canterbury, 1984. [11]
- Gaver, W.W. (1991): Technology affordances, *CHI'91 Conference Proceedings*, ACM Press, New York, 1991. [70]
- Glinert, E.P. & Tanimoto, S.L. (1984): Pict: An interactive graphical programming environment, *IEEE Computer*, November, 1984, 7-25. [41]
- Glinert, E.P., Kopache, M.E. & McIntyre, D.W. (1990): Exploring the general-purpose visual alternative, *Journal of Visual Languages and Computing*, 1(1), 1990, 3-39. [41]
- Gonzalez, R.C. & Wintz, P. (1987): *Digital Image Processing*, 2nd Edition, Addison-Wesley, Reading, Massachusetts, 1987. [7, 11, 12, 13]
- Goodman, D. (1990): *The complete HyperCard 2.0 handbook*, 3rd Ed., Bantam Books, New York, 1990. [78]
- Goodman, P. & Spence, R. (1978): The effect of system response time on interactive computer aided problem solving, *Computer Graphics*, 12, 1978, 100-104. [95]
- Hall, E.H. (1974): Almost uniform distributions for computer image enhancement, *IEEE Transactions on Computers*, C-23(2), 1974, 207-208. [11]
- Haralick, R.M. & Shapiro, L.G. (1991): Glossary of computer vision terms, *Artificial Intelligence*, 24(1), 1991, 69-93. [7]
- Haralick, R.M. (1986): Computer vision theory: the lack thereof, *Computer Graphics, Vision, and Image Processing*, 36, 1986, 372-386. [3]

- Hawke, D. (1989): Terrain modelling and GIS. In Mackaness, W. (Ed.): *Proceedings of the Inaugural Colloquium of the Spatial Information Research Centre*, University of Otago, New Zealand, 30 Nov - 1 Dec, 1989, 90-95. [11]
- Herman, G.T. (1980): Image reconstruction from projections: fundamentals of computerized tomography, Academic Press, New York, 1980. [11]
- Herman, G.T. (Ed.) (1979): Image reconstruction from projections: implementation and applications, Springer-Verlag, Berlin, 1979. [11]
- Hildreth, E. (1980): A computer implementation of a theory of edge detection, *MIT AI Laboratory Technical Report*, TR-579, 1980. [12]
- Hirakawa, M., Tanaka, M. & Ichikawa, T. (1990): An iconic programming system, HI-VISUAL, *IEEE Transactions on Software Engineering*, SE-16(10), 1990, 1178-1184. [40]
- Hollingham, J. (1984): Machine vision: the eyes of automation, Springer-Verlag, Berlin, Germany, 1984. [8]
- Hounsfield, G.N. (1980): Computer medical imaging, Nobel Lecture, *Journal of computer-assisted tomography*, 4, 1980, 665ff. [10]
- Huang, T.S. (1991): Computer vision needs more experiments and applications, *CVGIP: Image Understanding*, 53(1), 1991, 125-126. [8]
- Hummel, R. (1977): Image enhancement by histogram transformation, *Computer Graphics and Image Processing*, 6, 1977, 184-195. [11]
- Ichikawa, T. & Hirakawa, M. (1990): Iconic programming: where to go?, *IEEE Software*, November 1990, 63-68. [40]
- Imagelab™ (1987): User manual, Werner Frei Associates, Santa Monica, California.[40]
- Imaging Technology (1987): ImageActionplus user's guide, Imaging Technology Incorporated, Massachusetts, 1987. [33]
- Imaging Technology (1989a): ITEX 200 image processing functions, Imaging Technology Incorporated, Massachusetts, 1989. [32, 33]
- Imaging Technology (1989b): ImageAction 200 user's guide, Imaging Technology Incorporated, Massachusetts, 1989. [27]
- Jain, R.C. & Binford, T.O. (1991): Ignorance, myopia, and naivete in computer visions systems, *CVGIP: Image Understanding*, 53(1), 1991, 112-117. [8]
- Jain, R.C. & Jain, A.K. (Eds.) (1990): Analysis and interpretation of range images, Springer-Verlag, New York, 1990. [11]

- Kacmar, C.J. & Carey, J.M. (1991): Assessing the usability of icons in user interfaces, *Behaviour and Information Technology*, 10(6), 1991, 443-457. [52]
- Kennegieter, N.J. (1989): A study of the distal hindlimb muscles and nerves in normal and laryngeal hemiplegic horses, Ph.D. Thesis, Department of Veterinary Science, Massey University, Palmerston North, New Zealand, 1989. [102]
- Knoll, G.F. (1983): Single-photon emission computed tomography, *Proceedings of the IEEE*, 71(3), 1983, 320-329. [10]
- Kruger, R.P. & Thompson, W.B. (1981): A technical and economic assessment of computer vision for industrial inspection and robotic assembly, *Proceedings of the IEEE*, 69(12), 1981, 1524-1538. [14]
- LabVIEW (1990): LabVIEW 2 User Manual, National Instruments Corporation, Austin, Texas. [52]
- Lane, R.G. (1988): Blind deconvolution and phase retrieval, Ph.D. Thesis, Electrical and Electronic Engineering Department, University of Canterbury, Christchurch, New Zealand. [27]
- Lauterbur, P.C. (1973): Image formation by induced local interactions: examples employing nuclear magnetic resonance, *Nature*, 242, 1973, 190-191. [11]
- Lawton, D.T. & McConnell, C.C. (1988): Image understanding environments, *Proceedings of the IEEE*, 76(8), 1988, 1036-1050. [8]
- Lee, H. & Wade, G. (Eds.) (1990): *Acoustical Imaging*, 18, Plenum Press, New York, 1990. [11]
- Lenat, D.B. (1983): Towards a theory of heuristics, in Groner, R., Groner, M. & Bischof, W.F. (Eds.): *Methods of heuristics*, Lawrence Erlbaum Associates, New Jersey, 1983. [22]
- Lesczczynski, K.W. & Shalev, S. (1989): A robust algorithm for contrast enhancement by local histogram modification, *Image and Vision Computing*, 7(3), 1989, 205-209. [11]
- Lewitt, R.M. (1983): Reconstruction algorithms: transform methods, *Proceedings of the IEEE*, 71(3), 1983, 390-408. [11]
- Manohar, M., Rao, P.S. & Iyengar, S.S. (1990): Template quadrees for representing region and line data present in binary images, *Computer Vision, Graphics, and Image Processing*, 51, 1990, 338-354. [13]
- Marr, D. (1982): *Vision: a computational investigation into the human representation and processing of visual information*, W.H. Freeman and Company, San Francisco, 1982. [3, 11]

- Marshall, S. (1989): Review of shape coding techniques, *Image and Vision Computing*, 7(4), 1989, 281-294. [15]
- McKeown, D.M., Harvey, W.A. & McDermott, J. (1985): Rule-based interpretation of aerial imagery, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-7(5), 1985, 570-585. [8]
- Milne, A.A. (1928): *The house at Pooh Corner*, Methuen Co. Ltd., London, 1928. [1]
- Monden, N., Yoshimoto, I., Hirakawa, M., Tanaka, M. & Ichikawa, T. (1984): HI-VISUAL: A language supporting visual interaction in programming, *Proceedings of the 1984 IEEE Workshop on Visual Languages*, 1984, 199-205. [40]
- Mucciardi, A.N. & Gose, E.E. (1971): A comparison of seven techniques for choosing subsets of pattern recognition properties, *IEEE Transactions on Computers*, C-20(9), 1971, 1023-1031. [15]
- Myers, B.A. (1990): Taxonomies of visual programming and program visualisation, *Journal of Visual Languages and Computing*, 1(1), 1990, 97-123. [40, 61]
- Naylor, M.J. (1987): The use of structured lighting in 3D image capture, *Proceedings of the 2nd New Zealand image processing workshop*, University of Canterbury, Christchurch, 20-21 August 1987. [11]
- Newell, A. (1983): The heuristics of George Polya, in Groner, R., Groner, M. & Bischof, W.F. (Eds.): *Methods of heuristics*, Lawrence Erlbaum Associates, New Jersey, 1983. [22]
- Newell, A., Shaw, J.C. & Simon, H.A. (1958): Elements of a theory of human problem solving, *Psychological Review*, 65, 151-166. [19]
- Newell, A. & Simon, H. A. (1972): *Human problem solving*, Prentice-Hall, Englewood Cliffs, New Jersey, 1972. 19, 47]
- Ngan, P.M., Apperley, M.D. & Hodgson, R.M. (1990): The user-oriented development of an interface for image processing, *Proceedings of the 5th New Zealand Image Processing Workshop*, Massey University, Palmerston North, New Zealand, 9-10 August, 1990, 63-68. [67]
- Nievergelt, J. & Weydert, J. (1980): Sites, modes, and trails: telling the user of an interactive system where he is, what he can do, and how to get places. In Guedj, R.A., ten Hagen, P.J.W., Hopgood, F.R.A., Tucker, H.A. & Duce, D.A. (Eds.): *Methodology of Interaction*, North Holland, 327-338. [29]
- Norman, D.A. & Chin, J.P. (1990): The menu metaphor: food for thought, *Behaviour and Information Technology*, 8(2), 125-134, 1989. [37]
- Norman, D.A. (1981): The trouble with UNIX, *Datamation*, 27(12), November 1981, 139-150. [30]

- Norman, D.A. (1990): The design of everyday things, Doubleday Currency, New York, 1990. [37, 38, 46, 47, 111]
- Pavlidis, T. (1978): A review of algorithms for shape analysis, *Computer Graphics and Image Processing*, 7, 1978, 243-258. [15]
- Pavlidis, T. (1986): A critical survey of image analysis methods, *Proceedings of the IAPR Eighth International Conference on Pattern Recognition*, Paris, France, Oct 27-31, 1986, 502-511. [12]
- Perry, A. (1989): Segmentation of texture regions using points of sharp intensity change, in Cantoni, V, Cordella, L.P., Levialdi, S. & Sanniti di Baja, G. (Eds.): Progress in image analysis and processing, World Scientific Publishing Co., Singapore, 1990, 99-105. [13]
- Petkovic, D. & Wilder, J. (1991): Machine vision in the 1990s: applications and how to get there, *Machine Vision and Applications*, 4, 1991, 113-126. [3]
- Pietrzykowski, T. & Matwin, S. (1984): PROGRAPH: A preliminary report, University of Ottawa Technical Report, TR-84-07, April, 1984. [32, 78]
- PixelPaint Pro™ (1989): User's manual, SuperMac Technology, Sunnyvale, California, 1989. [32]
- Polya, G. (1957): How to solve it; a new aspect of mathematical method, Doubleday, New York, 1957. [19, 47]
- Polya, G. (1962): Mathematical discovery: on understanding, learning, and teaching problem solving, Vol. 1, John Wiley and Sons, New York, 1962. [18, 19, 47]
- Preston, K. (1989): The Abingdon Cross benchmark survey, *IEEE Computer*, July 1989, 9-18.
- Price, K. (1986): Anything you can do, I can do better (no you can't), *Computer Vision, Graphics, and Image Processing*, 36, 1986, 387-391. [24]
- Prince, B. & Salters, R.H.W. (1992): ICs going on a 3-V diet, *IEEE Spectrum*, 29(5), 22-25, 1992. [90]
- Rasband, W. (1992): Image user's manual, National Institutes of Health, Bethesda, Maryland. [32, 34]
- Rasure, J., Argiro, D., Sauer, T. & Williams, C. (1990): A Visual Language and Software Development Environment for Image Processing, *International Journal of Imaging Systems and Technology*, 2, 1990, 183-199. [41]
- Rasure, J.R. & Williams, C.S. (1991): An integrated data flow visual language and software development environment, *Journal of Visual Languages and Computing*, 2, (1991), 1-30. [41, 51]

- Rogers, Y. (1989): Icons at the interface: their usefulness, *Interacting with Computers*, 1(1), 1989, 105-117. [51]
- Rosenfeld, A. (1984): Image Analysis: problems, progress and prospects, *Pattern Recognition*, 17(1), 1984, 3-12. [10]
- Rosenfeld, A. (1988): Computer vision: basic principles, *Proceedings of the IEEE*, 76(8), 1988, 863-868. [9, 12]
- Rutovitz, D. (1989): Efficient processing of 2-D images, in Cantoni, V, Cordella, L.P., Levialdi, S. & Sanniti di Baja, G. (Eds.): *Progress in image analysis and processing*, World Scientific Publishing Co., Singapore, 1990, 99-105. [13]
- Sahoo, P.K., Soltani, S., Wong, K.C. & Chen, Y.C. (1988): A survey of thresholding techniques, *Computer Vision, Graphics, and Image Processing*, 41(2), 1988, 233-260. [12]
- Sakaue, K. & Tamura, H. (1985): Automatic generation of image processing program by knowledge-based verification, *IEEE Proceedings on Computer Vision and Pattern Recognition*, San Francisco, California, June 19-23, 1985, 189-192. [114, 115]
- Scott, P.D. (1990): Applied machine vision, in Leibovic, K.N. (Ed.): *Science of vision*, Springer-Verlag, New York, 1990, 439-465. [2]
- Shneiderman, B. (1983): Direct manipulation: a step beyond programming languages, *IEEE Computer*, 16(8), 1983, 57-62. [34]
- Shneiderman, B. (1988): We can design better user interfaces: a review of human-computer interaction styles, *Ergonomics*, 31(5), 1988, 699-710. [29]
- Shu, N.C. (1988): *Visual programming*, Van Nostrand Reinhold Company, New York. [40]
- Simon, H.A. (1981): *The sciences of the artificial*, 2nd Edition, The MIT Press, Massachusetts, 1981. [21, 22]
- Smith, S., Schreirer, H.E. & Brown, S. (1989): Analysis of forage crops using GIS and image analysis techniques. In Mackaness, W. (Ed.): *Proceedings of the Inaugural Colloquium of the Spatial Information Research Centre*, University of Otago, New Zealand, 30 Nov-1 Dec, 1989, 155-169. [11]
- Smith, S.L. & Mosier, J.N. (1986): Guidelines for designing user interface software, MITRE, ESD-TR-86-278, Bedford, Massachusetts, 1986. [30]
- Snyder, M.A. (1991): A commentary on the paper by Jain and Binford, *CVGIP: Image Understanding*, 53(1), 1991, 118-119. [8]
- Sommerville, I. (1989): *Software engineering*, 3rd Ed., Addison-Wesley, Reading, Massachusetts, 1989. [78]

- Spence, R. & Apperley, M. (1977): The interactive-graphic man-computer dialogue in computer-aided circuit design, *IEEE Transactions on Circuits and Systems*, CAS-24(2), 1977, 49-61. [48, 49]
- Sykes, J.B. (Ed.) (1982): The concise Oxford dictionary, 7th Edition, Clarendon Press, Oxford. [37]
- Symantec Corporation (1991): THINK C Object-oriented programming manual, Symantec Corporation, Cupertino, California, 1991. [62]
- Tanimoto, S.L. & Kent, E.W. (1990): Architectures and algorithms for iconic-to-symbolic transformations, *Pattern Recognition*, 23(12), 1990, 1377-1388. [13]
- Tanimoto, S.L. (1990): VIVA: A visual language for image processing, *Technical Report #90-02-04*, Department of Computer Science and Engineering, University of Washington, Seattle, Washington. [42, 114]
- Thorndike, E.L. (1898): Animal intelligence: an experimental study of the associative processes in animals, *The Psychological Review, Monograph Supplements*, 2, No.4., 1898. [18]
- Thorpe, C. & Kanade, T. (1989): Carnegie Mellon Navlab vision, *Proceedings of the Image Understanding Workshop*, DARPA/ISTO, Palo Alto, California, May 23-26, 1989, 219-231. [8]
- Unser, M. & Eden, M. (1988): A Multi-resolution feature reduction technique for image segmentation with multiple components, *Proceedings of Computer Vision and Pattern Recognition*, Ann Arbor, Michigan, June 5-9, 1988, 568-573. [12]
- Unser, M., Pelle, G., Prun, P. & Eden, M. (1988): Computer analysis of m-mode echocardiograms: sequential extraction of myocardial borders, In Brun, P., Chadwick, R. S. & Levy, B. (Eds.): *Cardiovascular dynamics and models*, Paris, INSERM, 1988, 304-310. [21]
- Wegner, P. (1987): Dimensions of object-based language design, *Proceedings of the OOPSLA '87 Conference*, October 4-8, 1987, Orlando, Florida. [79]
- White, A.G. & Johnstone, N.M. (1991): Measurement of fruit surface colour in 'Gala' apple (*Malus pumila* Mill.) and twenty of its sports by image analysis, *New Zealand Journal of Crop and Horticultural Science*, 19, 1991, 221-223. [3]
- Wilson, J.C. (1987): Models of the human visual system applied to pattern recognition, Ph.D. Thesis, Department of Electrical and Electronic Engineering, University of Canterbury, Christchurch, New Zealand, 1987. [27, 53]

Appendix 1

Summary of the OpShop Software

A1.1 Introduction

OpShop is general purpose image analysis software for colour Macintosh computers. The distinctive feature of OpShop is its human-computer interface, which is composed of an iconic visual language, where operations are denoted by colour icons, and algorithms by a dataflow network of icons. Construction of algorithms in OpShop are preformed by first selecting operations from the main menu, then interconnecting the data paths between of the data terminals of the operations. Input data terminals are denoted by triangular icons on the left side of an icon, while output data terminals are denoted by triangular icons on the right side of an icon. Each output terminal for an operation has an associated output buffer in which the result of a calculation is stored. The contents of the buffer is displayed by option-clicking on the output terminal icon. The content of the result buffer persists until it is overwritten by a subsequent calculation. Each operation has an associated parameter control panel that can be opened by option-clicking on the operation icon. Execution of an algorithm is performed in

three ways: (i) by double clicking operation icons, (ii) by adjusting a parameter value, and (iii) by adjusting the algorithm data flow topology.

OpShop supports many standard image processing operations including those for image arithmetic, data input and output, geometric transformation, test image generation, linear and non-linear filters, Fourier transformations, contrast enhancement, segmentation, and measurement. Images may be written to and read from PICT files. Subflows are created by selecting the operations to be grouped in the subflow and then choosing the **Create Subflow** menu item.

A1.2 System requirements

OpShop requires a Macintosh with an 8-bit video card for the display of 256 shades of grey. The program requires 3 MBytes of memory by default, but this can be reduced if only a few operations are used. OpShop requires system software greater than version 6.0.7. System 6 system software must be used in conjunction with Apple's 32-bit Quickdraw INIT (available with the Color Disk distribution).

A1.3 Description of Operations

Key to symbols:

- tbi* denotes 'to be implemented'.
- 2D denotes two dimensional data array.
- ME denotes mutually exclusive parameter list.
- S denotes an ON-OFF switch.
- [a .. b] denotes an integer scalar value between a and b inclusively.

A1.3.1 General Menu



Read Image

Brief description: Loads a PICT image file into OpShop.

Outputs:

(2D): The loaded image.

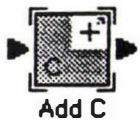


Save Image

Brief description (*tbi*): Saves an OpShop as a PICT file.

Input:

(2D) The image to save.



Add C

Brief description (*tbi*): Adds a constant value to a 2D image.

Input:

- (i) (2D) Input image.

Output:

- (i) (2D) Input image + Offset.

Parameters:

- (i) [0.. 255] Offset: Value to be added to the input image.

- (ii) (ME)

Wrap: Overflow and underflow result values are wrapped around.

Saturate: Overflow and underflow result values are constrained to the range 0 to 255.



Sub C

Brief description (*tbi*): Subtract a constant value from a 2D image.

Input:

- (i) (2D) Input image.

Output:

- (i) (2D) Input image - Offset.

Parameters:

- (i) [0.. 255] Offset: Value to be subtracted from the input image.

- (ii) (ME)

Wrap: Overflow and underflow result values are wrapped around.

Saturate: Overflow and underflow result values are constrained to the range 0 to 255.



Mult C

Brief description (*tbi*): Multiplies a 2D image by a constant.

Input:

- (i) (2D) Input image.

Output:

- (i) (2D) Input image * Factor.

Parameters:

- (i) (Real) Factor: Multiplication factor.

- (ii) (ME)

Wrap: Overflow and underflow result values are wrapped around.

Saturate: Overflow and underflow result values are constrained to the range 0 to 255.



Div C

Brief description (*tbi*): Divides a 2D image by a constant.

Input:

- (i) (2D) Input image.

Output:

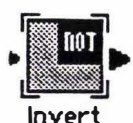
- (i) (2D) Input image / Denominator.

Parameters:

- (i) (Real) Denominator: The value to be divided by.
- (ii) (ME)

Wrap: Overflow and underflow result values are wrapped around.

Saturate: Overflow and underflow result values are constrained to the range 0 to 255.



Invert

Brief description: Inverts the intensity range of a 2D image.

Input:

- (i) (2D) Input image.

Output:

- (i) (2D) Inverted image.



Clear Image

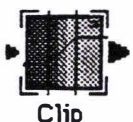
Brief description (*tbi*): Sets all pixels in an image to zero.

Input:

- (i) (2D) Input image.

Output:

- (i) (2D) Cleared image.



Clip

Brief description (*tbi*): Clips the intensity range of a 2D image between two intensities. Any pixel whose value is outside these intensities is truncated to the nearer intensity. Pixel values between the two intensities are not affected.

Input:

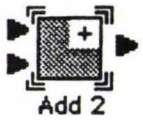
- (i) (2D) Input image.

Output:

- (i) (2D) Clipped image.

Parameters:

- (i) [0 .. 255] Lower intensity
- (ii) [0 .. 255] Upper intensity



Brief description (*tbi*): Adds two images.

Input:

- (i) (2D) First addend.
- (ii) (2D) Second addend.

Output:

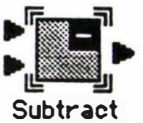
- (i) (2D) The sum of the two addends.

Parameters:

- (i) (ME)

Wrap: Overflow and underflow result values are wrapped around.

Saturate: Overflow and underflow result values are constrained to the range 0 to 255.



Brief description: Calculates the difference of two images.

Input:

- (i) (2D) Reference image.
- (ii) (2D) Image to be subtracted.

Output:

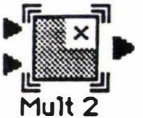
- (i) (2D) The image representing Input (i) - Input (ii).

Parameters:

- (i) (ME)

Wrap: Overflow and underflow result values are wrapped around.

Saturate: Overflow and underflow result values are constrained to the range 0 to 255.



Brief description (*tbi*): Multiplies two images.

Input:

- (i) (2D) First factor.
- (ii) (2D) Second factor.

Output:

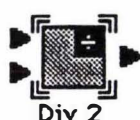
- (i) (2D) The product of the two factors.

Parameters:

- (i) (ME)

Wrap: Overflow and underflow result values are wrapped around.

Saturate: Overflow and underflow result values are constrained to the range 0 to 255.



Brief description (*tbi*): Divides two images.

Input:

- (i) (2D) Numerator image.
- (ii) (2D) Denominator image.

Output:

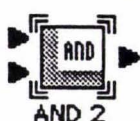
- (i) (2D) The ratio image

Parameters:

- (i) (ME)

Wrap: Overflow and underflow result values are wrapped around.

Saturate: Overflow and underflow result values are constrained to the range 0 to 255.



Brief description (*tbi*): Calculates the bitwise boolean AND of intensities of two images. The truth table for a bitwise boolean AND is:

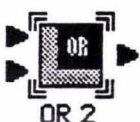
Input 1	Input 2	Output
0	0	0
0	1	0
1	0	0
1	1	1

Input:

- (i) (2D) First image.
- (ii) (2D) Second image.

Output:

- (i) (2D) ANDed image.



Brief description (*tbi*): Calculates the bitwise boolean OR of intensities of two images. The truth table for a bitwise boolean OR is:

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	1

Input:

- (i) (2D) First image.
- (ii) (2D) Second image.

Output:

- (i) (2D) ORed image.



Brief description (*tbi*): Calculates the bitwise boolean Exclusive OR of intensities of two images. The truth table for a bitwise boolean XOR is:

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Input:

- (i) (2D) First image.
- (ii) (2D) Second image.

Output:

- (i) (2D) XORed image.



Brief description (*tbi*): Geometrically reflect the pixel intensities along a vertical line that divides the image into equal left and right halves.

Input:

- (i) (2D) Input image.

Output:

- (i) (2D) Transformed image.



Brief description (*tbi*): Geometrically reflect the pixel intensities along a horizontal line that divides the image into equal top and bottom halves.

Input:

- (i) (2D) Input image.

Output:

- (i) (2D) Transformed image.



Brief description (*tbi*): Geometrically rotate the pixel intensities through an arbitrary angle/

Input:

- (i) (2D) Input image.

Output:

- (i) (2D) Transformed image.

Parameters:

- (i) (Real): Angle (in degrees) to rotate image. Positive values denote an anti-clockwise rotation, while negative values denote a clockwise rotation.



Brief description (*tbi*): Geometrically transpose the pixels intensities lying along the rows and columns of an image.

Input:

- (i) (2D) Input image.

Output:

- (i) (2D) Transposed image.



Brief description (*tbi*): Resize the image.

Input:

- (i) (2D) Input image.

Output:

- (i) (2D) Resized image.

Parameters:

- (i) [0 .. 2048] New x dimension
- (ii) [0 .. 2048] New y dimension
- (iii) (ME) Nearest Neighbour / 1st Order: Order of the interpolation or extrapolation.



Brief description: Generate a Wedge test image.

Output:

- (i) (2D) Wedge image.

Parameters:

- (i) [0 .. 255] Period of the wedge in y direction.



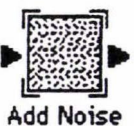
Brief description: Generate a Cross test image.

Output:

- (i) (2D) Cross image.

Parameters:

- (i) [0 .. 255] Cross intensity.
- (ii) [0 .. 255] Background intensity.



Brief description: Superimpose Gaussian distributed intensity noise on an image.

Input:

- (i) (2D) Input image upon which noise is to be superimposed

Output:

- (i) (2D) Image with superimposed noise.

Parameters:

- (i) [0 .. 255] Standard deviation of the noise distribution.



Brief description: Sets the intensities of the border pixels of an image to zero.
The border to a single pixel wide.

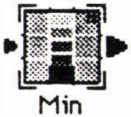
Input:

- (i) (2D) Input image.

Output:

- (i) (2D) Image with cleared border.

A1.3.2 Preprocessing Menu



Brief description(*tbi*): Filters the image selecting as the output the minimum pixel value within a specified size window.

Input:

- (i) (2D) The image to be filtered.

Output:

- (i) (2D) The resultant filtered image.

Parameters:

- (i) [0..255] X dimension of the window
- (ii) [0..255] Y dimension of the window



Brief description (*tbi*): Filters the image selecting as the output the maximum pixel value within a specified size window.

Input:

- (i) (2D) The image to be filtered.

Output:

- (i) (2D) The resultant filtered image.

Parameters:

- (i) [0..255] X dimension of the window
- (ii) [0..255] Y dimension of the window



Brief description (*tbi*): Filters the image selecting as the output the difference between the maximum and minimum pixel value within a specified size window. Such a filter is useful for detecting edges within the input image.

Input:

- (i) (2D) The image to be filtered.

Output:

- (i) (2D) The resultant filtered image.

Parameters:

- (i) [0..255] X dimension of the window
- (ii) [0..255] Y dimension of the window



Brief description (*tbi*): Rank filters an image using a 3 x 3 square window.

Input:

- (i) (2D) The image to be filtered.

Output:

- (i) (2D) The resultant filtered image.

Parameters:

- (i) [0..9] The rank value to be used.



Brief description: Filters an image using a moving average in a window.

Input:

- (i) (2D) The image to be filtered.

Output:

- (i) (2D) The resultant filtered image.

Parameters:

- (i) [0..255] X dimension of the window
- (ii) [0..255] Y dimension of the window
- (iii) (ME) Trim/ Zero.

Trim: As the window nears the edge of the image, the window is reduced in size by taking the average only of the window pixels within the image.

Zero: The window remains the specified size, and the pixels outside the image are assumed to be zero.



Brief description (*tbi*): Filters an image using a Gaussian weighted moving average in a window.

Input:

- (i) (2D) The image to be filtered.

Output:

- (i) (2D) The resultant filtered image.

Parameters:

- (i) [0..255] X dimension of the window
- (ii) [0..255] Y dimension of the window
- (iii) (ME) Trim/ Zero.

Trim: As the window nears the edge of the image, the window is reduced in size by taking the average only of the window pixels within the image.

Zero: The window remains the specified size, and the pixels outside the image are assumed to be zero.



Enhance

Brief description (*tbi*): Filter an image selecting as the output pixel either the minimum or maximum value from within a specified sized window, depending on which is closer to the original pixel value. Such a filter is useful for enhancing edges within the input image.

Input:

- (i) (2D) The image to be filtered.

Output:

- (i) (2D) The resultant filtered image.

Parameters:

- (i) [0..255] X dimension of the window
- (ii) [0..255] Y dimension of the window



Roberts

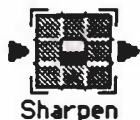
Brief description (*tbi*): Filter an image by taking a Robert's product within a moving 2x2 window in the input image.

Input:

- (i) (2D) The image to be filtered.

Output:

- (i) (2D) The resultant filtered image.



Sharpen

Brief description (*tbi*): Filter an image by convolving the input image with a Laplacian kernel.

Input:

- (i) (2D) The image to be filtered.

Output:

- (i) (2D) The resultant filtered image.

Parameters:

- (i) [0..255] X dimension of the window
- (ii) [0..255] Y dimension of the window



Mode

Brief description (*tbi*): Filters the image selecting as the output the mode pixel value within a specified size window.

Input:

- (i) (2D) The image to be filtered.

Output:

- (i) (2D) The resultant filtered image.

Parameters:

- (i) [0..255] X dimension of the window
- (ii) [0..255] Y dimension of the window



Brief description (*tbi*): Filters an image by calculating the variance within a specified size window.

Input:

- (i) (2D) The image to be filtered.

Output:

- (i) (2D) The resultant filtered image.

Parameters:

- (i) [0..255] X dimension of the window
- (ii) [0..255] Y dimension of the window



Brief description (*tbi*): Filters the image selecting as the output the median pixel value within a specified size window.

Input:

- (i) (2D) The image to be filtered.

Output:

- (i) (2D) The resultant filtered image.

Parameters:

- (i) [0..255] X dimension of the window
- (ii) [0..255] Y dimension of the window



Brief description (*tbi*): Performs the forward Fast Fourier Transform.

Input:

- (i) (2D Complex) The spatial image.

Output:

- (i) (2D Complex) The frequency space image.

Parameters:

- (i) (ME) Top-Left Origin/Centre Origin.
 Top-Left Origin: Origin of the images is at (0, 0)
 Centre Origin: Origin of the images is at the centre of the images.



Brief description (*tbi*): Performs the reverse Fast Fourier Transform.

Input:

- (i) (2D Complex) The frequency space image.

Output:

- (i) (2D Complex) The spatial image.

Parameters:

- (i) (ME) Top-Left Origin/Centre Origin.
 Top-Left Origin: Origin of the images is at (0, 0)
 Centre Origin: Origin of the images is at the centre of the images.



Make Z

Brief description (*tbi*): Create a complex image from real and imaginary components or magnitude and phase components.

Input:

- (i) (2D) Real image or magnitude image.
- (ii) (2D) Imaginary image or phase image.

Output:

- (i) (2D Complex) The resultant complex image.

Parameter:

- (i) (ME) Rectangular/Polar: Form of the input and output components.



Split Z

Brief description (*tbi*): Split components of a complex image into real and imaginary components or magnitude and phase components.

Input:

- (i) (2D Complex) The complex input image.

Output:

- (i) (2D) Real image or magnitude image.
- (ii) (2D) Imaginary image or phase image.

Parameter:

- (i) (ME) Rectangular/Polar: Form of the input and output components.



Hist Stretch

Brief description: Stretches the contrast range of an image linearly so that the given intensity range fills the available intensity range.

Input:

- (i) (2D) The image to be stretched

Output:

- (i) (2D) The stretched image

Parameters:

- (i) [0.255] The intensity level that is set to 0.
- (ii) [0.255] The intensity level that is set to 255.



Loc Hist Str

Brief description: Performs a local contrast enhancement by stretching the intensities linearly within a window of the specified size. For each window position, only the pixel in the centre of the window is stretched.

Input:

- (i) (2D) The image to be filtered.

Output:

- (i) (2D) The resultant filtered image.

Parameters:

- (i) [0..255] X dimension of the window
- (ii) [0..255] Y dimension of the window



Brief description (*tbi*): Performs histogram equalisation on an image.

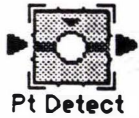
Input:

- (i) (2D) The image whose contrast is to be enhanced.

Output:

- (i) (2D) The contrast enhanced image.

A1.3.3 Segmentation Menu



Brief description (*tbi*): Performs 3x3 linear filter to detect isolated pixels.

Convolution kernel:

-1	-1	-1
-1	8	-1
-1	-1	-1

Input:

- (i) (2D) The image to be filtered.

Output:

- (i) (2D) The filtered image.



Brief description (*tbi*): Performs 3x3 linear filter to detect isolated pixels.

e.g. Convolution kernel for horizontal lines:

-1	-1	-1
2	2	2
-1	-1	-1

Input:

- (i) (2D) The image to be filtered.

Output:

- (i) (2D) The filtered image.

Parameters:

- (i) (ME) Horizontal/Vertical/BR-TL/TR-BR: Direction of detected lines.



Sobel

Brief description (*tbi*): Filters an image with a SOBEL filter. This is used for detecting edges.

Input:

- (i) (2D) The image to be filtered.

Output:

- (i) (2D) The resultant filtered image.

Parameters:

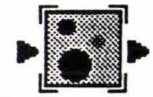
- (i) (3x3 MASK) The weights within the 3x3 window. These are weights are for a linear convolution filter. The input image is filtered twice, once with the weights and again with the weights transposed.

- (ii) (ME) RMS/SUM/MAX:

RMS: The square root of the sum of the squares of the two filtered images is returned.

SUM: Half of the sum of the absolute values of the two filtered images is returned.

MAX: The maximum of the absolute values of the two filtered images is returned.



Label Region

Brief description: Finds and labels the intensity peaks in an image .

Input:

- (i) (2D) The image with the peaks.

Output:

- (i) (2D) The labelled image.

Parameters:

- (i) [0..255] An allowable tolerance between the peaks.



Fill

Brief description: Fills a region in an image within a specified boundary.

Input:

- (i) (2D) The image to be filled. A region from the seed is grown until the specified boundary is reached.

Output:

- (i) (2D) The filled image.

Parameters:

- (i) [0..255] Fill intensity.
- (ii) [0..255] X position of seed pixel.
- (iii) [0..255] Y position of seed pixel.
- (iv) (S) Diagonal: True assumes 8-connected boundary; False assumes 4-connected boundary.
- (v) (ME) Boundary == Fill: Fill boundary intensity is equal to fill intensity
Boundary <= Fill: Fill boundary intensity is less than or equal to fill intensity.
Boundary >= Fill: Fill boundary intensity is greater than or equal to fill intensity.



Brief description: Thresholds an image so that pixel values between two threshold intensities are set to 255, while those outside are set to 0.

Input:

- (i) (2D) The image to be thresholded.

Output:

- (i) (2D) The resultant thresholded image.

Parameters:

- (i) [0.255] The lower threshold intensity
- (ii) [0.255] The upper threshold intensity



Brief description: Creates an intensity histogram of a 2D grey-scale image.

Input:

- (i) (2D) The image from which the histogram is to be calculated.

Output:

- (i) (1D) The resultant histogram.



Brief description (*tbi*): Transforms the intensities of a grey-scale image by a look-up table.

Input:

- (i) (2D) The image to be transformed.
- (ii) (1D) The look-up table that specifies the transformation.

Output:

- (i) (2D) The transformed image.



Brief description (*tbi*): Assigns a Thresholding function to a LUT.

Input:

- (i) (1D) The input look-up table.

Output:

- (i) (1D) The look-up table containing the thresholding function.

Parameters:

- (i) [0.255] The lower threshold intensity
- (ii) [0.255] The upper threshold intensity



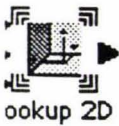
Brief description: Creates 2D intensity histogram of two 2D grey-scale images. The first image forms of the histogram, while the second image forms the vertical axis.

Input:

- (i) (2D) The first image from which the horizontal axis of the histogram is to be calculated.
- (ii) (2D) The second image from which the vertical axis of the histogram is to be calculated.

Output:

- (i) (2D) The resultant 2D histogram.



Brief description: Transforms the intensities of two grey-scale image by a 2D look-up table to create a single 2D image.

Input:

- (i) (2D) The 2D look-up table that specifies the transformation.
- (ii) (2D) The first image to be transformed (indexes the horizontal axis).
- (iii) (2D) The second image to be transformed (indexes the vertical axis).

Output:

- (i) (2D) The transformed image.



Brief description (*tbi*): Dilate white elements in a region.

Input:

- (i) (2D) The binary or grey-scale image to be dilated.

Output:

- (i) (2D) The dilated image.

Parameters:

- (i) (ME) Binary/Grey Scale: type of input image.



Brief description (*tbi*): Erode white elements in a region.

Input:

- (i) (2D) The binary or grey-scale image to be eroded.

Output:

- (i) (2D) The eroded image.

Parameters:

- (i) (ME) Binary/Grey Scale: type of input image.



Inten CHULL

Brief description (*tbi*): Performs the convex hull of intensities along the rows of an image.

Input:

- (i) (2D) The grey-scale image to be hulled.

Output:

- (i) (2D) The convex hull of the input image.



Bin CHULL

Brief description (*tbi*): Performs a convex hull operation on every white blob in a binary image. If two or more blobs merge as a result of this operation, the convex hull of the composite is returned. A convex hull is the minimum area convex polygon that totally contains the object.

Input:

- (i) (2D) The binary image to be hulled.

Output:

- (i) (2D) The convex hull of the input image.



Skeletonize

Brief description: Thins an image down to a single pixel wide skeleton.

Input:

- (i) (2D) The binary image to be thinned. The pixels on the edge of the image are cleared before processing begins.

Output:

- (i) (2D) The skeletonised image.

Parameters:

- (i) [0..255] Prune Length: Branches less than this number are excluded from the skeletonised result.

- (ii) (ME) Binary/Coded:

Binary: The skeleton is returned uncoded.

Coded: The skeleton is coded with the distance from the edge of the blob.

A1.3.4 Measurement Menu



Inten Prof

Brief description (*tbi*): Measures an intensity profile across the image. The line of the profile is specified interactively by the user.

Input:

- (i) (2D) The image to be measured.

Output:

- (i) (1D) The list of intensity values.

Parameters:

- (i) [0..255] X co-ordinate of point 1.
- (ii) [0..255] Y co-ordinate of point 1.
- (iii) [0..255] X co-ordinate of point 2.
- (iv) [0..255] Y co-ordinate of point 2.

Note:

Eventually the line will specified by drawing a line over the image in the display window, in addition to specifying the co-ordinates of the end points.



Stats

Brief description (*tbi*): Returns statistics to describe the intensity distribution of an image. The measured statistics include: mean, standard deviation, maximum, minimum, range, mode.

Input:

- (i) (2D) The image to be measured.

Output:

- (i) (Scalar) The single valued measurement.

Parameters:

- (i) (ME) Mean/standard deviation/Maximum/Minimum/Range/Mode:
The statistic to be returned.



Distribution

Brief description: Measures the area distribution for a collection of white regions.

Input:

- (i) (2D) The image to be measured.

Output:

- (i) (1D) The unsorted list of blob areas.
- (ii) (1D) The histogram of blob area distributions.

A1.3.5 Miscellaneous interface elements

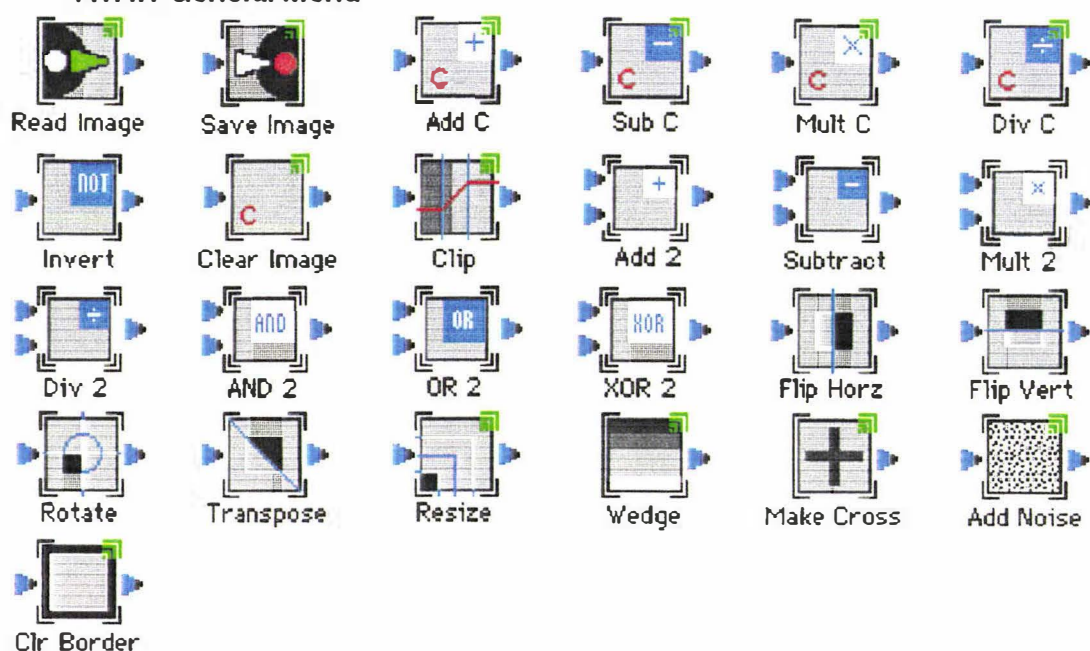


Subflow

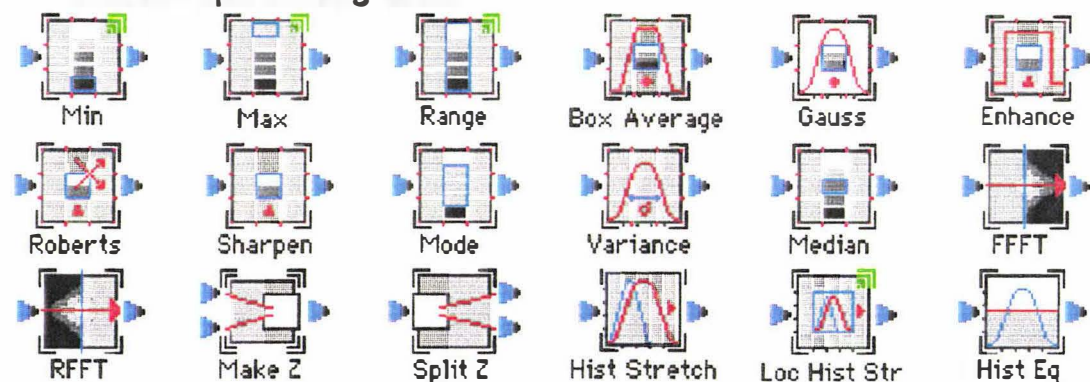
The icon for an OpShop subflow.

A1.4 Operation Icons

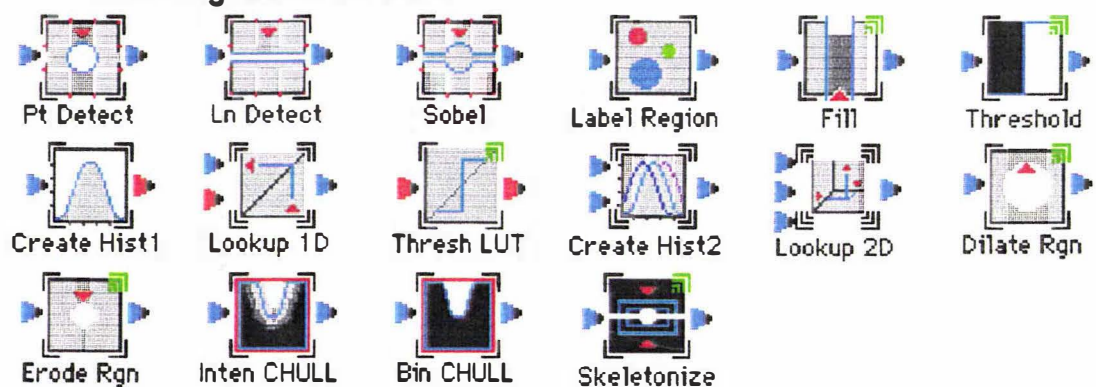
A1.4.1 General Menu



A1.4.2 Preprocessing Menu



A1.4.3 Segmentation Menu



A1.4.4 Measurement Menu

