Construction and Use of Research Tools for Image Processing

A Thesis submitted for the degree of

Doctor of Philosophy

of the University of London

by

Barry Michael Cook

Physics Department

Royal Holloway College

University of London

June 1983

B.M.COOK

Construction and use of Research Tools for Image Processing

ABSTRACT

Image processing now has a wide variety of applications and a large amount of algorithm development is required. Clearly, a convenient and easily used development system is a great advantage. Some preliminary work with an existing machine indicated that a carefully tailored interactive facility could provide such an environment.

An image storage unit containing a novel, fast, method of accessing the window to be processed has been constructed. By delegating to the storage unit some of the tasks normally performed by image processing software a considerable increase in processing speed has been achieved. While the improvement is not sufficient for an industrial system, it does allow for the convenient investigation of algorithms of considerably greater complexity than has hitherto been found possible on a moderately priced machine.

To make full use of the hardware and to provide a concise notation for the description of processing algorithms, a versatile computer language, PPL2, has been developed. PPL2 provides, in addition to an extensive range of operators, a very concise yet very efficient method of denoting image operations. A compiler for this language has been incorporated into a complete image processing system for fast interactive development and testing of programs.

Use has been made of the system to investigate the possible application of the quadtree in image processing and also for the formation of the skeleton description of an object. In the latter application interest centered around the possible advantages of a 5 x 5 over a 3 x 3 pixel window.

Awareness of the potential industrial applications of image processing has led to observations and comments on the hardware and software required for image processing. Conclusions are reached concerning the relative merits of parallel versus sequential algorithms and of various types of processors.

CONTENTS

Appendices

CHAPTER 1

CHAPTER 2

CHAPTER 3

A LANGUAGE FOR IMAGE PROCESSING.

CHAPTER 4

AN IMPLEMENTATION OF PPL2 ON A PDP11/34A

CHAPTER 5

IMAGE STORAGE AND ACCESS LOGIC

CHAPTER 6

AN APPLICATION OF THE GREY SCALE QUADTREE IN SMALL PART LOCATION

CHAPTER 7

SKELETONISATION

CHAPTER 8

CONCLUSIONS

LIST OF FIGURES

1. INTRODUCTION

Humans make very extensive use of vision and have built their environment on this basis. It is estimated that as much as 75% of the information received by humans is visual in origin, they are well adapted to using it in this form. Children very soon learn to recognise objects they see and analyse their surroundings using this advanced visual capacity.

For tasks in which people are less capable, machinery has been built to help them. These machines provide mechanical aids to the performance of duties not normally possible, for example handling very large or small loads or working in hostile environments. Assistance in numerical fields has been given by the electronic computer which can perform mathematical operations at a speed and precision far beyond human capabilities. The supremecy of the computer in this field has led to its being regarded as a mechanical brain which can be used to tirelessly perform the same tasks as humans. In many areas this has been achieved and computers can be found controlling complex operations as well as any person could.

Despite their high speed, complexity and apparent thinking ability computers are still very poor at communicating with the world in which people live. Advances have been made since the time when binary numbers had to be laboriously produced to feed the machine and the results, again in binary form, had to be carefully decoded to reveal the required answer. It is now possible to feed the machine instructions in a form not dissimilar to normal language and to receive results in printed or graphical form.

Vision is a sense still not truly available to a computer in that it is impossible to present a scene to a machine and expect it to interpret what it sees in a manner even remotely as complex as a human can do without conscious effort. The old adage that "a picture is worth a thousand words" becomes painfully obvious when describing even extremely simple scenes to a computer. It is not surprising that a great deal of effort has been and still is being expended on the problem of giving computers sight.

1.1 APPLICATIONS OF IMAGE PROCESSING

Given a machine capable of receiving and processing images there are many possible applications. Hall [1] has identified five major areas -

1. Image enhancement.
2. Object Reconstruction.
3. Communications.
4. Segmentation and description.
5. Scene matching and recognition.

1.1.1 Image Enhancement

Images that have been in some way degraded may need to be improved if they are to be useful. In cases where it is not possible to re-take a poor photograph or a clean signal is unobtainable, it is essential that enhancement be applied. A major example of this work is seen in the images produced by the Jet Propulsion Laboratory from satellite mounted cameras flown through the solar system photographing planets and their moons in passing. It has been possible to compensate for such image degradations as random noise, interference (non-random noise), geometrical distortion, field nonuniformity, contrast loss and blurring. Further application areas include imaging through the atmosphere where blurring due to air turbulence takes place and in criminal investigations where photographs are hastily taken and can be of very poor quality.

1.1.2 Object Reconstruction

Most physical objects are three-dimensional but cameras produce only two-dimensional images of them. The ability to extract information about the third dimension is invaluable. It is found in medical computer tomography which, using X-rays, provides information about concealed organs without the need for surgery. Industrially it is useful to determine such information as when objects are partially or completely occluded by others.

1.1.3 Communications

Image transmission of even moderate resolution requires a large bandwidth for moving scenes, the 625-line television standard needing some 6MHz. It is often found, however, that much of the information transmitted is redundant and could be removed to considerably reduce the bandwidth required. Processing capable of removing the redundant information and of reconstructing the image from what is left would be very useful in making feasible such devices as video-telephones using voice channel bandwidths.

1.1.4 Segmentation and Description

Verbal description of a scene requires the use of a very great deal of intelligence in the choice of features that are significant, and whilst occurring without much conscious effort in humans proves a difficult task for machines. Since much of our experience of the world comes from observation, the faculty of sight would be invaluable to a computer. The potential number of applications of seeing machines is immense. It has been recognised that the problem can be usefully divided into two parts, segmentation and description.

Segmentation of a scene into regions of similar properties can be treated as a two- or three-dimensional clustering problem in which clusters of points having similar properties can be grouped together. The property chosen to cluster points will depend on the application - for example regions of similar grey intensity or similar texture might be suitable in different problems.

Description of a scene relies on both the relationships of the segments of the scene to one another and the absolute properties of each segment. Structural descriptions are of particular interest in that they often approach the human description of the objects seen; for example a pencil can be described as a cylindrical object with one flat and one conical end.

In some cases it is better not to separate segmentation and description too fully as a partial description may be of use in the segmentation process. Figure 1.1 shows a part of a scene which appears fairly homogenous and might, reasonably, be considered a segment of the scene. Figure 1.2 shows the scene from which figure 1.1 was taken, the dotted box enclosing figure 1.1. In this case a region boundary can be seen; it is rather more obvious here because the easily identified part of the object at the top has been determined to be part of a circle - extrapolation of the expected edge into the unclear region makes it more visible.

1.1.5 Scene Matching and Recognition

Rather than having a detailed description of an object of interest it would be convenient to present one or more examples and to be able to recognise similar objects. This learning ability is such a natural

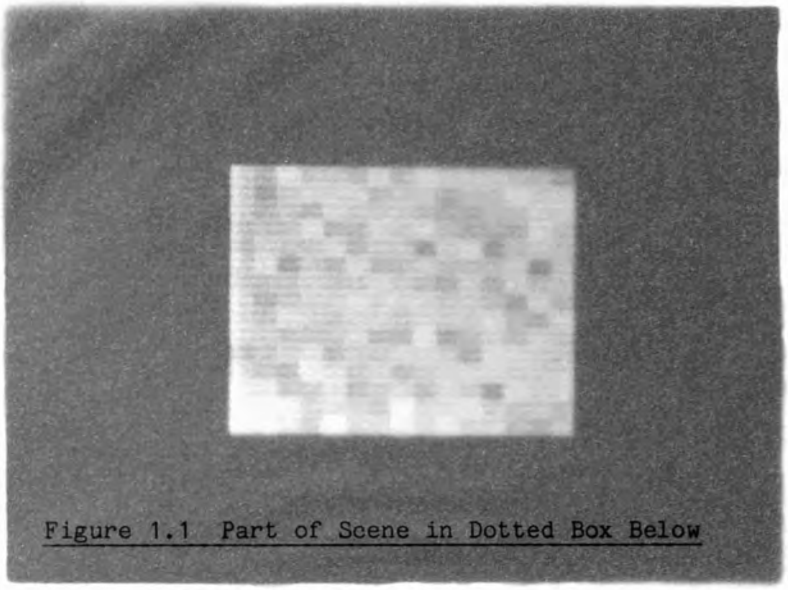


Figure 1.1 Part of Scene in Dotted Box Below

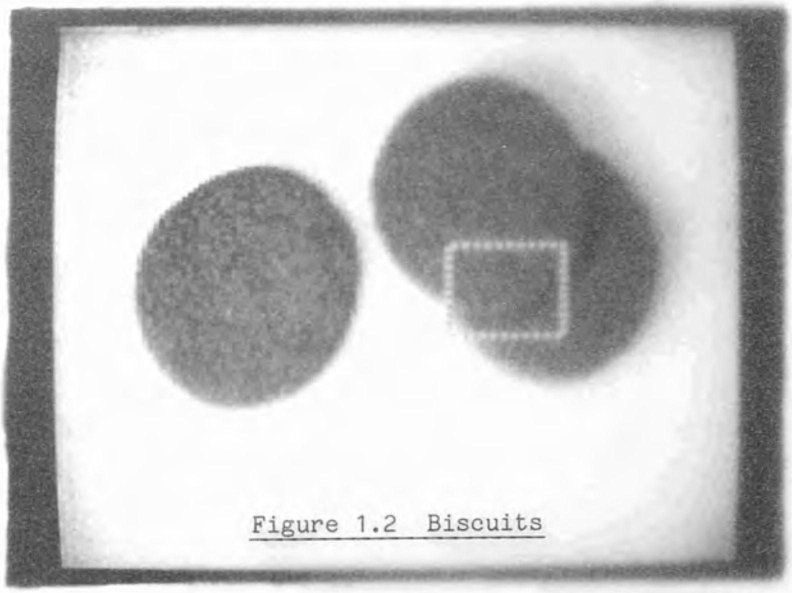
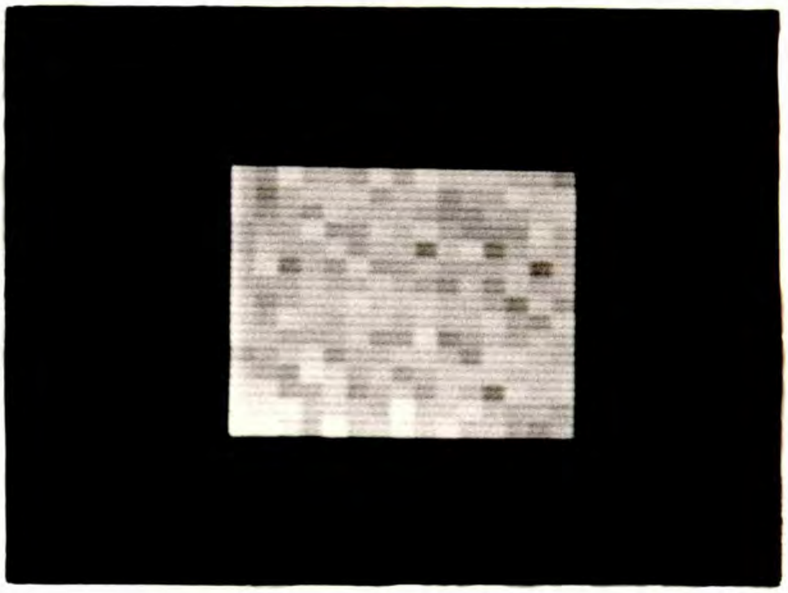
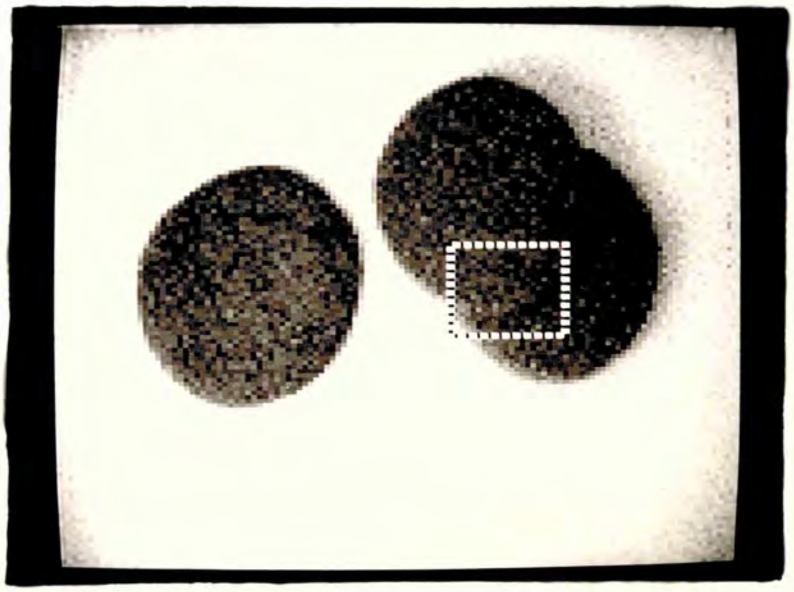


Figure 1.2 Biscuits





procedure for humans that it has been suggested that it is an inherent function of the eye-brain system. Computer programs capable of learning have been devised, some being based on the human vision system; see, for example [2].

1.2 INDUSTRIAL APPLICATIONS

Industrialists are showing increasing interest in image processing and pattern recognition for production applications. Most large factories are now highly mechanized in order to improve rate of output and consistency of product. There are, however, parts of the production cycle that can be satisfactorily performed by human operators. Many such jobs are highly repetitive and very boring with the result that by the end of a shift the error rate often rises to intolerable levels. There is considerable interest in mechanizing these functions to avoid the need to employ people in such mindless jobs and to improve output by reducing errors. There are two broad categories of requirements for such machinery, inspection and assembly.

1.2.1 Automatic Inspection

In order to ensure that a product meets the quality requirements it must be inspected before being passed to the customer. Most products undergo a number of steps in their manufacturing process, each step adding to the value of that part. It is obviously pointless to process further an already damaged part - it should be rejected as soon as possible. Unfortunately it is not usually possible to accommodate human inspectors at all stages in the production as many of them take place in unpleasant or dangerous surroundings, for example within fast moving machinery or at high temperatures. As a result the inspection often takes place at a very late stage in production after expensive processing of reject parts!

Availability of a mechanical inspection system would mean that much more inspection would be possible and soon pay for itself in saved processing of faulty parts. A further benefit of detailed inspection is the possibility of performing careful statistical tests on the products to predict the oncoming failure of a production machine to enable routine maintenance to take place and avoid a costly break in production.

1.2.2 Automatic Assembly

Another tedious job in the manufacture of a product is the assembly of its component parts. Careful presentation of the components with regard to their position and orientation can make mechanical assembly possible but the accuracy of location required implies a higher cost before assembly. An ideal assembler would take the required part from a mixed pile of parts so that there would be no need for elaborate packaging techniques but simply a method of tipping parts into a convenient area. The identification of single isolated parts is a problem that has been only partially solved: that of a "pile of parts" is significantly more difficult and no general solution exists, although in some particular applications a satisfactory performance has been achieved.

1.3 IMAGE SOURCES

Before a computer can attempt to extract information from a scene it must be presented with the information contained therein in a suitable format. Computers deal with numeric values represented by electrical signals and some form of transducer must be used to convert optical data to this form. There are many devices available for this purpose but most provide an analogue signal which must be further processed to digital form.

1.3.1 Television Cameras

Conversion of optical information to electrical signals was first employed for the transmission of scenes in the form of television. Much effort has been put into developing suitable cameras for this purpose and all produce signals to one of a few common standards. It is thus possible to choose a camera well suited to a particular application; very many different types exist for a great number of special purposes. The long period of development has resulted in a range of cheap and reliable signal sources which may be used for input of image information.

1.3.2 Charge Coupled Devices

Solid-state cameras using charge coupled device technology to produce moderate resolution images are now available. Current development work promises high resolution devices in the not too distant future. The sensor can be either a single line or an array of photocells, the former being known as a line scan camera and the latter producing a standard television signal. To produce a distortion free image from a standard television system camera the subject has to be stationary for a complete frame scan. This is not always convenient in applications involving, for example, objects on a production line which is continuously moving. The line scan camera, however, relies on the subject moving across its line of view to produce a complete raster scan and is more suited to the type of application mentioned above.

1.3.3 Other Sources

Data for the image processor may not have come directly from a camera but possibly via a communications link (as for satellite pictures), or from a non-optical sensor, or an X-ray detector in a medical application. Provided the signals can be converted into a form that can be interpreted by the image processor (generally, a set of digital values) the exact source is unimportant.

1.4 DIGITISATION

Before it can be processed the signal representing the image must be converted into a form that the computer can use; this is generally a set of digital values. The image is, initially, a continuous field of values and must be sampled at intervals and the samples digitised to yield actual values. At each sampled point a characteristic of the signal is measured and represented by a digital value; the most usual characteristic of interest in an image is the brightness of the point. The resolution to which the characteristic is digitised depends upon the image source and the application; for television signals between 1- and 8-bits resolution are employed, corresponding to 2- (binary) to 256-levels of brightness.

Sampling must occur at regular intervals over the image to provide meaningful data. It is usual to consider the image as being covered by a regular tessellation of one shape with the sampling occurring at the centre point of that shape. There are only three basic shapes that can be used to cover an area, triangles, rectangles and hexagons [3]. Triangles are rarely used for image processing but both rectangles and hexagons are seen, rectangles being the most common.

1.4.1 Connectivity

Each image point has a number of neighbours and may be considered as connected to some or all of them, this is known as the connectedness of the point. An image digitised on a hexagonal grid is 6-connected, that is each image point is connected to 6 neighbours; a rectangularly digitised image can be considered as either 4- or 8-connected, see figure 1.3 and [3].

Choice of 4- or 8-connected pixels (in the case of rectangularly digitised images) is not arbitrary. Consider the following arrangement of pixels -

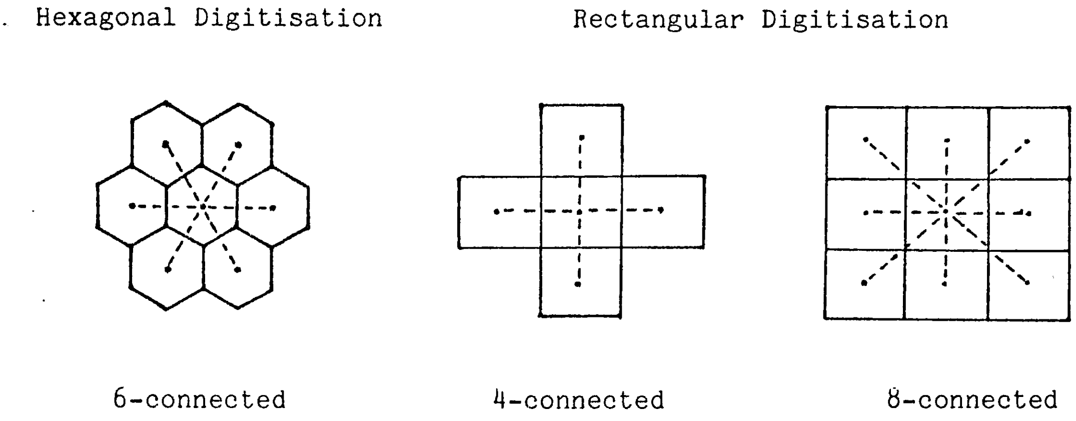
```
0 1
1 0
```

Considering the pixels as 4-connected means that the 1´s are connected to form a line but so are the 0´s; this causes a conflict in interpretation of the scene. To overcome this paradox it is usual to consider either foreground or background to be 4-connected and the other to be 8-connected. By defining the background to be 8-connected and the foreground to be 4-connected objects that meet only at corners are not considered to be touching.

1.5 IMAGE PROCESSING

Data as input is not generally in the required format for the application. For example it may be that the number and positions of certain objects are wanted and the data available is an array of values. It is necessary to convert the information from the input device into another format for subsequent use: it may be another image or, as in the example above, into a completely new form.

Generally the value of an output pixel can be defined as a function of the input pixels. A complete definition in this form would



Hexagonal Digitisation

Rectangular Digitisation

6-connected

4-connected

8-connected

Figure 1.3 Connectedness

be impossibly large, there being as many functions as pixels in the output image, each taking the total number of input image pixels as parameters. For a 128 x 128 pixel image there would be 16 000 functions each with 16 000 parameters! Fortunately it is possible to use more local functions and to apply them to all points in the image, thus requiring fewer functions with fewer parameters. The price to be paid is that the desired output may not be obtained with a single function but several may need to be applied one after another.

Single parameter functions are the simplest that can usefully applied; for these the value of the output pixel is a function only of the corresponding input pixel. (Parameterless functions may only produce a fixed value and are only useful for setting the whole image to a single value.) The function may be linear or non-linear. This simple type of function is useful for such operations as adjusting grey levels and contrast within an image, often in conjunction with data obtained from a histogram of the brightness distributions within the image.

More useful functions take a number of parameters, usually small, from a region around a central point. Repeated application can spread information through the image so that the final result is a function of all pixels in the image. The small region from which parameters are derived is known as a window on the image since only those pixels within it can be "seen" by the function. A commonly used window is 3 x 3 pixels from which 5 (4-connected) or 9 (8-connected) parameters may be taken. A very large number of functions with 5 or 9 parameters may be derived and a much larger set constructed by concatenating them. Windows larger than 3 x 3 are much less frequently seen as the difficulties in designing them rapidly increase. An exception in which much larger windows are found is the case of functions derived by sound mathematical techniques such as Fourier transforms. One application in which the 3 x 3 window provides less than the ideal amount of information is that of finding the skeleton of a shape, although most work has used this size. Chapter 7 implements one skeleton finding method and investigates the advantage of a larger (5 x 5) window.

1.5.1 Parallel and Sequential Algorithms

Window functions must be applied at all points on the image to produce a complete output image. It is possible to consider the function as being performed by as many processors as there are output pixels working in parallel or by a single processor operating on each point in turn. In the case of a complete set of parallel processors the result is a function of only the window elements. In the case of a single processor a window function is derived and the processor moved to the next pixel in an ordered scan of the image. The values obtained may be placed into a separate output image producing the same output as a parallel array of processors. Alternatively they may be returned to the input image so that the next application of the function uses the result of the previous applications: this is known as a sequential procedure.

1.6 SYSTEM REQUIREMENTS

Image processing, like most practical subjects, can be categorised into two different areas of use each with different requirements. Initially a system for research and development is required, the results from which are used to construct a production system. Whilst both systems have a common requirement, in this case to process images, there are many differences between them. The production system must be as simple as possible to aid reliability and serviceability, have as few controls as possible to prevent mal-adjustment, and be adequately fast. A research machine, while sharing the requirements of reliability, must yield a vast quantity of detailed information concerning the processes it is performing to allow careful study; it must be flexible to allow changes to be made and tested and easy enough to use that research is not an uphill struggle. Research is, clearly, the first step in the process and this thesis describes a system designed to facilitate investigations into image processing problems and their solutions.

Central to the task of image processing is the description to the computer doing the calculation of the actions required. A number of languages exist that may be used for this purpose but none in common use provides a clear efficient method of coding window operations that must be performed over the image. A short-hand notation for the pixels in the window not only makes the program clearer but also simplifies writing and, in turn, reduces errors. A simple method of indicating that the operation is to be performed over the whole image is of immense value in uncluttering programs, as will be demonstrated in chapter 3. A further benefit gained from the use of a clear programming language is that the program itself can be used as a description of the task, thus avoiding transcription errors. A disadvantage of subroutine or macro libraries, such as those used in [4], is that the actual processing taking place is hidden from view and a fairly large effort is involved in understanding it and in making any changes that may be required.

Greatest interaction can be obtained from an interpretive language (cf BASIC) but, unfortunately, such languages are relatively slow in execution. When an operation is to be performed over a whole image it must be repeated a large number of times (16 384 for a 128 x 128 image) and execution speed is highly relevant. A complete program is composed of not only image operations but also structures to control the flow of these operations. PPL2 was designed to adopt a middle course in that the control statements are interpreted while the image operations are locally compiled and executed. Thus, except when actually operating on an image, full interaction is maintained and short programs may be entered from the keyboard for immediate execution. Many people have noted that the writing of software can form a notably expensive part of the project whilst actually being only a small fraction of the total design. It is therefore particularly important that an easy to use design system be available to reduce the disproportionate time and costs involved.

1.7 PAST AND PRESENT

PPL was first conceived in 1977 for use on a small microprocessor based system [5] and saw extensive use on this machine. When a larger machine became available it was reviewed in the light of past experience and future requirements and a new language, PPL2, designed.

A language is of little use without supporting facilities and these were provided by calling upon existing systems programs and by writing a sub-system for image processing.

In this thesis chapter 2 looks (briefly) at the architectural considerations that coloured the design of PPL2 both in terms of the computers used for image processing and of the software structures required. Chapter 3 discusses some wider issues affecting the language and describes PPL2. The implementation of PPL2 as a compiler/interpreter is described in chapter 4, and a user guide to the complete sub-system appears as appendix C. In order to improve the execution speed and image storage capability of the system a special image storage unit was constructed and its features are described in chapter 5.

The system developed contains many of the features required by a general purpose computer vision system [6]. Chapters 6 and 7 describe two of the very many investigations into image processing algorithms performed on this system. Some tasks were very small, as simple as one or two line programs (when written in PPL2) whilst others required several different files of programs to be used.

Finally, chapter 8 highlights some of the problems encountered in image processing algorithms, the languages used to program them and also in the computers used for their execution, suggesting possible improvements for future implementations.

2. SOME ARCHITECTURAL CONSIDERATIONS IN DIGITAL IMAGE PROCESSING

2.1 Introduction

Image processing is a highly practical subject in which it is not possible to progress very far without testing ideas on real scenes. There is a difference between the requirements of a researcher and an engineer when processing images. The researcher is relatively unconstrained by time but requires detailed results for further study. However, a long execution time will limit the quantity of test data investigated. The engineer may be severely limited in the time available to perform a function when applied, for example, to a production line in a factory.

Trends in computing have been away from the use of low-level languages toward higher-level languages that can be written more quickly and are more machine independent for portability. An unfortunate disadvantage of the use of higher level languages is the programmer´s inability to fully utilize special conditions to provide faster programs. This has not been a problem in commercial programming where the computer has spare capacity for the job and the programmer´s time is very expensive. Image processing has seen a change in conditions where the computer is rarely fast enough and effort has to be placed in producing highly efficient programs.

Comments in this chapter are intended to give some ideas of ways in which the image processing task can be performed faster. Sections 2.2 and 2.3 briefly describe some of the machinery that may be used for image processing. Section 2.4 looks at some of the considerations that may be applied when choosing a programming language. Some techniques that may be incorporated into programs to improve their speed are described in section 2.5. ,

2.2 SEQUENTIAL PROCESSORS

A wide range of sequential processors is available; many are not specifically designed for image processing but, being general purpose, can be used for this application. Where there are good reasons for doing so, such as high speed, specially designed sequential image processors are found.

2.2.1 Main-Frame Computers

Main-frame computers have been the traditional tool for research into image processing - often because these were the only machines available. Such machines usually provide for large amounts of data handling and storage with a high speed processing capability. Some recent machines can provide an extremely fast processing capability for the type of data found in image processing [7].

It would seem that this type of computer is ideally suited to the task; however there are many disadvantages which make its usefulness rather dubious. Most of these problems stem from the fact that such a computer is very expensive and can therefore only be justified if it is to serve a number of users. The large space these machines occupy and the services they require, effectively rule out their use in a factory environment. Even if such a machine could be justified on the grounds that it could service several processes, the effect of a single breakdown on the whole of the factory output should be compared with the more localised effect when using several smaller computers.

Operating systems for these machines usually provide either batch processing or time sharing between users. The former is not conducive to the writing of efficient algorithms or to real time use, as the results of a run may take many hours or even days to become available.

The latter type of system slows down the response to each individual user so that it may take several minutes to get a few seconds of processing time, again precluding serious real time work.

Perhaps the worst problem in image processing is the quantity of data involved. This has to be transferred to storage within the computer system before it can be used. An image of 128 by 128 pixels digitised to 8-bit resolution occupies 16 384 bytes which, transferred over a 1200 baud line, requires more than 2 minutes of transmission time. At this speed interactive programming is very tedious and not at all conducive to development work.

2.2.2 Minicomputers

Although relatively short of memory and slower than the main frame computer, a ´mini´ has distinct advantages which make it very suitable for image processing. Usually it is possible to gain complete control of the machine and use the whole of the processing time available. The machine can be opened up without incurring the wrath of the computer centre staff and special interfaces can be installed for high speed data transfer. Despite the relative lack of processing speed these other advantages result in faster program execution than the main frame. Most of the memory requirement is for image storage and can be placed outside the computer with an appropriate access system (see, for example, chapter 5).

2.2.3 Microprocessors

Since they are available at very little cost microprocessors have aroused considerable interest. Unfortunately, when compared with main-frame and mini computers, they have been rather slow - a factor which could limit their effectiveness for image processing. The cheapness of these devices means that the use of multi-processor techniques is quite feasible and leads to a considerable increase in speed. Early processors were also limited in their lack of certain basic functions, such as multiply and divide, which incurred a time penalty in the software required to perform the functions. Recent additions to the microprocessor range (such as the Motorola 68000, National Semiconductor 16032, etc.) provide these functions and speed improvements to the same order as those of mini-computers.

2.2.4 Bit Slice Processors

Bit slice processors are special forms of microprocessors which are very fast and can be microprogrammed. Generally they are available in the form of a 4-bit processor per package, but with the provision for connecting several together for long data words. The ability to micro-program them means that any special instructions required for image processing can be provided, which, when compared with the fixed set of a microprocessor gives a distinct speed advantage.

2.2.5 Shift Register Processors

Image data for the above mentioned processors is stored as a static array of values with access restricted to one value at a time. As many accesses are required as there are points in the window in order to extract data for each window operation.

The shift register processor provides an architecture whereby the data is dynamic and all pixels in a window are available at one time - see figure 2.1. (The immediate availability of all pixels in the window has led to its use for the simulation of a parallel processor [8].)

Each of the long registers holds one line of image data to provide

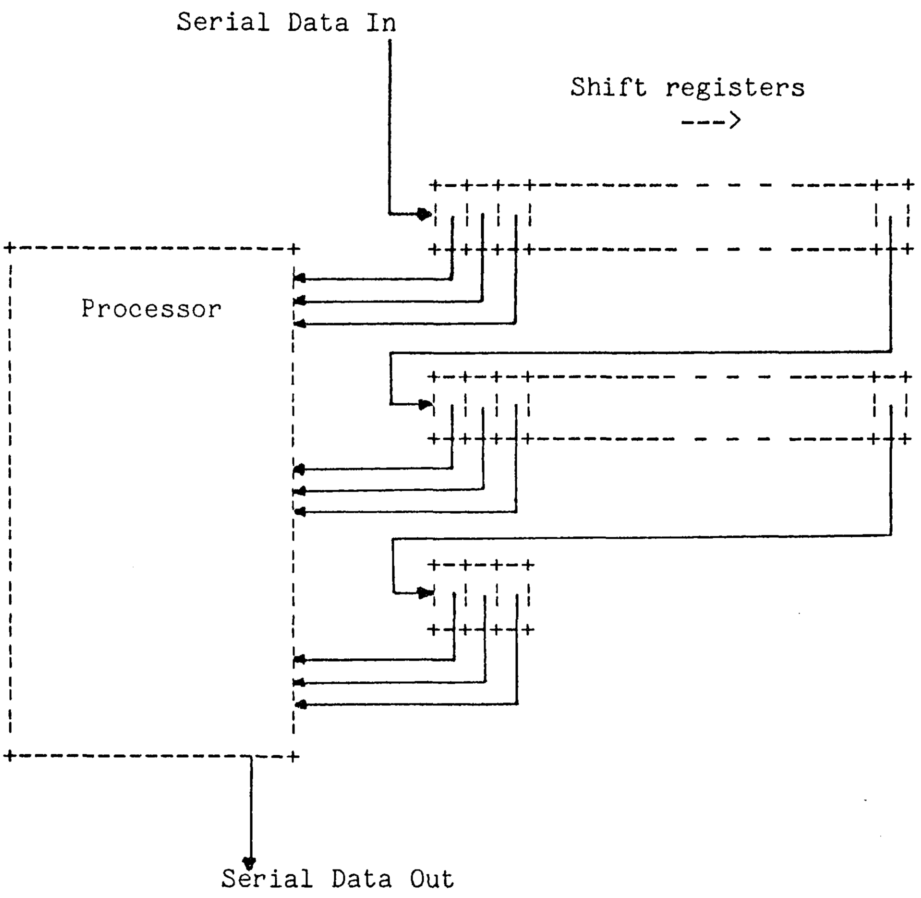


Figure 2.1 Shift Register Processor

the delay required to present all window points in parallel to the processor; the latter then produces a serial data stream as output. The processed data stream can be passed to a further shift register processor. Cascading processors in this way does not reduce the speed at which images can be handled but merely increases the propagation delay in handling data.

The processor has to handle the data in real time. For a standard 625 line (CCIR) video signal digitised to 128 by 128 pixels, a processing time of at most 400nS is required - too fast for most microcomputers. In the particular case of a 3 x 3 window processing a binary image there are only $2^9 = 512$ possible combinations of input data which may be processed by simply looking up the result in a table. Memories with access times of better than 50nS are available making this type of processing quite feasible.

Data from a slower device, such as a CCD camera, could probably be processed with a small computer, or maybe a set of such devices.

2.2.6 Hard Wired Logic

Undoubtedly the fastest solution to image processing is the construction of special dedicated circuitry. Unfortunately this is very inflexible and any changes, however minor, to the algorithm require a major reconstruction. Some standard operations that will remain invariant may beneficially be implemented in this way, particularly in conjunction with the shift register processor described above.

2.3 PARALLEL PROCESSORS

The ability to process all the points of an image together in a suitable parallel processor was considered desirable and designed as long ago as 1958 [9] but technology had not, then, advanced to the point that implementation was practical and worthwhile. A small (36 x 36) array of this form was later simulated in 1959 by Unger [10], producing results that caused that author to suggest that character recognition at the rate of 2500 characters per second would be achieved within five years.

Later developments led to such parallel processors as the ILLIAC III with 1024 processors [11] and the ILLIAC IV, with an array of 256 processors [12]. A later (1973) machine, described as parallel but, in fact, a shift register processor implementation for images of 64 x 64 pixels was constructed by Kruse [8]. At about this time a truly parallel processor aimed specifically at image processing emerged from Duff [13] with an array size of 12 x 16 pixels.

Other application areas for parallel processors had, by the early 1970's, arisen and main-frame manufacturers started introducing them as integral parts of their machines. One recent example of this being the Distributed Array Processor (DAP) of the ICL 2980 series machines [14,15]. (The cells of the DAP are only 4-connected but it is possible to simulate 8-connectedness without too large an overhead [15].) Meanwhile, Duff had been developing large scale integrated circuits for a more powerful image processing machine, CLIP 4 [16,17].

CLIP 4 has brought the ability to process quite large (96 x 96 pixel) images at very high speed and is now available complete with mini-computer controller and high-level language [18]. This is undoubtedly a powerful machine but is, unfortunately, also rather expensive and has some problems with the availability of components

[19]. Since the design of CLIP other parallel machines have been produced, for example [20].

2.4 THE PROGRAMMING LANGUAGE

Writing instructions for a computer by feeding it series of numeric instructions via a set of switches is no longer considered a sensible method of programming. Some method of writing the instructions in a manner more easily understood is required.

Early in the life of computing a simple mnemonic code was used to write programs in the machine′s internal code, but with the advantage of increased readability. As programs became more complicated it was evident that rather more powerful constructions had to be simply described.

Languages which described particularly useful ideas in the field of application were evolved. Perhaps the earliest well known such language is FORTRAN, intended to be used for describing numerical operations. A vast number of special languages have been devised for particular applications. There is not, as yet, a universal language for all applications (although there have been attempts to produce them). When programming for a particular application a language providing the type of facilities most often required must be chosen.

2.4.1 High or Low Level

Languages closely linked to the architecture of the computer are known as low-level. Those which provide a program description which is (relatively) independent of the machine are known as high-level languages.

Using a high level language to write programs can bring significant advantages for the user. Concepts rather than specific instructions are written and a compiler used to generate the actual instructions needed by the computer to carry out the task. Only a few such commands are required to specify a large number of machine operations. It has been found that programmers tend to produce code at a constant rate, regardless of language, so that writing at a high level produces more in a given time [21]. Since the instructions that are written are relatively easy to understand, the program is less likely to include errors; those that do creep in are generally easier to find and correct. The lack of machine dependence provides for the easy transportation of programs to different computers; this has been considered one of the major advantages of using high level languages. Unfortunately, even at the best of times, transportation is not as easy as it should be. There is often a dependence upon, for example, the word length of the computer which affects the precision to which values can be stored. The problem is compounded in image processing by the lack of standard image storage and access machinery resulting in considerable program tailoring before use. This is a problem even with FORTRAN. No doubt this will become less troublesome as standard units become available.

Low-level languages, on the other hand, tend to be long and rather obscure in their detailed operation. To counteract this it is very easy to make use of any special features provided by the computer or image storage system. Careful programming in assembly language can produce highly efficient and fast programs that even the best of compilers cannot match. The provision of ´macro´ facilities can greatly assist in the writing of clear programs without sacrificing speed in execution. Assembly language programs can only be transferred to other computers of the same type and are not at all suitable for the wide distribution of programs.

Perhaps a compromise should be sought for image processing; the major flow of the program can be controlled by a high-level language program with small subroutines at low-level to provide high speed operation and interfaces to special devices.

2.4.2 Program Structures

Much of the development in high level languages has been in program and data structures. Clearer methods of representing the required program flow not only make programming easier but also lead to fewer errors.

Very few basic instructions are needed to control the flow of a program. All languages eventually use those provided by the computer itself - conditional and unconditional jumps to new sections of machine code. As anyone who has looked at a complex machine code program will know, these basic instructions do not produce easily understood programs. To combat this problem higher level structures are usually provided and the compiler is left to generate the low-level structures.

Early languages often reflected the machine architecture in its provision of high-level structures - the IF statement of FORTRAN 4, for example. It is now recognised that to be genuinely useful rather more complex structures are required.

Conditional statements, as typified by the occurrence of the word IF somewhere in them, have changed considerably in the history of computing. Early languages strongly reflected the computer´s own machine code in providing only a jump to a new section of the program - for example in early versions of BAS1C. This was soon found to be too restrictive and conditional statements which caused a general action to be taken upon the result of a test were included, c.f. later versions of BASIC and the logical-IF of FORTRAN 4. Action often has to be taken when the condition is found to be false as well as when it is true and a clear method of doing so provided in languages such as ALGOL60, ALGOL68 and, lately FORTRAN 77. The final form, as typified in the ALGOL68 implementation, not only provides a clear description of the actions required but also encourages good programming. A speed improvement, at the expense of clarity, can be obtained in some circumstances by returning to a low level implementation (see section 2.6)

Controlled program repetition can be achieved with a conditional statement and an unconditional loop back to the start of the program section to be repeated. Machine code can only use this method, with the result that programs are rarely clearly understood. Higher level notation for such repetition is generally provided. Mathematical languages, especially those requiring the use of matrices, generally employ a simple repeating structure with an index variable taking on successive values. The DO loop of FORTRAN is an example of such a structure, the index variable taking a range of positive values, perfectly adequate for array indexing. As computers were required to be used for more general purposes this loop proved inadequate, the lack of negative values or negative increments requiring extra programming. A far more general looping command is provided in ALGOL60 which allows the variable to take values from a list of sub-ranges. As well as needing an indexing loop, a control structure that repeats a section of program until a condition is met or while a condition is true is required, these have been provided as extensions of the DO loop in PASCAL and ALGOL68 respectively. The REPEAT ... UNTIL ... of PASCAL can be implemented within ALGOL68s WHILE ... DO ... OD if required. Again some speed improvements can be achieved by re-coding in machine code - see section 2.11.

Definition of an operation to be performed over the whole image array is a structure peculiar to image processing and not to be found in the well known high-level languages. The simplest structure is a pair of nested loops to scan the image with a processing operation within them. A simple notation for this ([[...]]) is employed in PPL2, see chapter 3, which uses information about the image storage hardware to implement an efficient loop.

2.4.3 Data Structures

Data structuring is extremely useful for dealing with complex inter-relationships between data items, especially when they are dynamically changing. This structuring is very useful for re-ordering data in pattern recognition but is not often required for image processing as such. In order to allow the description of relationships, extra information must be appended to the data. As yet there are no computers available that automatically deal with this extra information and extra programming (and hence, time) is required to make use of it.

Data may, very usefully, be given a data type (integer, real, etc.) and the compiler be required to check that valid combinations of data types are being used. Such typing can allow the shortest required data representation to be used, and hence the fastest execution time to be obtained.

Languages such as ALGOL68, which allow dynamic type changing and expect execution time type checking, also incur a time penalty (until such time as a computer with hardware for this purpose is available).

Pyramidal image descriptions (see Chapter 6) do require the use of structured data as the new image description and a language incorporating these features must be used. For those applications where images are processed to form other images, data structuring is not generally required.

2.4.4 Operators

Image processing is unusual in computing terms in requiring not only the standard set of mathematical operators but also a comprehensive range of bit-level operators, such as boolean AND, OR and NOT. Of the languages that do provide such bit-level operators most require the use of special words rather than symbols for their inclusion in a program. This is inconvenient when such operators are used often. PPL2 (chapter 3) assigns single character symbols to the set of boolean operators for ease of use.

The ability to define new operators if they are found to be frequently required is useful. FORTRAN allows both one line definitions and complete routines to perform this function; unfortunately, alphanumeric names must be chosen. ALGOL68, on the other hand, does allow such symbolic operators to be defined. Whenever possible the function should be written in machine code using any basic instructions that are available for speed in execution.

Years of programming for mathematical and commercial data processing have shown computer designers what types of basic operations should be provided by the machine. There is not as much experience available to call upon in the programming of image processing algorithms with the result that some operations may have to be implemented relatively slowly in software. Examples of such operators that I have found are needed often are the maximum and minimum of two values: these could be incorporated into a computer´s instruction set but instead a sequence of operations is required. One advantage of using a bit slice processor is the ability to micro-program it to perform such functions at high speed.

2.4.5 Subroutines

Groups of instructions that perform a function required in several places in the program need not be duplicated but instead a method of accessing them from each place is used. Such groups of instructions are known as subroutines: they are started by calling them and upon completion they return to the calling program. If the subroutine has to operate on different data each time it is called then some method of passing the data to the subroutine is required. Unfortunately, passing data to and from the subroutine results in a time overhead which would not occur if the group of instructions were rewritten each time they were required with explicit references to the data. This reduction of size in the program can hence cause an increase in the time taken to execute it.

The method chosen for data passing is dependant upon the facilities provided by the language. Those, such as ALGOL68, that provide for recursive subroutine calling need to pass data on a stack, whilst those, such as FORTRAN which do not provide this facility can use a pre-determined data area. The fastest method of data transfer is to use the machine's registers, but this is only workable if there are only a few items to be passed. Since the compiler has to be general purpose, this method cannot be implemented in a high-level language. Most compilers for non-recursive languages, however, put the result of the subroutine execution in a register as there is, generally, only a limited amount of data that can be returned.

Overheads incurred in calling a subroutine will extend the execution time of the program by a proportion that will depend on the amount of processing performed in the routine. Sections of program that are short would be better not called as subroutines but repeated as necessary. It is important to note that any code repeated a large number of times, for example a window operator to be performed at each point in the image, will see any overheads multiplied by the same number. As a general rule subroutines should be avoided in window operations.

2.5 EFFICIENT PROGRAMMING TECHNIQUES

Programs must accurately describe the operation to be performed in as clear a way as possible. For anything but the very simplest of algorithms there will be a large number of possible solutions. Each of the solutions will require a certain amount of time to execute, and not all will be the same. For image processing there is a great need for fast programs.

Optimisation for speed can be performed, to some extent, by the compiler of a high level language but even the best cannot quite match a good programmer. Some possible speed improvements cannot be left to a compiler as they require knowledge that is not available to it (see, for example section 2.5.3).

Improvements can be made to both high- and low-level language programs. Unfortunately, some of the more cunning tricks are not possible at a high level and recourse has to be made to a low-level language to squeeze out the last ounce of speed.

Some of the techniques that I have found useful for speeding up programs are presented here. Some are applicable to both high- and low-level languages, others only to low-level programs.

2.5.1 Minimal Loop Contents

Any instructions contained within a loop are executed each time round the loop. Care should be taken not to waste time by repeating the same calculation unnecessarily. For example the section of program

```
FOR I FROM 0 TO 255 DO
  IF ARRAY[I] > N/2 THEN COUNT:=COUNT+1 FI
                  OD
```

calculates N/2 some 256 times; once would have been enough. A better program would be

```
HALFN:=N/2;
FOR I FROM 0 TO 255 DO
  IF ARRAY[I] < HALFN THEN COUNT:=COUNT+1 FI
                  OD
```

Advanced optimising compilers may notice the repetition and remove it, however it is better to write the latter version to be sure of wasting no time. If hand compilation and optimisation is intended to take place the second version is far easier to deal with.

2.5.2 Common Sub-Expressions

Occasionally a sub-expression is repeated in different places so that the same expression is calculated several times. This is a milder form of the repetition above but nevertheless should be eliminated. Thus the command

```
ARRAY[I+J-4]:=ARRAY[I+J-4]*3+ARRAY[I+J-3]
```

should be re-written as

```
IJ:=I+J-4;
ARRAY[IJ]:=ARRAY[IJ]*3+ARRAY[IJ+1]
```

Most good compilers will notice this, at least locally, and implement the improved version.

If a good common expression eliminator is built into the compiler the second form of the program may be less efficient than the first. The use of an explicit variable, IJ, forces the compiler to store the result of the sub-expression rather than simply using a register temporarily. The only way of checking this is to dump the machine code produced by the compiler and study it carefully!

2.5.3 Conditional Statements

Conditional or ´IF´ statements occur frequently in any decision making process. The basic statement is easily compiled but compound statements can often be improved.

Consider the statement

IF A=B ! C=D THEN {then part} ... ELSE {else part} ... FI .

It would be normal to compare A with B to produce a boolean result, then compare C with D to produce a second boolean result, OR these together and test the final value to determine which of the THEN or ELSE parts is to be executed.

If A is found to be equal to B it is immediately possible to proceed to the THEN part without ever testing whether C is equal to D or not. This gives, at worst the same execution time as the normal scheme above, and at best requires only a single test. The assembly code implementation of such a scheme might look something like this-

```
        CMP     A,B
        BEQ     THEN
        CMP     C,D
        BNE     ELSE
THEN:   ...     ...  ; then part
        ...     ...
        BR      END
ELSE:   ...     ...  ; else part
        ...     ...
END:
```

To make the fullest use of this scheme the condition most likely to be true is placed first in the sequence of tests.

A similar method is applicable when the conditions tested are ANDed together, the first false value to be found forcing control to proceed to the else part. In this case the test most likely to be false is placed first in the sequence.

Execution can be considerably speeded up in this way, especially if the different parts of the conditional occur with differing probabilities.

While a good compiler can employ this basic scheme it cannot appropriately order the tests for the best results as it does not know the relative probabilities of the terms. It is not always certain that the terms will be tested in the order in which they are written down and care should be exercised when using high level languages in this way. If at all possible the machine code produced by the compiler should be checked. This ambiguity does not occur when statements are hand compiled or the program is written in a low level language.

2.5.4 Window Access

By far the largest number of data accesses occur to pixels within the image. Any unnecessary overheads here will drastically extend the execution time of the algorithm.

Images are two-dimensional arrays of pixels, any particular pixel being referenced by a pair of index values, one for each dimension. When a pixel is referenced in this way the two index values must be combined with the start address of the image to produce the address of the required pixel.

Arrays are stored in memory as a set of contiguous locations. Storage may be allocated on a row by row or a column by column basis, FORTRAN employs the row convention and ALGOL the column convention (hence their incompatability). This is illustrated in figure 2.2. Given an N by M array the offset of a general point with indices X,Y can be calculated as X+N*Y in FORTRAN and X*M+Y in ALGOL.

Language :	FORTRAN	ALGOL68
Array Declaration :	INTEGER A(2,4)	[2,4] INT A;
Memory address		
A+0	A(1,1)	A[1,1]
A+1	A(2,1)	A[1,2]
A+2	A(1,2)	A[1,3]
A+3	A(2,2)	A[1,4]
A+4	A(1,3)	A[2,1]
A+5	A(2,3)	A[2,2]
A+6	A(1,4)	A[2,3]
A+7	A(2,4)	A[2,4]

Figure 2.2 Array Storage Comparison

It is seen that for each pixel access a multiplication and an addition are required. Both these operations take time, the multiplication may be quite slow on machines without hardware multiplication (most of the earlier microprocessors). If the machine does not have a suitable indexing arrangement a further addition is required to add the base address of the image to the offset to form the actual address of the pixel. In the case of fixed offsets (i.e. X and Y above are constants) a compile time calculation can be performed to alleviate this overhead.

Pixels within a window are referenced by the address of the centre point of the pixel and an offset relative to this point. Thus, for example, to access the window point defined as IMAGE(X+1,Y-2), where IMAGE is a 128 by 128 pixel array, the following sequence of instructions would be required -

```
MOV     Y,R1            ; R1:=Y
ADD     #-2,R1          ; R1:=Y-2
MUL     R1,#128.        ; R1:=(Y-2)*128
ADD     X,R1            ; R1:=(Y-2)*128+X
INC     R1              ; R1:=(Y-2)*128+X+1
MOV     IMAGE(R1),R0    ; R0:=IMAGE(X+1,Y-2)
```

This calculation must be performed for each pixel within the window for each position of the window. If any pixel is accessed more than once for any given position of the window the calculation is repeated as an overhead. One possible solution is to transfer all the pixel values within the window to known locations from which they may be used any number of times without any overhead. Any resulting values must be written back into the image array before moving the window to its next position. It is useful for such operations as averaging to place the pixels in contiguous locations so that they may be accessed as a one-dimensional array of window points. See figure 2.3.

Using external memory to store the images allows this re-mapping to be performed by fast hardware so that the overhead is minimised. The hardware described in chapter 5 performs the mapping in very little more than the normal memory access time so that the overhead virtually disappears.

2.5.5 Edge Detection

When a window is near the edge of the image some of the points within it are off the edge of the image and have no defined values. The result of processing such an undefined value is meaningless. In order to take some sensible action in such cases the fact that the pixel is not on the image must be detected. After detection suitable action must be taken - often the substitution of a constant value which is considered to form a border around the image. Detection of an out of range value can result in a large program overhead considerably extending the execution time. Testing must be performed each time the address of a pixel is calculated by checking for validity in both the X and Y axes. For most of the image scan the pixel addresses generated will be well within the image. For a small fraction of the time (only 4% for a 3 by 3 window on a 128 by 128 image) an out of range condition exists and must be dealt with.

Original Mapping

IMAGE(X-1,Y-1)	IMAGE(X,Y-1)	IMAGE(X+1,Y-1)
IMAGE(X-1,Y)	IMAGE(X,Y)	IMAGE(X+1,Y)
IMAGE(X-1,Y+1)	IMAGE(X,Y+1)	IMAGE(X+1,Y+1)

Re-mapped Window

IM+0	=	IMAGE(X,Y)
IM+1	=	IMAGE(X+1,Y)
IM+2	=	IMAGE(X+1,Y-1)
IM+3	=	IMAGE(X,Y-1)
IM+4	=	IMAGE(X-1,Y-1)
IM+5	=	IMAGE(X-1,Y)
IM+6	=	IMAGE(X-1,Y+1)
IM+7	=	IMAGE(X,Y+1)
IM+8	=	IMAGE(X+1,Y+1)

IM+4	IM+3	IM+2
IM+5	IM+0	IM+1
IM+6	IM+7	IM+8

Figure 2.3 Window Re-mapping

Fortunately this overhead can be removed if a hardware re-mapping scheme such as that described above is in use, since it can detect these conditions at very high speed and return an edge value from a table (see chapter 5).

2.5.6 Register Use

Generally, it is quicker to access a value in one of the machine's registers than to retrieve it from memory. It is a good idea to leave as many variables in registers as possible. Obviously it is impossible to keep all the variables used by the program in registers so preference should be given to those used most frequently. A particular section may use a variable extensively and then not need it for a considerable time; thus the variable can be placed in a register for the duration of its use and returned to memory to free the register until it is required again.

Typical common uses of registers are for temporary storage in a long arithmetic expression or for loop counters which can be forgotten at the end of the loop.

2.5.7 Addressing Modes

Most computers provide a choice of addressing modes for different types of data access. It is often found that any one of a number of modes would be suitable in a given case. Care in the choice of the mode used can provide a worthwhile increase in speed.

For example, it is normal practice to use relative addressing when possible as this makes programs relocatable and requires no address calculations at load time. In practice the load time is a small fraction of the total run time and overheads here matter little. However, relative instructions require a run time address calculation which is performed a large number of times. Using absolute addresses would provide a considerable time saving.

Each computer has its own fastest and slowest addressing mode and reference must be made to the appropriate manual to determine the best mode for the machine in use.

2.5.8 Loop Counters

A great deal of image processing takes place in loops which must incorporate tests for completion. The test must occur within the loop so that any time saving in making the test is multiplied by the number of times the loop is executed.

It is normal to think of a loop running from one value, often 1, to another, larger, value. Less often the loop is required to run downwards from a large to a small value. A general loop such as that in PPL2 will allow both positive and negative increments; this introduces another overhead.

Consider the steps required in a loop of the form

FOR I FROM A BY B TO C DO ... OD

At the start of the loop it is necessary to set I to the initial value, A. At the end the increment, B, must be added to I and the result compared to the final value, C. If the increment is positive then I must be less than or equal to C to repeat the loop; if B is negative then I must be greater than or equal to C to repeat it.

The machine code produced by the PPL2 compiler for such a statement is as follows

```
                        ; FOR I   table entry only
     MOV     A,INIT     ; FROM A  save values so that they
     MOV     B,STEP     ; BY B     cannot be changed in the
     MOV     C,FINI     ; TO C     loop
                        ; DO
     MOV     #BGT 2,2$  ;         to test I > C at end
     TST     STEP       ;         is B < 0 ?
     BGE     1$
     MOV     #BLT 2,2$  ;         yes, test I < C at end
1$:  MOV     INIT,I     ;         give I its value
;    ...             the loop
                        ; OD
     ADD     STEP,INIT  ;         increment value
     CMP     INIT,FINI  ;         compare with final value
2$:   0                 ;         test condition from above
     JMP     1$         ;         go round again if OK
```

Determination of the terminating test has been moved outside the loop to save time.

Testing whether the loop is finished involves comparing the loop variable with the final value. Computers without the compare instruction would have to subtract the final value and test the result relative to zero. It is almost always faster to compare with zero as this is implemented in hardware within the processor.

In many cases it is possible to arrange that the loop runs from some value towards zero to reduce time. If it is only required to repeat a section of program a number of times the program below would be sufficient -

```
          MOV     TIMES,R0  ; use register for speed
     1$:
     ;    ...   the loop
          DEC     R0        ; decrement
          BNE     1$        ; repeat if not yet zero
```

A further improvement could be affected if the machine provides an instruction such as SOB "subtract one and branch if non-zero" (PDP11/34A) or DJNZ "decrement and jump if non-zero" (Z80). This results in the compact program -

```
          MOV     TIMES,R0
     1$:
     ;    ...   the loop
          SOB     R0,1$
```

It is quite permissible to run the counter from a negative value toward zero, although highly compact test instructions such as those above are rarely provided for such a purpose.

2.6 CONCLUSION

Choosing hardware and software for image processing is not easy and the relative merits of the different systems available are not easy to compare. Whilst a parallel processor (e.g. CLIP 4) seems to be the most suited to the task it is not obvious whether a string of shift register processors or a multiple micro-processor system would be more economic. Programming, it would seem, should be in a high-level language to allow the expression of essential concepts but the required speed may only be obtainable by the careful writing of low-level routines. The choice of system is very likely to affect or be affected by the method chosen to solve the problem, the ramifications of a particular choice may be extensive [22].

At present each task has to taken on its individual merits and the system chosen accordingly. When specialised hardware is more cheaply available and special image processing languages designed a more suitable system seems certain to emerge.

3. A LANGUAGE FOR IMAGE PROCESSING

3.1 INTRODUCTION

A great number of algorithms and techniques for image processing have been and continue to be developed. Unfortunately each researcher has used his own notation for setting down the procedures involved. It is thus difficult to compare published techniques to select the one most appropriate to the problem in hand. For a comparison to be made it is necessary to translate from the notation used in publication to that required by the researcher before he can assess the value of the technique. A great deal of time may be required to understand the original notation, translate it into the required form and then check that the new form does actually produce the same result as the original.

Comprehending the published algorithm may be easy or difficult depending on whether the user is familiar with the notation used and whether the notation is suitable for clearly describing the actions to be taken. A notation normally known to the scientist is that used to describe mathematical formulae. Many of the commonly used procedures in image processing can only be clumsily described in the standard mathematical notation. For this reason the technique being described may be obscured amongst notational detail. Some of the more straightforward techniques have been described in plain English, which cuts through the notational jungle to present the heart of the matter. English, unless very carefully written, can be ambiguous and open to a degree of personal misinterpretation and can only be used for the simplest of technical ideas.

Translation from the original to the final notation can be a source of error, particularly if the translator is not fully conversant with the notations used. In the course of transmitting algorithms and techniques from one researcher to another the notation used in development will often be translated into another form more suitable for publication and then translated again before use. There are two possible sources of error here which could result in the condemnation of a technique as unfit as a result of inaccurate implementation.

Ideally the notation used for development would be the same as that used for publication and implementation by another user. Errors are less likely to occur this way as checking against the original is straightforward and easy.

This chapter gives the reasoning behind the choice of a language and a description of the language designed. Chapter 4 gives details of its implementation on a mini-computer system.

3.2 A SUITABLE NOTATION

The need to find a notation suitable for communicating information between two or more people is not new. It was first apparent when early man recognized the advantages of group effort over individualism, for example when planning a hunting expedition. The notation that evolved is, of course, known as language.

Human language has developed over a very long period to the form we find it in today, in such a way that commonly used concepts have simple representations. When we breathe, for example, we require a substance called simply "air"; it is unnecessary to speak of "a gaseous mixture consisting of 78% Nitrogen, 21% Oxygen, ... etc.". Natural language has evolved to become an ideal method of communicating concepts found in our normal lives but is not well suited to some purposes.

Early mathematicians recognized the need for a more precise and more compact notation than than their natural language provided. Scholars developed a collection of symbols, and rules to manipulate them which were well suited to their needs. The mathematical language has also developed over a period of time to embrace the needs of today´s mathematicians.

Automatic manipulation of formulae by machine has long been an attractive idea. Many ingenious mechanisms have been devised for this purpose but it was not until the advent of the electronic computer that a truly useful machine could be said to have been built. The characteristics of the computer are such that it was found that the languages previously developed were inadequate for man-machine communication.

Initially computers were instructed, or programmed, to perform their tasks in a very basic language that corresponded closely with the machine´s electronic design. The construction of different models of computer led to a variety of languages and resulted in the need to translate one machine´s programs into the language of another. It was difficult to use on one computer techniques developed on another and errors occurred in translation. Much the same situation is seen today in image processing machines.

Scientists recognized the need for a standard language in which algorithms and techniques could be clearly written and transferred from machine to machine. The languages devised were written in such a way as to be close enough to natural language to be readily understood by people, yet precise enough to be unambiguous to the machine. A number of languages have been devised, each with particular short-hand notations for commonly used elements in the class of problem the language was intended to describe. One of the earliest languages devised was FORTRAN, its name, a contraction of "FORmula TRANslation", reflecting the intention of using it to describe mathematical formulae. The idea of a standard descriptive language was found to be so powerful that a new all embracing language for problem description and solution was devised some fourteen years ago, viz ALGOL68.

Languages such as FORTRAN and ALGOL68 share the common feature of generality: the language is not peculiar to one computer only but programs may be transported, without change, from one machine to another. The absence of the need to laboriously translate algorithms has led to a much easier exchange of problem solving ability to the profit of all concerned.

Unfortunately none of the common computer languages has all the features required to compactly describe image processing algorithms. Although some languages have been designed for particular machines [23,24,25] none, at the time of commencing this project, appears particularly well suited to the system described in chapter 2.

3.3 PPL2

The need for a language better suited to image processing than those commonly available was recognized several years ago and led to the development of PPL [5]. PPL, a Picture Processing Language, contains many features which make it particularly suited to the description of image processing techniques. After using this language for a while it became apparent that some features such as the conditional statements were too compact for easy reading and that some facilities such as a simple loop structure were lacking. In order to correct these shortcomings a new language was developed, based on the original PPL, and known as PPL2.

PPL2 is designed to be a descriptive language for image processing. It contains features that are of particular use when dealing with operations on one or more pictures. The language is "high level" which means that it describes operations in a machine independent manner. The programmer need not know the details of the particular computer or image storage hardware that he is using. Provision exists for specifying that an algorithm be performed sequentially or in parallel but the user need not know whether the machine is actually doing what it is told or merely simulating it.

Some of the constructs used in PPL2 are based on those found in ALGOL68, in particular the version known as ALGOL68-R and described in [26]. It is a block structured language which has been found to be very convenient in use.

Variables are given no particular type; they are simply used as words and may be treated as integers or logical variables as required.

3.3.1 Keywords

Keywords used by constructs are not marked in any special way but are simply used as written. For this reason the programmer must not use them for variable names, as they are reserved words. In practice this is not a disadvantage but makes the program more readable by people. Since there is only a handful of keywords to be avoided these are quickly learnt.

An interesting, though controversial and fallible, measure of the sophistication of a language is the number of reserved words it contains (a very small number may indicate a language with few useful facilities while a very large number may indicate a needlessly complex structure). There are 35 reserved words for PPL2, which may be compared with 30 for ALGOL68, 31 for PASCAL, 64 for ADA and 38 for FORTRAN.

3.3.2 Variables

A variable name may be between one and six characters long. The first character must be a letter and the remainder letters or digits. A variable name may not be the same as a keyword.

3.3.3 Comments

If a program is to be understood clearly it is valuable to be able to add comments which do not form part of the program. The comment may, for example, be a plain English description of the program. Comments in PPL2 may be placed anywhere in the program by enclosing them within curly brackets.

e.g. { this is a comment in PPL2 }

Here, as elsewhere, the choice of symbols used to indicate a particular function within PPL2 is dictated partly by convenience and partly by the need to use characters available on a standard ASCII keyboard.

3.3.4 Images

Operations on images are generally such that the pixel value at each point on the new image is some function of pixels on the same or other images. Algorithms utilise a point on an image and its near neighbours together with corresponding areas on other images to form a pixel value for the new image. The point and its near neighbours are known as a window on the image. How near the near neighbours are depends on the algorithm in use. The bulk of the existing work in image processing uses only the adjacent cells giving a window of 3 cells by 3 cells. It might be that "less near" neighbours are required; this ought, at least, not to be precluded.

Images are normally stored as an array of pixels and access to a particular point made by the use of indices. If a neighbour is required its position relative to the centre point must be calculated and explicitly written as an index.

For example, if the centre point of image P is P(x,y) the neighbour to its left is P(x-1,y) and that above it P(x,y-1).

This method of specification becomes extremely tedious if the algorithm in use is any more than very short. Since it is inconvenient to specify the indices each time, PPL2 assigns a name to the centre point and to each of the neighbours so that the program becomes much easier to follow.

The centre point of image P is called P0 and its neighbours P1, P2, P3, ..., Pn; by the following convention :

```
        P4  P3  P2
        P5  P0  P1  etc., continuing in an anti-clockwise spiral.
        P6  P7  P8  P9
```

Thus the centre point is always P0 and its left-hand neighbour always P5, regardless of where P0 is on the image. A complete list of the point numbers in a 15 x 15 window is given in Figure 5.9.

Up to 26 images may be specified, their centre points being A0, B0, C0, ..., Z0 with corresponding neighbours following the convention above.

3.3.5 Assignment

The assignment indicator for giving variables a value follows the ALGOL convention and uses the symbol ":=".

e.g. THREE:=3 assigns the value 3 to the variable THREE.

Multiple assignments may be performed, the assignment proceeding from right to left.

e.g. A::=B::=C::=10 assigns the value 10 to all three variables "A", "B" and "C".

3.3.6 Operators

To perform anything useful operators must be available to generate new values as a function of one or more given values.

The normal range of arithmetic operators is provided:

+	Addition
-	Subtraction
*	Multiplication
/	Division

And the logical operations so that "bit level" operations may be performed :

&	And
!	Or
$	Exclusive or
~	Not (ones complement)

The symbols chosen are those found on a standard ASCII keyboard looking similar to symbols normally used to represent the functions.

With the normal comparison operators:

>	Greater than
<	Less than
=	Equal to
#	Not equal (c.f. crossed through =, normally used in mathematics)
>=	Greater than or equal to
<=	Less than or equal to

The comparison operators return a True or False value. True is defined as a word full of one bits (-1) and False a word full of zero bits (0).

A few other operations have been found to occur quite frequently in image processing: these are fairly lengthy to write out normally. Operators which take much space to set out often obscure the effect they are intended to have and therefore simpler notation for them has been used.

Since these operators all ask a question, i.e. could be written with an "IF" statement, the symbols chosen for them begin with a question mark. By doing this a second symbol can be appropriately chosen rather than looking for an otherwise unused character on the keyboard. This two character representation is much shorter than the notation found in other languages which prevents the program detail being obscured by it.

The operators found most useful are:

```
?> Whichever is the greater of two operands.
    e.g. A?>B will return the greater of A and B.
         i.e. IF A>B THEN A ELSE B FI
         A?>B?>C?>D returns the largest value.

?< Whichever is the smaller of the two operands.
    c.f. ?>

?+ Absolute value of the operand.
    i.e. ?+A is equivalent to IF A>0 THEN A ELSE -A FI

?¦ In range. This checks that the operand has a value between
    black and white, if not it returns the appropriate
    limiting value.
   i.e. ?¦A is equivalent to  IF A<0   THEN 0
                             ELSF A>255 THEN 255
                                        ELSE A    FI
```

A complete list of operators and their priorities is given in the User Guide, appendix C.

Subexpressions may be grouped within round brackets to change the order of evaluation from that dictated by the priorities of the operators used.

3.3.7 Constants

A constant value in an expression may be decimal, octal or the ASCII value of a character.

A decimal constant is a string of digits, each one in the range 0 to 9:

e.g. 9101

An octal constant is a string of digits in the range 0 to 7 preceeded by the character "^":

e.g. ^240 (=160)

A character constant is any character between two prime symbols, the 7 bit ASCII value of the character is used:

e.g. ´A´ (=^101 =65)

3.3.8 Image Operations

To process the whole image the operation specified for the window must be performed with the window occupying each of its possible positions. Parallel processing involves applying the window operator at all points on the image at the same time, whilst sequential processing involves scanning the window in a particular direction. A characteristic of parallel processing is that data for the window operation has not been changed by a previous application of the operation. The effect of parallel processing can be achieved even on a serial machine if the result of the operation is placed into a different picture space and does not overwrite the original information.

Conventional languages would require that the picture scan be explicitly written with a pair of nested loops, one for the X direction and one for Y.

In FORTRAN, for example:

```
       DO 200 J=1,128
       Y=J-1
       DO 100 I=1,128
       X=I-1
C   Window operation applied 16 384 times in a
C forward raster scan.
100    CONTINUE
200    CONTINUE
```

Or in ALGOL68:

```
'FOR' Y 'FROM' 0 'TO' 127 'DO'
'FOR' X 'FROM' 0 'TO' 127 'DO'
  'COMMENT'  Window operation applied 16 384 times
          in a forward raster scan.
  'COMMENT'
'OD'
'OD';
```

This long winded loop specification can obscure the real processing going on by losing it amongst repetitive control statements. Indeed, many of the window operations are very simple, if not trivial (for example adding two pictures together) and the control statements are many times their length.

PPL2 provides a succinct method of expressing the desire to perform the window operation over the whole picture. The window operation is enclosed within pairs of square brackets:

```
[[  { Window operation }  ]]
```

This clearly shows the detail of the operation without cluttering the page with clumsy control structures.

For example, to generate picture A as the average of the two pictures B and C in FORTRAN we would write:

```
      DO 200 Y=1,128
      DO 100 X=1,128
      A(X,Y)=(B(X,Y)+C(X,Y))/2
100   CONTINUE
200   CONTINUE
```

whilst in PPL2 this would appear as:

```
[[ A0:=(B0+C0)/2 ]]
```

Notice that in FORTRAN and ALGOL68 etc. the size of the picture must be known to define limits for the loops, whilst in PPL2 the operation is simply specified over the whole picture, whatever its size may be. Thus PPL2 algorithms may be applied without change to any size of picture.

The examples given above specify parallel processing of the picture, if sequential processing is required it becomes necessary to specify the direction, forward or reverse scan. A simple extension to the above notation achieves this:

```
[[+  { Window operation, forward scan }  +]]
[[-  { Window operation, reverse scan }  -]]
```

The position of the centre of the window is available, if required, by reference to the variables 'X' for the x co-ordinate and 'Y' for the y co-ordinate.

3.3.9 Edges

When the centre point of the window is on or near the edge of the picture some of the neighbours may be outside the defined picture. To perform meaningful processing these neighbours must acquire a sensible value, so they are given what is known as the edge value. A picture is considered to be completely surrounded by an infinite border of points all having the edge value. Each picture has its own edge value which is set by assigning a value to the appropriate variable of EDGEA, EDGEB, ..., EDGEZ.

For example, to surround picture A by a black border we would write, simply:

```
EDGEA:=0
```

3.3.10 Indexing

Some programs require the acquisition of tables of values, to produce a histogram, for example; this is conveniently done in an array. An array in PPL2 is given a name following the convention given in section 3.3.2, and its elements referred to by indexing. Indices are written in square brackets after the variable name.

For example to access the third element of array "FRED", whose index values start at zero, we would write:

```
FRED[2]
```

An index value of zero may be omitted.

i.e. FRED[0] refers to the same location as FRED.

3.3.11 Conditionals

The conditional statement directs the flow of a program depending on the states of variables within it. This statement is characterised in English and computer languages by the word "IF". It is commonly known as an IF-statement.

The form of conditional statement used for PPL2 is the same as that used in ALGOL68 ([2]). This is a particularly general purpose and clear form.

The simplest version is simply:

```
IF A THEN B FI
```

"A" may be a simple variable or a complex statement returning a value , if the value of "A" is logically True then the expression "B" is performed.

For example, scan the picture P and change any pixels with value less than 100 to have value 0 (this is a form of thresholding):

```
[[ IF PO < 100 THEN PO:=0 FI ]]
```

It is often useful to be able to specify an alternative course of action if "A" returns a value logically False, for this the form including an "ELSE" clause is used:

```
IF A THEN B ELSE C FI
```

Only one of "B" or "C" is performed depending on the outcome of "A".

The inclusion of "FI" as a delimiter removes ambiguity that may occur when statements are nested.

For example, without using Fl´s the statement below is ambiguous:

```
IF A THEN IF B THEN C ELSE D
```

It is not clear which IF the ELSE belongs to. Using FI´s the two possible forms would be:

```
IF A THEN  IF B THEN C ELSE D FI  FI

IF A THEN  IF B THEN C FI  ELSE D FI ,
```

clarifying the situation.

It may be required to test a number of conditions such that a statement like the following may occur:

```
IF A < 10 THEN B ELSE
 IF A > 20 THEN C ELSE
  IF A = 15 THEN D ELSE
   E
  FI
 FI
FI
```

In this type of statement the pair of words "ELSE IF" may be reduced to the word "ELSF" and also the corresponding "FI" be removed to give:

```
  IF A < 10 THEN B
ELSF A > 20 THEN C
ELSF A = 15 THEN D
             ELSE E
  FI
```

An IF-statement with both a THEN and an ELSE part returns a value which is the value of the part performed; the whole "IF ..... FI" expression has a value and may be used in, for example:

```
A:=IF B < 10 THEN 4 ELSE 5 FI
```

in which A takes the value 4 if B is less than 10 and 5 if it is not.

3.3.12 Loops

It is often useful to be able to perform a section of program a number of times, the precise number of times may not be known when the program is written and so a mechanism for providing this function under program control is useful. The mechanism provided is one whereby the program loops back from the end of a section to the beginning, repeating the section a number of times or until a specified condition exists.

This facility is provided in FORTRAN by the DO-loop and in ALGOL68 by the FOR-loop. The ALGOL68 ([26]) version has been adopted in PPL2 as it provides a highly versatile method of program section repetition.

The complete form of the loop is:

FOR A FROM B BY C TO D WHILE E DO {program section} OD

In this case A is a variable which is initially given the value B; providing E returns a logical True the program section is executed. After execution the value C is added to A and the result compared to D; If C is positive and A is greater than D or if C is negative and A is less than D then the loop terminates and execution continues from the statement following the OD. If the new value of A is acceptable then condition E is checked, and if still True the program section is repeated. If E is found to be False the loop terminates as above.

In the example above A must be a variable but B, C, D and E can be complex statements provided they return suitable values. B, C and D will only be calculated once, on the first time through the loop so that changing them within the program section will not affect the number of times the loop is performed. E is executed each time round.

In most cases the full form given above is not required; at such times it may be abbreviated. The only parts of the statement which must occur at all times are the DO and OD, all other parts may be omitted when not required. Those parts which do appear must do so in the order FOR FROM BY TO WHILE DO OD. Those parts omitted are given default values:

Part	Default	Comments
FOR	-none-	No variable is given values
FROM	1	
BY	1	
TO	infinity	There is no final value
WHILE	-none-	No condition is tested

Examples:

```
TO 20 DO {program section} OD
      The program section is performed 20 times.

WHILE Z < 6 DO {program section} OD
      The program section is repeated until Z >= 6.

DO {program section} OD
      The program section is repeated ad infinitum!.
```

A statement of the form containing a WHILE clause may execute the program section a minimum of no times if the conditional is False on the first time through. The form having only a FOR part is executed a minimum of once as checking against the final value does not occur on the first pass.

Statements such as REPEAT ... UNTIL found in, for example, PASCAL [27] may be implemented by remembering that the WHILE clause may contain a complex statement:

REPEAT A UNTIL B can be written in PPL2 as:

WHILE A; B DO SKIP OD so that A is performed before testing B; the dummy statement SKIP is inserted between DO and OD for syntactic accuracy, it does nothing.

3.3.13 Statement Separators

Statements in PPL2 are separated by semicolons, as in ALGOL68 and not by newlines as in FORTRAN. In this way statements may be split over more than one line or several statements may occur on the same line; the user is left to set out his program in the most readable way.

In common with ALGOL68 dummy statements may not appear; consecutive semi-colons are forbidden, as are semi-colons before THEN, ELSE, FI, OD etc.. Semi-colons are thus statement separators and not terminators. A series of statements separated by semi-colons may be considered as one longer statement.

3.3.14 Sub-Programs

Common sequences of expressions may be required to be executed at several places in a program. To avoid the need to retype these sequences each time they are required they may be grouped together into sub-programs and called when required.

Each sub-program is given a name by preceeding the sequence of expressions in the sub-program by the name and a colon. The end of the sequence is indicated by the word RETURN appearing.

For example:

```
SUBONE: {sequence of expressions in the sub-program} RETURN;
```

At the point in the program at which the sub-program is to be performed its name, preceeded by the "@" sign, should be written.

i.e. as:

```
@SUBONE
```

For example:

```
P: @A; @B; @A RETURN;
A: {sub-program A} RETURN;
B: {sub-program B} RETURN;
```

is equivalent to:

```
P: {sub-program A};{sub-program B};{sub-program A} RETURN;
```

3.3.15 Input and Output

Communication between the program and the user is necessary for the program to give information about what it is doing or has discovered and for the user to provide input to it.

The type of input and output provided by FORTRAN has been adopted for PPL2 as it is perfectly adequate for the job and is well established. This provides both free format I/O and comprehensive formatting when required for clear tabulation.

Free format input is of the form:

```
READ A,B,C,.....;
```

Where A,B,C,..... is a list of variables whose values are to be taken from the terminal. The list must contain at least one item.

Free format output is of the form:

```
WRITE A,B,C,.....;
```

Again A,B,C,..... is a list of items to be printed on the terminal, there must be at least one item. These items may be constants, variables or expressions.

When formatted input or output is required the READ or WRITE must be followed by a normal FORTRAN style format enclosed within double quote symbols.

For example:

```
READ "(2I3)",A,B;    {to read A and B in I3 format }
WRITE "(10X,'FRED=',I6)",FRED;
WRITE "(' Message to the user')";
```

A read statement must have at least one variable to be assigned a value, but the formatted write need not have any.

3.3.16 Spaces

In general PPL2 disregards any spaces it finds in the program; however, there are some places where they must and some where they must not occur.

Spaces MUST occur:

To separate keywords from variable names.

Spaces MUST NOT occur:

Within a keyword or variable name,
Within a decimal or octal constant,
Within the primes of an ASCII constant, unless it
  is the character required,
Between the ^ and digits of an octal number,
Between the @ symbol and sub-program name,
Between the characters of a multi-character symbol
  (e.g. [[+ ?> <= etc.).

3.3.17 Special Hardware

It is quite likely that the image processing system will have some special hardware that the program will want to access. Given the computers´ address of this hardware PPL2 is easily able to access it.

The indexing mechanism performs an address calculation; in the case of A[B] the value of B is added to the address of A to form the new address. If the address of A is known it is a simple matter to access any particular memory address. A special variable, MEMORY which has address 0 is provided for this purpose.

For example to set the variable MEM to contain the value at address 1000 octal we would write:

```
MEM:=MEMORY[^1000]
```

Similarly values can be written to memory.

4. AN IMPLEMENTATION OF PPL2 ON A PDP1134/A

4.1 INTRODUCTION

Algorithms must be translated from the notation used for their description into a notation that can be understood by the computer processing the image. A great deal of time may be spent in the translation and checking of algorithms composing the user´s library of routines. Adding new routines to the library can be time consuming, as can making changes to existing routines. This will still apply even for small changes, such as changing a threshold value or test condition.

Notations that can be used both to describe image processing algorithms clearly and to program a computer would save time that could be better spent studying the technique described. So that PPL2 would fall into this category of notations a program was written that would interpret the language to perform image processing algorithms directly.

4.2 OVERVIEW

Source code statements may be either interpreted to directly produce actions in the computer or used to compile machine code which is later run, producing the required actions. The major blocks of actions required are compared in figure 4.1.

From this figure it can be seen that the compiler does more work before producing any action. However, if it is desired to run the program more than once there is no need to look at the source code repeatedly. The interpreter is well suited to programs that run once only, for example setting up initial values. A feature of image processing is that an action to be performed on a window operation has

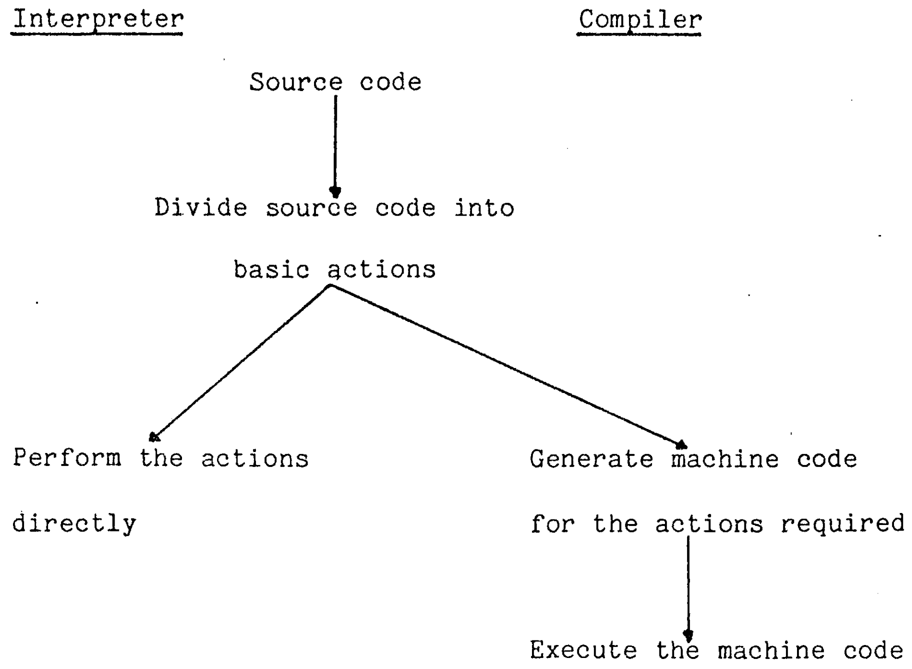

Interpreter
Compiler
Source code
Divide source code into basic actions
Perform the actions directly
Generate machine code for the actions required
Execute the machine code

Figure 4.1 A Comparison of an Interpreter and a Compiler

to be repeated once for each pixel in the image; this would be horrendously slow with an interpreter though a compiler is ideally suited to this task.

Complete algorithms will contain a mixture of control statements and image operations which makes the choice between compilation and interpretation less than obvious. The case is further confused in a research environment (where algorithms are being developed) by the need to make many minor changes, for example to threshold values; such needs are ideally met by an interpreter.

Since a great deal of the work to be done by an interpreter or compiler is common to both (see fig 4.1) it was considered practical to implement both within the same program. By this means the control structures could be interpreted and image operations compiled. The image operations have to be compiled each time they are called upon to be executed but are then run some 16 000 times over the whole image. Because only one image operation is compiled at a time, a small area for compiled code is sufficient; 1000 (decimal) words are at present allocated for this purpose although most basic image processing operations require less than 500 (decimal) words. The interpret mode allows variables to be quickly and easily accessible for modification or verification.

Interaction between the user and computer has been further enhanced by the provision of a number of extra facilities, including:

(a) Text editing ´on-screen´ for the rapid modification of programs. Programs may be stored on or retrieved from disc.

(b) Storage and retrieval of images on disc to allow standard inputs for comparing algorithms.

(c) Detailed examination of an image by displaying a movable window as numeric values on a visual display unit.

(d) Copying the image to the printer for detailed examination away from the computer.

These facilities are fully described in the user manual, Appendix C.

4.3 THE FUNCTIONS OF A COMPILER

Source code is presented as a string of characters which must be interpreted as a program, and the specified actions performed. The processing may conveniently be divided into three fairly independent sections: lexical analysis, syntactic analysis and code generation (when compiling) or direct execution (when interpreting).

4.3.1 Lexical Analysis

Lexical analysis divides the source text into groups of characters that each have a particular syntactic meaning; for example, groups of digits are constants, groups of letters are variable names or keywords, while other symbols may be brackets or operators. Once a syntactic unit has been identified the source characters are no longer required and the unit in passed through the rest of the compiler/interpreter as a compact internal representation.

Information at this stage may be used to build tables for use by the lexical analyser or other stages in the compiler/interpreter: such tables include, for example, variable names and program labels.

4.3.2 Syntactic Analysis

Syntactic analysis determines the meaning of groups of syntactic units, for example recognizing specific groups of variables and operators as arithmetic statements. This phase further breaks down its data into very simple operations which can be handled by the final phase of compilation.

4.3.3 Code Generation and Direct Execution

Commands generated by the syntax analyser are simple enough to be used either to generate machine code (in a compiler) or to cause the specified operations to be performed directly (in an interpreter). Use is made of tables generated earlier in the compilation process.

4.4 THE PPL2 APPROACH

Bottom up parsing is employed by PPL2, i.e. the input characters are interpreted as small program elements which are combined to form larger program structures. Lexical analysis of PPL2 source code has to identify only four syntactic groups to pass to the syntax analyser:

Group 1 - Keywords are recognised and converted, via a look-up table, to one or more operators or constants.

Group 2 - Constants, such as strings of characters or digits, are converted into single values.

Group 3 - Variable names are looked up in the symbol table; if they exist their address is passed on: if not, the name is entered into the table, an address assigned and passed on.

Group 4 - Special symbols such as brackets and operators. Some operators are represented by a group of symbols because of the restricted character set available. Certain symbols need to be converted, via a look-up table into strings of operators and constants.

Superfluous information (comments, spaces, indicators such as ^) is removed at this stage. Special conditions, such as the existence of unary operators, are trapped and replaced by a suitable sequence of operators and operands. Certain errors such as consecutive operators or operands are detected in this phase, an error message issued and compilation aborted.

Syntax analysis is made relatively easy by forcing the output of the lexical stage to obey the syntax of a mathematical expression. This allows the use of a very simple compiling technique, the most basic of the class of operator precedence compilers [28]. Section 4.5 describes the processing of a mathematical expression and section 4.6 illustrates the advantage of coercing other elements of the language to behave in the same way.

Ultimately the functions described in the source language must be used either to cause actions to be taken (interpreted) or to cause machine code to be produced (compiled). By this final stage in the process the source code has been divided into many very simple operations in the form of an operator taking two operands and producing a result. When it is best to do so (for operations not involving an image scan) these operations are directly performed by the interpreter. When dealing with image operations machine code is produced and stored for subsequent execution. It is quite straightforward to convert the simple operations demanded by the syntax analyser into basic machine operations. Section 4.7 discusses some other techniques required when generating machine code.

4.5 MATHEMATICAL EXPRESSIONS

Mathematical expressions may vary in complexity from simply a single value to something requiring many lines to write. All expressions are characterised by having one or more items having values, called operands, and zero or more items that combine them in defined ways, called operators.

For example :

1 is simply an operand with value one,

1+2 has two operands "1" and "2" combined by the operator "+" which is defined as an addition operation; the result of this expression is the simple operand "3".

Most operators perform some function with two operands to yield a single result, they are called binary operators; some take only a single operand and are called unary operators (e.g. "-" in -3 is a unary operator).

Operators with their operands yield, after evaluation, a single result which may then become an operand for another operator. By this means a long, complex expression will eventually yield a single result.

Rules must exist to specify the order of evaluation of operators in a complex expression or the result will be unpredictable. Such a set of rules has been evolved by mathematicians and is well known. (It is taught to everyone at school!) This set of rules has been formalised for computer use by assigning a priority ordering to the operators being used. The highest priority operators are performed first, followed by those of a lower priority. Thus, for example, all multiplications and divisions are performed before additions and subtractions. When two operators of the same priority occur in an expression the rule for most operators is that they are performed leftmost first. Some, however, are performed rightmost first - notably exponentiation and assignment (:=).

These rules can be illustrated by the following example :

```
Expression : 4+3*7*10/15-9
  evaluate     3*7     to give 21 and the expression becomes
             4+21*10/15-9
  evaluate     21*10   to give 210 and it becomes
             4+210/15-9
  evaluate     210/15  to give 14 and it becomes
             4+14-9
  evaluate   4+14      to give 18 and it becomes
             18-9  which is evaluated to give the final result of
             9.
```

This order of evaluation may not always be convenient, so a technique for changing the order is required. Subexpressions are enclosed within brackets to indicate that they must be evaluated first. The subexpressions are evaluated according to the rules above although brackets may again be used around sub-subexpressions.

For example :

Expression : 3*(4+5)

evaluate 4+5 to give 9 and the expression becomes

3*9 which is evaluated to give the result 27.

Operators need not return results which are a linear function of the operands; they may, for example, return one of two values only, i.e. a binary result. Such results are returned by the comparison operators, the values returned being one of the logical values TRUE or FALSE. These operators are found within conditional statements and may be treated in the same way as any other operator.

PPL2 has also to allow bit-level manipulation and is therefore provided with a range of logical operators.

A list of these mathematical operators and their priorities is given in figure 4.2.

OP	Priority	Result of A OP B
*	11	A multiplied by B
/	11	B divided by A
+	10	A added to B
-	10	B subtracted from A
=	8	A equal to B (logical value)
#	8	A not equal to B (logical value)
>	8	A greater than B (logical value)
<	8	A less than B (logical value)
>=	8	A greater than or equal to B (logical value)
<=	8	A less than or equal to B (logical value)
&	7	A AND B
!	6	A OR B
$	5	A Exclusive OR B

Figure 4.2 Mathematical Operators and Their Priorities

4.5.1 Compiling Mathematical Expressions

As was illustrated in the last section, expressions are not always calculated from left to right as they are written down. A technique for re-ordering the expression has to be employed to produce a simple sequence of instructions that may be performed on a computer. The method employed in PPL2 is described in [29] and involves scanning the input stream, storing items found on one of two stacks, until an operator may be executed; in this method the result is placed on one of the stacks for future use.

The sequence of events is shown in the flowchart in figure 4.3 and may be illustrated with a simple example :

Expression : 1+2*3- ... etc.

Step	Action	Comment
1:	Get "1"	The first item in the expression.
2:	Push onto stack B	We can do nothing with it for now, store it on the stack.
3:	Get "+"	The next item is an operator.
4:	Push onto stack A	The next operator may have higher priority so store this one until we know.
5:	Get "2"	As in step 1.
6:	Push onto stack B	As in step 2.
7:	Get "*"	Operator.
8:	Push onto A	As in step 4.
9:	Get "3"	As in step 1.
10:	Push onto B	As in step 2.
11:	Get "-"	Operator.

12:	Remove "3" and "2" from stack B;
	Remove "*" from A;
	Calculate 2*3;
	Push the result (6) onto stack B.
		Now we can do the "*" as it has higher priority than "-".

13:	Remove "6" and "1" from stack B;
	Remove "+" from A;
	Calculate 1+6;
	Push the result (7) onto stack B.
		Since "+" and "-" are the same priority we proceed from left to right.

14:	Push "-" onto A	As in step 4.
15:	etc. etc. etc.	

For clarity this example has actually performed the calculations, whereas in practice a compiler would generate code to implement them.

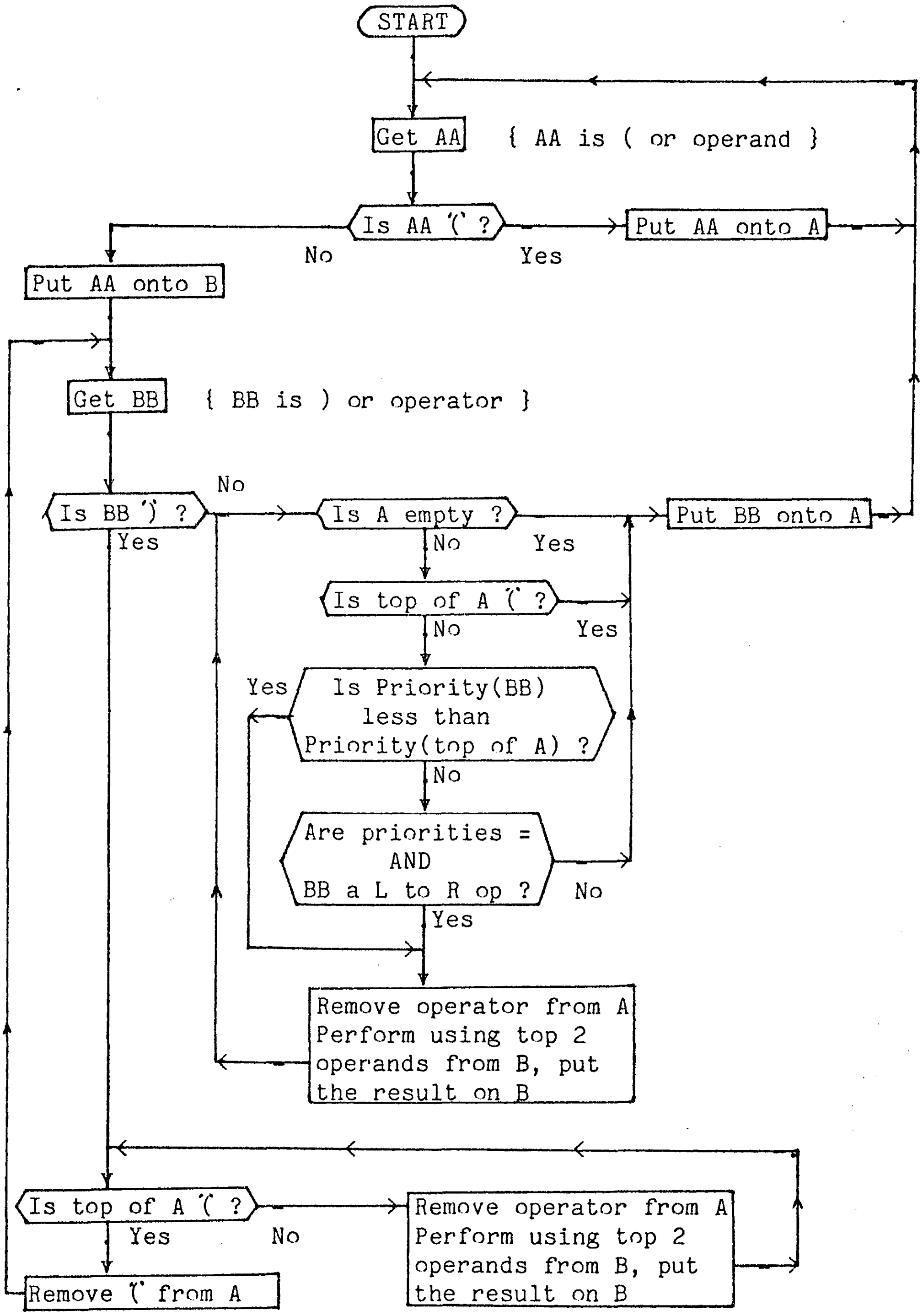

START
Get AA
{ AA is (or operand }
Is AA '(' ?
No
Yes
Put AA onto A
Put AA onto B
Get BB
{ BB is) or operator }
Is BB ')' ?
No
Yes
Is A empty ?
No
Yes
Put BB onto A
Is top of A '(' ?
No
Yes
Is Priority(BB) less than Priority(top of A) ?
Yes
No
Are priorities = AND BB a L to R op ?
No
Yes
Remove operator from A Perform using top 2 operands from B, put the result on B
Is top of A '(' ?
Yes
No
Remove '(' from A
Remove operator from A Perform using top 2 operands from B, put the result on B

Figure 4.3 Flowchart of Mathematical Expression Interpreter

For example, for "Calculate 2*3" read "Generate the code to calculate 2*3". Also, where constants appear above it is more likely that variable names would appear; if constants appear there is no need to generate code, PPL2 detects such occurrences so as not to produce redundant code.

The scheme discussed above can only deal with binary operators, i.e. operators having two arguments or operands. Most mathematical operators are binary but a few are unary, requiring only one operand; for example "-" as in A*-B. Since only a very few such operators occur (four in PPL2) it is possible to detect them and make some special provision for dealing with them. The simplest thing to do is simply to introduce a dummy operand in front of the operator so that it can be processed as though it were binary. When the time comes to generate code the dummy operand is simply ignored.

4.6 OTHER ELEMENTS OF THE LANGUAGE

Programs usually consist of more than just a mathematical expression whose value is to be calculated. The result of an expression is usually stored in a location known by name (a variable) and re-used later in some other expression. In addition the flow of the program is directed by control structures.

Symbols used to specify value assignment or to separate expressions may be interpreted directly as operators whilst control structures, by suitable interpretation of keywords, may also be forced to fit this scheme.

4.6.1 Value Assignment

The assignment operator, ":=", is a straightforward binary operator where the value of the right-hand operand is placed in the location specified by the left-hand operand. the left-hand operand must indicate a place to put the value, i.e. it must be a variable name; an expression such as 6:=A is meaningless.

The value resulting from this operator is simply the value of the right-hand operand.

When a sequence of assignment operators occurs in an expression they must be evaluated from the rightmost first so that an expression such as A:=B:=C:=6 can have meaning.

This operator has a lower priority than any other mathematical operator to ensure that it is performed last.

4.6.2 Statement Separation

Statements are separated by a semi-colon (";") which indicates that the expression to its left is complete and should be fully evaluated before the expression to its right is started. It is simply a binary operator with a very low priority which guarantees the required ordering. The result returned is that of its right-hand operand so that even a long series of expressions produces only a single value at the end (clearly, it must have a lower priority than the assignment operator to do this).

4.6.3 Item Separation

The comma (",") is used to separate items in a list; for PPL2 such lists are required only for input or output. This is also a binary operator, its priority lies between those of the statement separator and value assignment operators. When performed this operator constructs a list with the two operands which is returned as the result.

4.6.4 Control Structures

The order in which operators are performed is determined solely by their priorities and the use of brackets. This feature can be used to advantage when considering how to deal with control structures.

In many languages the processing in the compiler revolves around the keywords and control structures with occasional lapses into expression interpretation. By converting the keywords into suitable sequences of operators, operands and brackets the whole of a program in a block structured language such as ALGOL68 or PPL2 can be interpreted as a single (long) mathematical expression. Using this method the compiler does not need to keep a long list of "exceptions to the rules" and may thus be vastly simplified.

Keywords, when they are encountered, often signify an action that must take place at that point in the program, after finishing all operations before them and prior to starting any operations that come after them. This can be achieved by replacing the keyword with a sequence such as:

) ´A´ o ´B´ (

Where ´A´ is an operator with high priority,

o is a dummy operand for syntactic correctness

and ´B´ is an operator with the same priority as ´A´.

The close bracket ensures completion of preceeding operators before ´A´. When ´B´ is encountered ´A´ will be performed by the "left to right" rule for operators with the same priorities. Execution of ´A´ causes the required action for the keyword it replaces. Operator

´B´ is a dummy causing no action to take place. There is no need to generate any code if this is not required (when building a table). The final open bracket marks the beginning of the function to the right of the keyword. Some keywords employ variants on this scheme.

Rather than discuss this further in an abstract manner two examples from the implementation of PPL2 are given in figures 4.4 and 4.5 which illustrate the technique. Operators will be indicated by enclosing a symbol or symbols in quotes, operands by a numeric value or string of one or more symbols. Round brackets have their usual meaning.

4.6.5 Image Scan

Execution of an operation over the whole image requires a loop to be set up around the operation. This is indicated in PPL2 by the double square bracket notation.

Double open square brackets set up the initial loop values and notes addresses to loop back to at the end. Any program between the open and close double brackets then has to be completely compiled and the close brackets indicate the end of the loop, jumping back to the addresses noted earlier.

Source text expansion and subsequent evaluation order is shown in fig 4.4. Operator ´FORWARD´ is given a high priority to ensure its early execution to start the loop; the image processing is contained in brackets to ensure its complete compilation before the ´FEND´ operator is executed to end the loop. The dummy operand 0 and operator ´0´ ensure that ´FORWARD´ is not deferred until after the bracketed section.

```
[[ P0:=Q0 ]]     -->     ( 0 ´FORWARD´ 0 ´0´ ( P0 := Q0 ) ´FEND´ 0 )
```

SEQ	OPERAND	OPERATOR	OPERANDS -->		RESULT	COMMENTS
1	0	´FORWARD´	0	-->	0	Code, start of loop
2	P0	:=	Q0	-->	Q0	Code, P0:=Q0
3	0	´0´	Q0	-->	Q0	do nothing
4	Q0	´FEND´	0	-->	0	Code, end of loop

Figure 4.4 Image Scan Expansion and Result

4.6.6 Conditional Statements

Similar reasoning to that above is used to implement the conditional statement. A simple example with its expansion and subsequent order of evaluation is given in figure 4.5.

The first operator ´IF´ is required to store details of any outer conditional statements that may be in progress; it generates no run time code. A dummy operator ´0´ is included for correct sequencing and then the bracketed conditional expression is compiled. The ´THEN´ operator indicates that the logical value generated must be tested to determine whether the next section of program is to be run. The conditionally run program section is then compiled. Lastly the ´FI´ operator is executed to fill in the jump address of the ´THEN´ and to return to any outer conditionals in progress.

```
IF A=B THEN C:=5 FI
  --> ( 0 ´IF´ 0 ´0´ ( A = B ) ´THEN´ 0 ´0´ ( C := 5 ) ´FI´ 0 )
```

	OPERAND	OPERATOR	OPERAND	-->	RESULT	COMMENTS
1:	0	´IF´	0	-->	0	stack details of any outer IF´s
2:	A	=	B	-->	A=B	Code, compare A and B
3:	0	´0´	A=B	-->	A=B	do nothing
4:	A=B	´THEN´	0	-->	0	Code, test operand, possible jump
5:	C	:=	5	-->	5	Code, C:=5
6:	0	´0´	5	-->	5	do nothing
7:	5	´FI´	0	-->	0	Fill in jump address from THEN; Unstack details of outer IF

Figure 4.5 "IF" Statement Expansion and Result

4.6.7 Other Statements

A few other keywords and structures are also converted to strings of special operators and operands: these are fully listed in figure 4.6 and the operators detailed in figure 4.7. Those operators in figure 4.6 which may be unary or binary depending on where they occur have two entries, the first for binary use and the second for unary.

Some of the operators may occur only in the interpret mode or only in the compile mode; these return an error if they are found in the wrong mode.

PPL2 compiles machine code for image operations but interprets all other statements; the switch to compile mode is indicated by the operators associated with "[[" and the switch back to interpret mode by those associated with "]]"; these latter also causing execution of the image scan code generated.

4.7 CONCLUDING REMARKS

The complete system described here is composed of some 2000 lines of FORTRAN and MACRO programs, totalling 60 000 bytes of machine code. This is executed in overlays in about 43 000 bytes of machine memory. One thousand bytes of space are reserved for compiled code which, although sounding rather small, is adequate for highly complex programs due to the compiling method chosen.

Space is available for 8192 characters of PPL2 program which corresponds to about 250 ′average′ lines. To cope with this there is enough table space for 80 variable names which is found to be a suitable number for the total amount of text space. FOR loops and IF statements are each permitted to be nested to a depth of 10 levels, this being determined by the sizes of internal tables and found to be more than adequate.

In short, fairly simple techniques have been used in an unconventional manner to produce an extremely flexible system for developing and testing image processing algorithms. The need for a large and very expensive mainframe computer has been avoided.

END	-->	0 ´EOI´ 0 ´00´ 0
´[´	-->	´INDEX´ (
´[[´	-->	(0 ´FORWARD´ 0 ´0´ (
´[[+´	-->	(0 ´FORWARD´ 0 ´0´ (
´[[-´	-->	(0 ´REVERSE´ 0 ´0´ (
´]´	-->	)
´]]´	-->	) ´FEND´ 0)
´+]]´	-->	) ´FEND´ 0)
´-]]´	-->	) ´REND´ 0)
´@´	-->	0 ´CALL´
´IF´	-->	(0 ´IF´ 0 ´0´ (
´THEN´	-->	) ´THEN´ 0 ´0´ (
´ELSF´	-->	) ´ELSF´ 0 ´0´ (
´ELSE´	-->	) ´ELSE´ 0 ´0´ (
´FI´	-->	) ´FI´ 0)
´FOR´	-->	(0 ´FOR´ (
´FROM´	-->	) ´FROM´ (
	-->	(0 ´OFROM´ (
´BY´	-->	) ´BY´ (
	-->	(0 ´OBY´ (
´TO´	-->	) ´TO´ (
	-->	(0 ´OTO´ (
´WHILE´	-->	) ´WHILE´ 0 ´0´ (
	-->	(0 ´OWHILE´ 0 ´0´ (
´DO´	-->	) ´DO´ 0 ´0´ (
	-->	(0 ´ODO´ 0 ´0´ (
´OD´	-->	) ´OD´ 0)
´GOTO´	-->	0 ´GOTO´
´EXIT´	-->	0 ´EXIT´ 0
´MODULO´	-->	´MODULO´
READ	-->	0 ´READ´
WRITE	-->	0 ´WRITE´

Figure 4.6 Keyword Conversions

OPERATOR		Pri	Mode	Result of A OPERATOR B		
-	(unary)	-12	I/C	Twos complement of B; A ignored		
~	(unary)	-12	I/C	Ones complement of B; A ignored		
*	MULT	11	I/C	A * B	Integer result	
/	DIV	11	I/C	A / B	ditto	
+	ADD	10	I/C	A + B	ditto	
-	SUB	10	I/C	A - B	ditto	
=	.EQ.	8	I/C	A = B	Logical values;	TRUE = -1
#	.NE.	8	I/C	A # B	ditto	FALSE = 0
>	.GT.	8	I/C	A > B	ditto	
<	.LT.	8	I/C	A < B	ditto	
>=	.GE.	8	I/C	A >= B	ditto	
<=	.LE.	8	I/C	A <= B	ditto	
&	AND	7	I/C	A & B	Bit by bit logical	AND
!	OR	6	I/C	A ! B	ditto	OR
$	EX OR	5	I/C	A $ B	ditto	EXCLUSIVE OR
,		-4	I/C	0		
:=	ASSIGN	-3	I/C	B	A gets the value of B	
;		2	I/C	B	A ignored	
'IF'		20	I/C	A	B ignored	
'THEN'		20	I/C	A	B ignored	
'ELSF'		20	I/C	A	B ignored	
'ELSE'		20	I/C	A	B ignored	
'FI'		20	I/C	A	B ignored	
'0'		20	I/C	B	A ignored	Dummy
'00'		0	I/C	B	A ignored	Dummy
'INDEX'		15	I/C	ADDRESS	Address of A + 2*(value of B)	
'FORWARD'		20	I	0	Generate code; Set Compile mode	
			C	ERROR 7		
'REVERSE'		20	I	0	Generate code; Set Compile mode	
			C	ERROR 7		
'FEND'		20	I	ERROR 3		
			C	B	Code is run; Set Interpret mode	
'REND'		20	I	ERROR 3		
			C	B	Code is run; Set Interpret mode	
'GOTO'		20	I	0		
			C	ERROR 8		
'EXIT'		20	I	ERROR 4		
			C	B	Generate code	
:	LABEL	20	I/C	B	A ignored	
?>	MAX	9	I/C	The greater of A and B		
?<	MIN	9	I/C	The lesser of A and B		
?+	(unary)	-12	I/C	The absolute value of B A ignored		
?¦	(unary)	-12	I/C	IF B<0 THEN 0 ELSF B>255 THEN 255 ELSE B FI ignore A		
'MODULO'		14	I/C	The remainder after dividing A by B		
'E-O-I'		1	I/C	-none-	The program terminates	

Figure 4.7 Actions of the Operators

OPERATOR	Pri	Mode	Result of A OPERATOR B	
´FOR´	20	I/C	0	Tables are built
´FROM´	20	I/C	0	ditto
´BY´	20	I/C	0	ditto
´TO´	20	I/C	0	ditto
´WHILE´	20	I/C	0	ditto
´DO´	20	I/C	0	ditto
´OD´	20	I/C	0	ditto
´OFROM´	20	I/C	0	ditto
´OBY´	20	I/C	0	ditto
´OTO´	20	I/C	0	ditto
´OWHILE´	20	I/C	0	ditto
´ODO´	20	I/C	0	ditto
READ	-4	I	0	
		C	ERROR	No I/O in [[...]]
WRITE	-4	I	0	
		C	ERROR	No I/O in [[...]]

Notes -

Pri = Priority of operator (Negative values indicate right to left operators, the magnitude of the priority determines the order of evaluation.

I = Interpret mode. i.e. outside [[...]]

C = Compile mode. i.e. within [[...]]

I/C = Interpret or Compile modes.

Figure 4.7 (Continued)

5. IMAGE STORAGE AND ACCESS LOGIC

5.1 INTRODUCTION

Before any image processing can be contemplated an image must be made available to the computer. Images may be obtained from a wide variety of sources; probably the most versatile is the standard television camera.

Television cameras have existed for more than forty years and have undergone numerous improvements in that time. Modern cameras can be obtained with a wide variety of different tubes, each with its own advantages, and in many styles of construction for use in a large number of applications. Adaptors are available for viewing anything from the microscopic to the stars; from freezing conditions to the inside of a furnace. They may also be placed in situations that a man could not tolerate.

Standards of operation have been laid down and are well established so that cameras may be interchanged with ease. A commonly used standard is that defined as CCIR(625-line) which appears both industrially and in domestic television sets. This system provides high resolution motion pictures.

Fully semiconductor cameras using the newly developed charge coupled device image sensor are now becoming available. These are being developed to provide signals conforming to the CCIR (625-line) standard and will be fully compatible with current cameras.

Signals from such a camera are analogue in nature and provide, in addition to the image information, signals for the synchronisation of the display device. In order to use these signals for digital image processing they need to be decoded and converted into a suitable digital form.

The resolution of cameras is such that they may provide more than forty million bits of information in one second. This rate of information transfer requires very fast machinery to handle and process it.

Research into the type of processing to be performed on the signals does not normally need to occur at this speed. Development of algorithms can adequately be achieved with much slower machinery and the resultant methods transferred to special hardware.

Similarly it is unnecessary to work with a full resolution image - lower resolution versions can be used to prove the method. However, sufficient results should be retained to pick out the detail required for the process under investigation. Since it was not the aim of this project to attempt scene analysis but to study simple groups of objects such as a few nuts and bolts on a table, a resolution of 128 by 128 pixels, 16 384 pixels per image, was felt to be adequate.

Allowing eight bits per pixel for grey scale presentation requires some 16 000 bytes per image. Provision of several image spaces soon uses up a great deal of memory and while memory itself is not too expensive it quickly uses up address space on the host computer. To reduce the computer´s overhead and still provide a reasonable number of images, the memory for them has been placed in a separate interface unit.

In addition to the image memory, fast image input and output convertors have been provided with a novel accessing structure which increases program speed. Use of this accessing method has resulted in the ability to access over 250K bytes of image memory in only a 4K byte address space on the host computer.

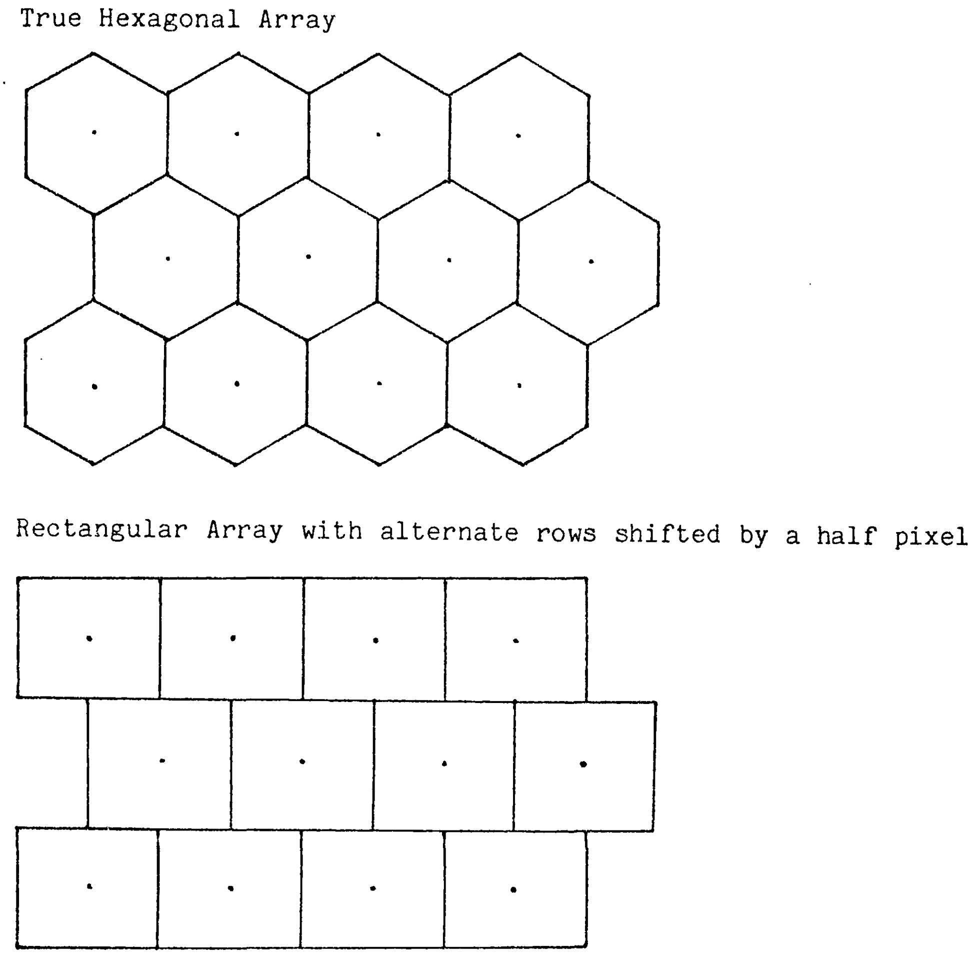
Space is available for storage of up to 16 images each of 128 by 128 pixels by 8 bits per pixel; a total of 128 bit planes. The well known CLIP4 image processing system has 32 bit planes of 96 by 96 pixels.

5.2 THE DIGITISATION SCHEME

Triangles, rectangles or hexagons may be used to form a regular tessellation covering the image. The triangular tessellation is very unpopular due to the over-complex nature of the algorithms using it. On the other hand both rectangular and hexagonal tessellations are in use. Hexagons have received relatively little attention, perhaps due to the complexity of accessing neighbouring window elements, but do provide the advantage of always being 6-connected. Rectangles are most commonly used: they can be simply accessed in most computing languages, but care has to be taken over whether to treat them as 4- or 8-connected. To allow investigation of algorithms on either rectangular or hexagonal tessellations the image storage circuitry has been designed to allow both.

Conversion of the incoming signal to a digital pattern is achieved by sampling the level at regular intervals and storing the digital value of the signal in a memory array. The intervals chosen depend upon whether a rectangular or hexagonal tessellation is being used.

The hexagonal array may, to a first approximation, be considered a rectangular array in which alternate rows are shifted a half-cell sideways (see fig 5.1). It is easy to achieve this shifting electronically and the mode of operation can be controlled by the computer. This extends the flexibility of the hardware with very little extra circuitry.



True Hexagonal Array

Rectangular Array with alternate rows shifted by a half pixel

Centre points marked with a dot.

Figure 5.1 Approximation of a Hexagonal Array

5.3 TIMING

Each line scan in the CCIR (625-line) standard takes 64uS to complete; of this 12uS is not used for image information, it being the blanking period which is required to suppress spurious signals during the ′flyback′. Thus there is some 52uS of image information per line. The bandwidth of the image information is not generally much more tham

6MHz for the majority of the cameras available at reasonable cost. It is not possible to extract more than about 800 pixels per line of information.

Frames, composed of 625 line scans in two interlacing fields of the image, take 40mS to complete. Of this time about 4mS are required for frame ´flyback´ leaving some 560 lines of information. The resolution chosen for this project of some 128 lines of 128 pixels each is well within this maximum limit.

Image information has to be sampled at approximately 400nS intervals at this resolution. It is possible to buy, from stock, analogue to digital convertors that can easily cope with this data rate at 6-, 7- or 8-bit resolution. At the time this project was undertaken only the 6-bit convertor was available at low enough cost; but with future diminishing product costs in mind the image memory was built for 8-bit resolution, enabling a future system upgrade.

Generating the timing signals to the CCIR(625-line) standard would have required a large number of integrated circuits were it not for the ZNA134J video timing generator from Ferranti [30]. This device needs to be supplied with a 2.5625MHz clock to produce all the required timing signals; this frequency converts to a clock time of 390nS which is ideal for the conversion clock. Frequency stability is ensured by deriving timing from a 10.25MHz crystal oscillator and dividing the frequency by four to feed the ZNA134J.

Counters are also driven to generate row and column addresses across the image, to be used as addresses for the memory when storing or retrieving the image for display. At the resolution required there is no need to account for scan interlace; furthermore it is only necessary to increment the row counter every other line for 128 active lines from 280. The counters are reset by the blanking signals from the ZNA134J and a short delay provided by auxilliary counters before beginning to count pixels or lines; this ensures a centered image. The column counter can be further delayed by a half pixel time on alternate rows when in hexagonal tessellation mode.

5.4 IMAGE PROCESSING

New images are formed by generating new pixel values as some function of the pixels in the original image. It is usually only necessary to use the corresponding pixel and its near neighbours from the original image; this small set of pixels is known as a window on the image. The size of the window determines the number of neighbours available for the function; in the extremes the window may be of zero size (to produce a constant image not dependant on the original) or as large as the image itself (for example, to find the mean pixel value). All image processing algorithms fall within these limits. It is usually only necessary to use a small number of neighbouring pixels, that is only a small window. The majority of published algorithms use only a 3 by 3 pixel window consisting of the corresponding pixel and its eight immediate neighbours, although larger windows are increasingly encountered.

Having defined the function for a window it is necessary to apply it to the whole image. Given a parallel processing machine it is necessary to feed the function to all the processors and issue the execute command. Execution on a sequential processor is achieved by evaluating the function with the window centered on each of the image pixels in turn.

What follows relates to a sequential machine used for image processing; it is likely to be some time before parallel processing machines completely oust them in this application.

5.5 ACCESSING WINDOW POINTS

Images are two dimensional arrays of pixels but must be stored as a single dimensional list of elements and a two- to one-dimensional mapping must be provided to access the desired point. The ability to declare a two dimensional array is provided in most high level languages, together with a suitable mapping which is usually invisible to the user, it being contained within the compiler. The subscript values are shown in figure 5.2.

A general point is IMAGE(X,Y) from which its neighbours can be given general subscripts, as shown in figure 5.3.

If a hexagonal tessellation is used a further complication arises in that the subscript referring to a neighbour on a line above or below the centre depends not only on the neighbour but on which line contains the centre point (see figures 5.4 and 5.5).

When accessing neighbouring image points in a window it must be remembered that when the centre point is near the edge of the image some of the neighbours may be off the edge. There are two ways round this:

1) to restrict the scan so that no neighbour within the window being used is off the edge of the image.

2) to detect that this has happened and take action to prevent spurious results.

Of these the latter is most often used in a simplified form. It is usually sufficient to return a particular value if the pixel is off the edge of the image and otherwise undefined. This "edge value" is set up before the scan commences and then used when required. Figure 5.6 shows a simple FORTRAN program to smooth an image by generating a resultant image in which each pixel has the value of the arithmetic

```
(1,1)  (2,1)  (3,1)  (4,1)  (5,1)  (6,1)  ... etc. ...
(1,2)  (2,2)  (3,2)  (4,2)  (5,2)  (6,2)  ... etc. ...
(1,3)  (2,3)  (3,3)  (4,3)  (5,3)  (6,3)  ... etc. ...
(1,4)  (2,4)  (3,4)  (4,4)  (5,4)  (6,4)  ... etc. ...
(1,5)  (2,5)  (3,5)  (4,5)  (5,5)  (6,5)  ... etc. ...
(1,6)  (2,6)  (3,6)  (4,6)  (5,6)  (6,6)  ... etc. ...
 ...    ...    ...    ...    ...    ...   up to the
 etc.   etc.   etc.   etc.   etc.   etc.  limits of
 ...    ...    ...    ...    ...    ...   the array
```

Figure 5.2 Subscript Values (Rectangular Tessellation)

```
(X-2,Y-2)  (X-1,Y-2)  (X,Y-2)  (X+1,Y-2)  (X+2,Y-2)
(X-2,Y-1)  (X-1,Y-1)  (X,Y-1)  (X+1,Y-1)  (X+2,Y-1)
(X-2,Y)    (X-1,Y)    (X,Y)    (X+1,Y)    (X+2,Y)
(X-2,Y+1)  (X-1,Y+1)  (X,Y+1)  (X+1,Y+1)  (X+2,Y+1)
(X-2,Y+2)  (X-1,Y+2)  (X,Y+2)  (X+1,Y+2)  (X+2,Y+2)
```

Figure 5.3 Neighbouring Subscripts (Rectangular)

```
                                                        line
(1,1)  (2,1)  (3,1)  (4,1)  (5,1)  (6,1)  ... etc. ...     a
   (1,2)  (2,2)  (3,2)  (4,2)  (5,2)  (6,2)  ... etc. ...  b
(1,3)  (2,3)  (3,3)  (4,3)  (5,3)  (6,3)  ... etc. ...     a
   (1,4)  (2,4)  (3,4)  (4,4)  (5,4)  (6,4)  ... etc. ...  b
(1,5)  (2,5)  (3,5)  (4,5)  (5,5)  (6,5)  ... etc. ...     a
   (1,6)  (2,6)  (3,6)  (4,6)  (5,6)  (6,6)  ... etc. ...  b
 ...    ...    ...    ...    ...    ...      up to the
    etc.   etc.   etc.   etc.   etc.   etc.  limits of
 ...    ...    ...    ...    ...    ...      the array
```

Figure 5.4 Subscript Values (Hexagonal Tessellation)

```
for lines "a"
            (X-1,Y-1)  (X,Y-1)
        (X-1,Y)    (X,Y)    (X+1,Y)
            (X-1,Y+1)  (X,Y+1)

for lines "b"
            (X,Y-1)    (X+1,Y)
        (X-1,Y)    (X,Y)    (X+1,Y)
            (X,Y+1)    (X+1,Y)
```

Figure 5.5 Neighbouring Subscripts (Hexagonal)

mean of the nine pixels in a 3 by 3 window on the original. When a pixel is off the image it is replaced by the constant value 0.

Clearly a great deal of work has to be done to calculate the position of the neighbours and to check the validity of those positions. This work is in addition to that required to achieve the required result. The first result of this is that the program does not clearly show the image processing being performed. The second is that a great deal of machine code is produced, all within the scan loop and hence rather slow in execution (for the program given there are 108 machine code instructions within the loop). Similar overheads will be incurred by any high level language operating on internal arrays of values.

Another problem that can occur in many computers is the large amount of memory required for image storage. Images of 128 by 128 pixels contain a total of 16 384 pixels, and for grey scale images a byte must be allocated for each. This is one quarter of the total memory available to most microprocessors and quite a large proportion of that in a mini-computer. It would be convenient to be able to store images without committing such a large amount of address space.

By using a novel address mapping architecture the image storage hardware described in this chapter overcomes the problems described above. The resultant device has enabled PPL2 to compile shorter and faster code than can be achieved using the more common high level languages such as FORTRAN or ALGOL using data in their own arrays.

5.6 A FAST SOLUTION

Although time consuming in software the address calculations are very simple, only requiring an offset of a few pixels or lines from the centre point. This can be achieved by using standard TTL integrated

```
      PROGRAM MEAN
C Declare the variables and images.
      INTEGER X,Y,TOTAL,INPUT(128,128),OUTPUT(128,128)
C Scan the image.
      DO 110 Y=1,128
      DO 100 X=1,128
C Sum the points.
C The centre is always OK.
      TOTAL=INPUT(X,Y)
C Checking that the rest are valid
      IF(X.GE.2)                 TOTAL=TOTAL+INPUT(X-1,Y)
      IF(X.LE.127)               TOTAL=TOTAL+INPUT(X+1,Y)
      IF(Y.GE.2)                 TOTAL=TOTAL+INPUT(X,Y-1)
      IF(Y.LE.127)               TOTAL=TOTAL+INPUT(X,Y+1)
      IF(X.GE.2.AND.Y.GE.2)      TOTAL=TOTAL+INPUT(X-1,Y-1)
      IF(X.GE.2.AND.Y.LE.127)    TOTAL=TOTAL+INPUT(X-1,Y+1)
      IF(X.LE.127.AND.Y.GE.2)    TOTAL=TOTAL+INPUT(X+1,Y-1)
      IF(X.LE.127.AND.Y.LE.127)  TOTAL=TOTAL+INPUT(X+1,Y+1)
C Divide by the total number of points to get the mean.
      OUTPUT(X,Y)=TOTAL/9
C End of the scan.
100   CONTINUE
110   CONTINUE
C We've finished.
      STOP
      END
```

Figure 5.6 Example Image Processing Program in FORTRAN

circuit adders fed with the centre point co-ordinate and the offset required.

Mapping the window rather than the image into the computer's address space brings two significant advantages:

1) A very large reduction of the amount of memory space occupied by the images. Only as much address space as total window area is required, a 15 by 15 window, which is quite large by image processing standards, requires only 15*15 = 225 bytes of storage per image. Compared to a 128 by 128 pixel image this is a factor of 80 reduction; a greater reduction is achieved on larger images.

2) The software can directly access points in the window without any address calculation overheads. The calculations occur at high

speed in the interface and can be made quite invisible to the user, both logically and timewise. Window points appear as a one dimensional array of values and by assigning a name to each window element (as in PPL2) all run time address calculations are avoided.

Additionally, outputs from the adders can be used to detect an attempted access to a point outside the image boundaries. The interface can, in this case, return a defined edge value to the computer.

Large reductions in program length, and hence in execution time, result from this technique; the example program in figure 5.6 requires only 13 machine code instructions rather than the 108 needed in FORTRAN.

5.7 CIRCUIT DESCRIPTION

Offsets from the centre point do not bear a simple relationship to the window point address fed to the interface. To avoid complex calculations the offsets are stored in a Read Only Memory whose input address is the window point required and output is the address to be used by the adders. It is easy to arrange for completely separate tables of offsets for rectangular and hexagonal tessellations; the hexagonal version requiring two tables for the cases of odd or even lines (see figure 5.5).

Each table is as long as there are addresses in the window; a 15 by 15 window has 225 addresses which is just below the rather convenient number 256. Rectangular tessellation requires one table whilst hexagonal requires two; it is convenient to assign two tables to the rectangular mode and fill them with identical values, thus avoiding some signal switching. A block diagram of the arrangement is given in figure 5.7.

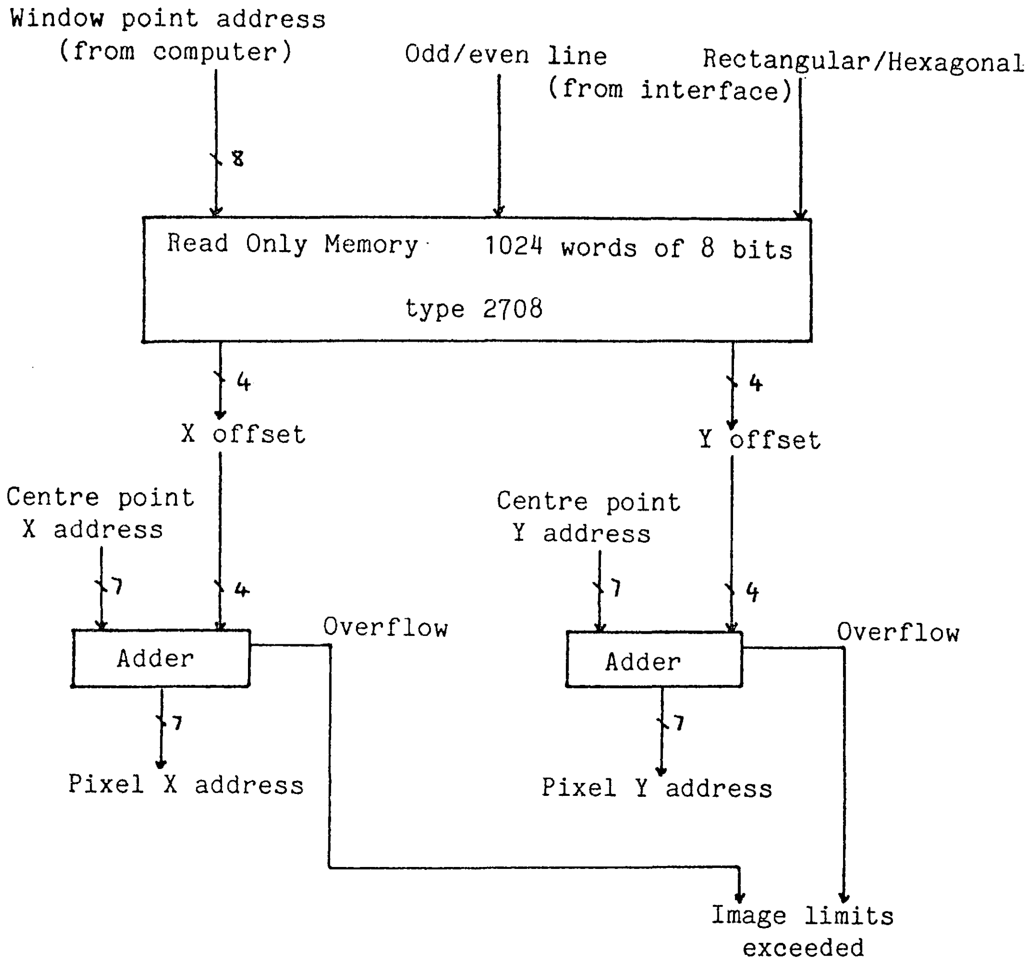


Figure 5.7 Address Mapper - Block Diagram

In addition to the image points it is necessary to be able to access control registers which determine the centre point address, the mode (rectangular or hexagonal), which image is to input from the camera, which is to displayed and the value to be returned if the image limits are exceeded.

Each byte from the ROM is divided into two nibbles, one for the X offset and the other for Y. The 4-bit value is interpreted as a two´s complement value in the range -7 to +7 to give the offsets needed for a 15 by 15 window. The remaining code, binary 1000 or -0, is used to signal special addresses that are used to access the control registers.

The most significant carry bits out of the adders are used to determine that the image limits have been exceeded. When this happens the address formed is replaced with the address of the register containing the edge value for the image being accessed.

5.8 OTHER FACILITIES

Image input, from a television camera, and output, to a monitor, is also provided within the interface. These facilities enable high speed data acquisition and continuous display of any desired image, even as it is being processed.

5.9 AS THE USER SEES IT

Sixteen blocks of memory are available to the user, one for each image within the interface. Each block has 256 addresses within it; of these 225 are required for the window and the rest are either control register addresses or unused.

An address map for one image block is shown in figure 5.8. The maps for the other images are the same. Addresses marked with an asterisk are registers common to all image spaces and may be accessed from within any image space.

5.9.1 The Window

Each point in the window has an address in the image block, the centre point is at address 0 and the other points at increasing addresses. A complete list of addresses is given in figure 5.9.

PDP11 Address (octal)	Contents
776	Edge value of the image
774*	Number of the image being displayed
772*	Number of the image to be input
770*	Mode (Rectangular or Hexagonal)
766*	Y co-ordinate of window centre point
764*	X co-ordinate of window centre point
762 ¦ 702	Not used
700 ¦ 0	Pixels in the window

Figure 5.8 Address Map

5.9.2 The Centre Point

The address of the centre point of the window must be given to the interface before any processing on the window is performed. The X and Y co-ordinate values may be given separately in addresses 764 and 766 respectively or use may be made of the concatenated registers in adresses 754 and 756 or 760 and 762. These last two pairs of registers allow a simple scanning method to be used with microprocessors having a 16-bit register that can be incremented or decremented. By cycling the 16-bit register and writing its value to the interface a complete scan may be accomplished with the minimum of programming.

```
X! -7   -6   -5   -4   -3   -2   -1    0    1    2    3    4    5    6    7
Y
-7! 196  195  194  193  192  191  190  189  188  187  186  185  184  183  182 -7
-6! 197  144  143  142  141  140  139  138  137  136  135  134  133  132  181 -6
-5! 198  145  100   99   98   97   96   95   94   93   92   91   90  131  180 -5
-4! 199  146  101   64   63   62   61   60   59   58   57   56   89  130  179 -4
-3! 200  147  102   65   36   35   34   33   32   31   30   55   88  129  178 -3
-2! 201  148  103   66   37   16   15   14   13   12   29   54   87  128  177 -2
-1! 202  149  104   67   38   17    4    3    2   11   28   53   86  127  176 -1
 0! 203  150  105   68   39   18    5    0    1   10   27   52   85  126  175  0
 1! 204  151  106   69   40   19    6    7    8    9   26   51   84  125  174  1
 2! 205  152  107   70   41   20   21   22   23   24   25   50   83  124  173  2
 3! 206  153  108   71   42   43   44   45   46   47   48   49   82  123  172  3
 4! 207  154  109   72   73   74   75   76   77   78   79   80   81  122  171  4
 5! 208  155  110  111  112  113  114  115  116  117  118  119  120  121  170  5
 6! 209  156  157  158  159  160  161  162  163  164  165  166  167  168  169  6
 7! 210  211  212  213  214  215  216  217  218  219  220  221  222  223  224  7
    -7   -6   -5   -4   -3   -2   -1    0    1    2    3    4    5    6    7
RING       X,Y       POINTS     NO.POINTS  CUM.NO.  NO.OUTSIDE
NO.     FROM  TO    FROM  TO               POINTS

0          0   0       0   0        1         1          224
1         -1   1       1   8        8         9          216
2         -2   2       9  24       16        25          200
3         -3   3      25  48       24        49          176
4         -4   4      49  80       32        81          144
5         -5   5      81 120       40       121          104
6         -6   6     121 168       48       169           56
7         -7   7     169 224       56       225            0
```

Figure_5.9__Window_Point_Addresses_(decimal)

5.9.3 Mode

The value set within this register determines whether the interface is to operate in the rectangular or hexagonal mode. Zero represents the rectangular mode and 1 the hexagonal mode. This value selects the appropriate mapping from the ROM and adjusts the timing for image input and output.

5.9.4 Input Image Number

Writing a value to this location starts the digitisation process to capture the next frame of image information from the camera. When an input is in progress the computer cannot gain control of the interface as this would probably corrupt the incoming data. If the computer writes to the interface in this time the command is ignored. If it tries to read it will receive the value -1. This may be used as an indicator that the interface is busy. When the interface has finished digitising the image this location will return the number written to it, i.e. the last image to be digitised.

5.9.5 Output Image Number

Display of a particular image is achieved by writing the number of the image to this location. This value can be read when desired.

5.9.6 Edge Value

This value is returned to the computer if an attempt to read a value from a point that is off the edge of the image.

5.10 CONSTRUCTION

Due to the very large amount of circuitry involved it was decided to divide the construction into a number of units each of only moderate complexity. A block diagram of the units involved is given in figure 5.10.

5.10.1 Timing and Control

This board contains the master crystal oscillator and frequency dividers to provide all the timing signals required by the interface. It also contains the bus arbitration circuitry that ensures that only one device at a time tries to drive the bus and that all drive changeovers are synchronised.

5.10.2 Display

Data is taken from the bus and presented to an ANALOGIC MP8308 digital to analogue convertor with composite sync and blanking. It produces a CCIR (625-line) composite video signal for use with a standard video monitor.

The sync and blanking signals are converted to 1v standard signals to feed the camera to synchronise it to the interface.

5.10.3 Colour Display

Colour output can be used to great effect to highlight particular areas of the image. The display is not intended to produce a colour image but merely to be able to produce colours. The eight data bits are split into 3-bits of red gun grey scale, 3-bits of green gun grey scale and 2-bits of blue gun grey scale. These signals are encoded to the PAL standard using the National Semiconductor LM1886 and LM1889 I.C.´s and fed, at UHF, to a domestic colour television as monitor.

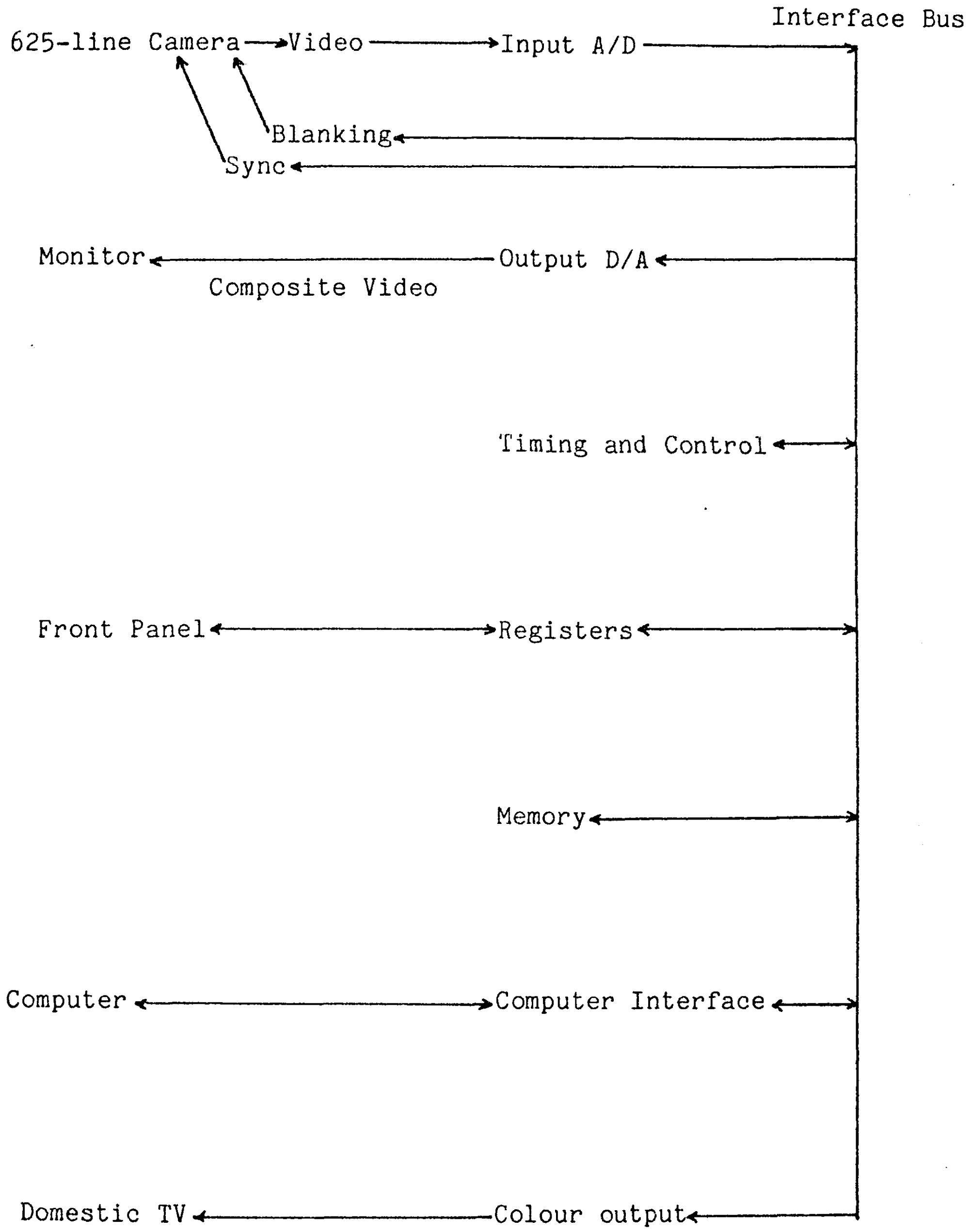


Figure 5.10 The Video Interface - Main Units

5.10.4 Input

Video from the camera is digitised by a TRW TDC1014J fast 6-bit analogue to digital convertor. The digital values are fed immediately into the image memory.

5.10.5 Memory

The image display runs all the time and accesses all the image points in turn enabling a dynamic memory to be used without any further complexity in refresh circuits. 16Kx1 (4116) memory chips were used, requiring eight devices per byte image.

5.10.6 Computer Interface

Access commands from the computer arrive at high speed (about 1MHz) over a fairly long length of cable (36 feet). 75-series balanced line drivers were used for this application and this board houses the necessary integrated circuits with tri-state drivers for the internal bus.

5.10.7 Registers

The registers board contains the centre point co-ordinate, mode, input number, output number and edge value registers. Drivers for the front panel indicators and switches are also provided.

5.10.8 Front Panel

Apart from the mains on/off switch, the front panel also contains (a) indicators to show the state of the interface and (b) manual override switches. The front panel is shown in figure 5.11.

The right-hand switches marked "HEX" and "RECT" indicate the

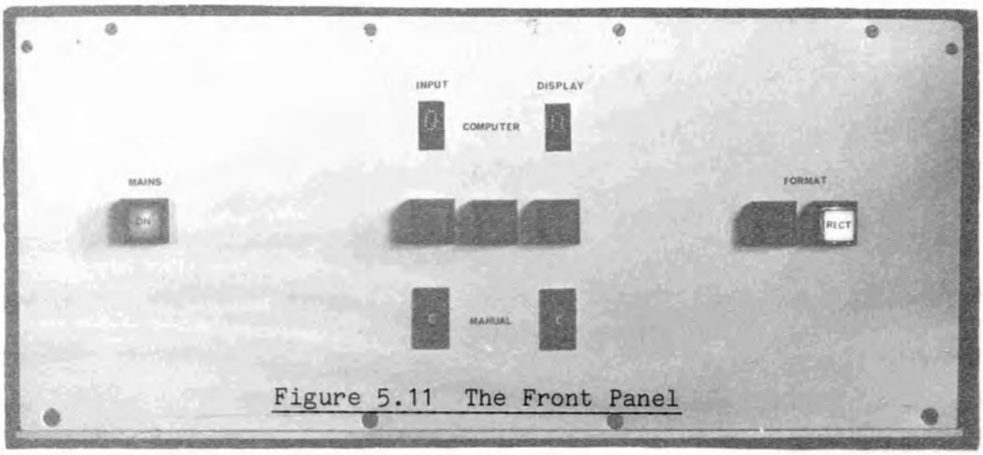
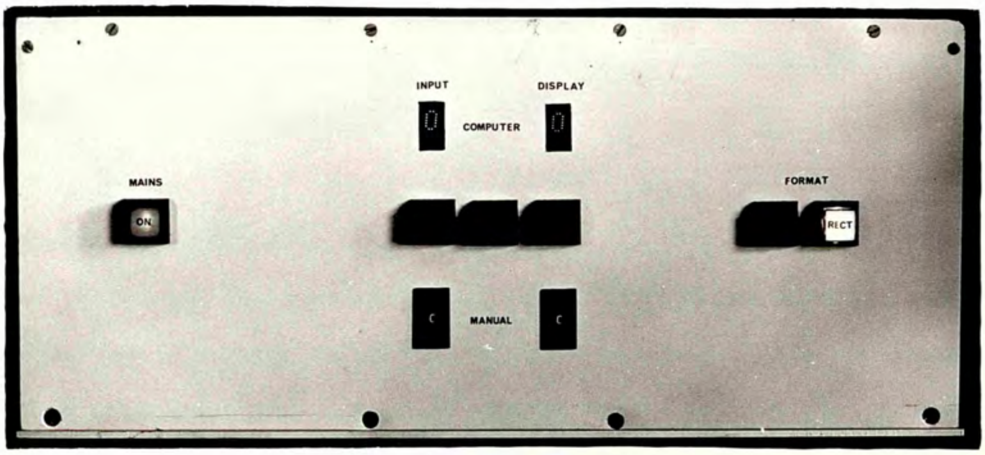


Figure 5.11 The Front Panel


INPUT
DISPLAY
COMPUTER
MAINS
ON
FORMAT
RECT
MANUAL

operating mode of the interface. Pushing a switch will select that mode, overriding any mode set under computer control.

Switch "VIEW" switches to continuous digitising mode and immediately re-converts the digital signal to video for the monitor. This provides a check that the analogue to digital and digital to analogue convertors are working correctly.

The displays at the top marked "COMPUTER" show the last image number to have been input or output by the interface under computer control. The arrow switches below them allow manual overriding to input or display the image selected on the thumbwheel switches below them.

5.11 PDP11 INTERFACE

A special board plugs in to the PDP11 unibus containing address recognition circuitry for the board and drivers for the balanced line to the interface. The interface is made to appear as word wide data so that the image values do not become negative values as they would if byte addressing were used. Bus protocol is converted from that used on the asynchronous Unibus [31] to the synchronous form required by the image storage logic.

The 16 images each require 256 words, a total of 4K words or 8K bytes which is, conveniently, one page of memory to the memory management unit and can easily be mapped into the driving program.

6. AN APPLICATION OF THE GREY SCALE QUADTREE IN SMALL PART LOCATION

6.1 INTRODUCTION

Binary images of resolution 512 x 512 contain more than one-quarter of a million bits - an eno rmous quantity of information. A grey scale image of similar resolution will contain well over one million bits. In these circumstances object recognition necessitates a vast data reduction, in the order of five orders of magnitude.

Individual pixels can often be grouped together with others having similar properties within the scene, such as belonging to an object or background. Interest in finding image descriptions on a larger scale than single points has led to schemes which generate more global descriptions of all or parts of the image. Operations on the more global attributes of the image can lead more quickly to the desired result than an unnecessarily detailed examination of the basically redundant information in many of the pixels [32]. Formation of a set of image descriptions of differing resolution provides a method whereby a very fast global search for items of interest can be followed, if necessary, by a more detailed local examination.

In this section we first of all discuss pyramidal and in particular quadtree representations. We then look at binary and grey-scale quadtree implementations and some of the particularly useful information made available for the thresholding of grey-scale images. Finally, a complete object segmentation scheme, using information derived from quadtrees, is described to cope with practical (off the camera) images of small components.

6.2 THE QUADTREE

Pyramidal structures for image representation have been proposed [33,34,35,36] as a means of data reduction which can speed up subsequent processing of the image. Such structures have been used to considerable advantage in applications such as scene matching [37]. Reliance is made upon the fact that most images contain regions of constant information in which the individual pixel carries very little information and groups can be replaced by a single overall description.

Quadtree representations are one form of pyramidal reduction technique which is easy to produce, can be generated at high speed and give a very usable data set for further processing [38,39,40,41].

Given an image that is square, with sides some power of two long, a quadtree representation may be formed as follows: if all the pixels have the same value the tree consists of only a single value, the root, with that value. If not, the image is divided into four quadrants and the root is given these four successors. The process is now repeated for each of the successors; if all pixels in the quadrant have the same value the node is given that value; if not it is further subdivided. The end result is a tree in which each node has four successors (i.e. a tree of degree 4, a quadtree); the successors may be leaves (for constant regions) or other nodes (for non-constant regions). An image of 2^n by 2^n pixels has a quadtree of maximum depth n+1. An example of the tree produced by a simple image is given in figure 6.1.

6.3 A BINARY QUADTREE IMPLEMENTATION

Only the simplest of trees can be drawn on paper, those found in quadtrees are too complex to be shown in this form. This can neatly be illustrated by displaying the depth of the leaf of the tree for each

Image

```
1 1 1 1 0 0 2 2
1 1 1 1 0 0 2 2
1 1 1 1 0 0 0 0
1 1 1 1 0 0 0 0
3 3 0 0 0 0 0 0
3 3 0 0 0 0 0 0
0 0 4 5 0 0 0 0
0 0 6 7 0 0 0 0
```

Quadtree Node

```
                                                A B
A,B,C,D  contains values or pointers to successors
                                                C D
```

Quadtree Structure

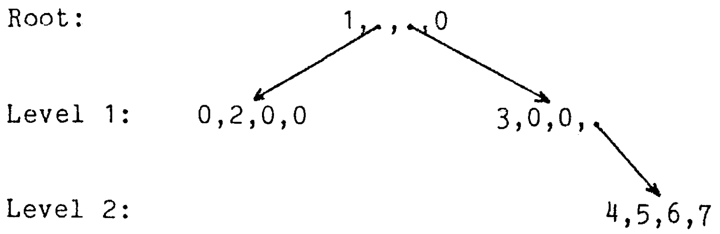


Figure 6.1 Example Quadtree

pixel in the image. This is not as compact as the tree (it requires a whole image) but does provide a useful visual representation and can be trimmed to the minimum tree if required.

Production of a quadtree for a binary image in depth format and its tree representation is very straightforward; a PPL2 program to do so is given in figure 6.2. Initially, the image is assumed to be uniform and all pixels marked as belonging to the root node. The image is then scanned to test this assumption and if it is found to be false the image is assumed to be composed of four regions at one deeper level of the tree; pixels are marked accordingly. Each of these assumptions is then tested and if the region is not uniform a deeper level is assumed. This continues until a region is discovered to be uniform, if necessary containing only one pixel.

An example of the result obtained is shown in figure 6.3, the brighter areas in the quadtree image correspond to deeper leaves in the tree. In order to make the different levels more visible the level information produced by the program has been multiplied by 36 before display.

Although this program does not employ the fastest possible strategy its speed is adequate (9 seconds on a 128^2 image using the PDP11 PPL2 system described in chapter 3) for research purposes. For industrial use a faster technique would have to be used.

It can be seen that large areas of constant value are situated near the root of the tree whilst areas of greater activity, near or on edges, fall to a much deeper level.

Depth labelling of pixels in this way allows particular levels of the tree to be extracted: figure 6.4 shows two different levels of the tree generated above. Deep leaves occur on object edges, and shallow leaves in large, uniform areas.

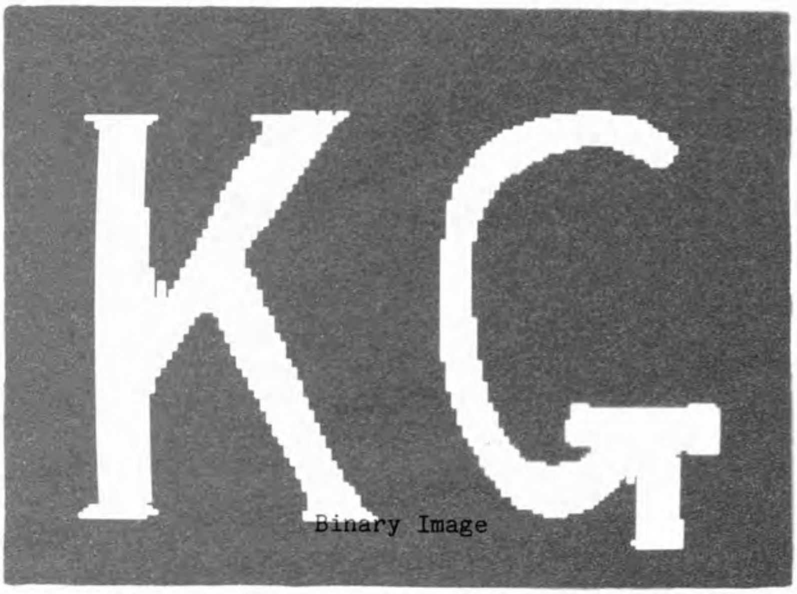
6.4 A GREY SCALE QUADTREE IMPLEMENTATION

Grey scale images, unlike binary, only very rarely contain areas of exactly the same pixel value. Real scenes generally contain both gentle brightness gradients on uniform surfaces as a result of the lighting conditions, and digitisation and noise errors in the digital form of the analogue signal. Large uniform areas can be produced by mechanical (artificial) image generation or by digitisation to only a small number of grey levels; the latter method often artificially dividing an area as the real intensity crosses a digitising threshold.

The program in figure 6.2 would work with a grey image but, except for the very rare or mechanically produced scene would result in almost

```
QUAD:
  { set all to root, initially }
      [[ Q0:=0 ]];

  { LEVEL is the level, 0 = root
    BLKSIZ is the block size, 128 = the whole image }
      BLKSIZ:=128;

  { scan for each level }
      FOR LEVEL FROM 0 TO 6 DO

  { go to top L.H. corner of each block of the specified size }
       FOR I FROM 0 BY BLKSIZ TO 128-BLKSIZ DO
        FOR J FROM 0 BY BLKSIZ TO 128-BLKSIZ DO
         X:=I; Y:=J;

  { if it needs checking - }
         IF Q0>=LEVEL THEN VALUE:=P0; SAME:=TRUE;
  { test all pixels for equality }
                       FOR X FROM I TO I+BLKSIZ-1 DO
                        FOR Y FROM J TO J+BLKSIZ-1 DO
                          SAME:=SAME & P0=VALUE
                                               OD
                                              OD;
  { if not all the same mark block as deeper level to study it
     next time through                                         }
                       IF ~SAME
                         THEN
                           FOR X FROM I TO I+BLKSIZ-1 DO
                            FOR Y FROM J TO J+BLKSIZ-1 DO
                                       Q0:=LEVEL+1
                                                   OD
                                                  OD
                       FI
         FI
                                     OD
                                    OD;

  { halve the block size for the next level }
        BLKSIZ:=BLKSIZ/2
           OD

      RETURN;
```

Figure 6.2 Quadtree Generation Program



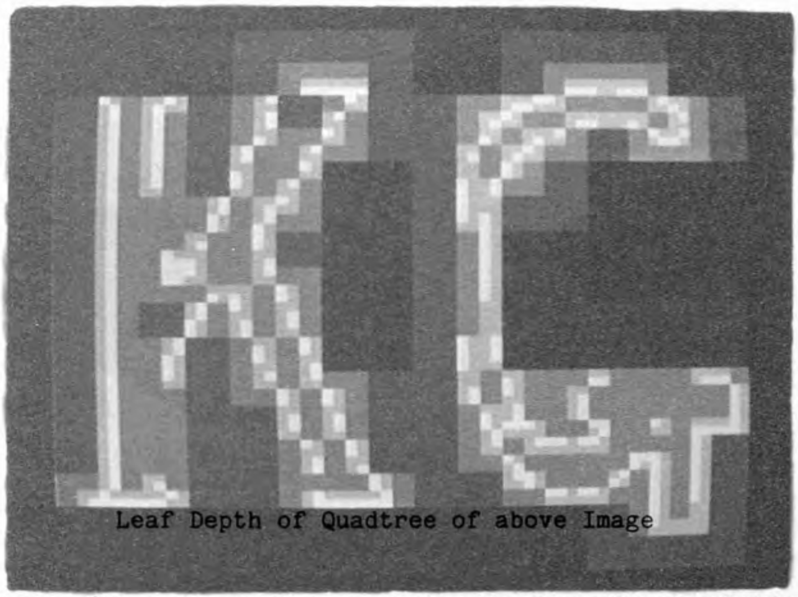
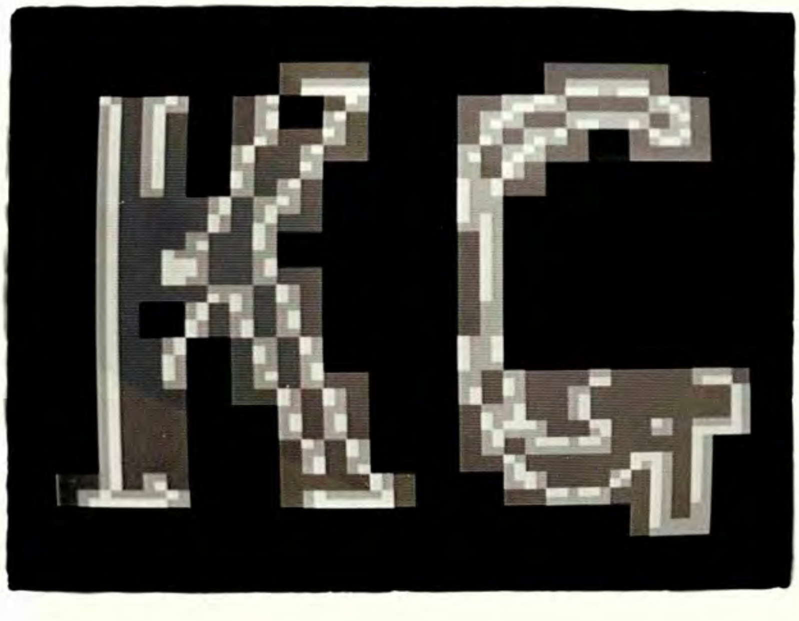
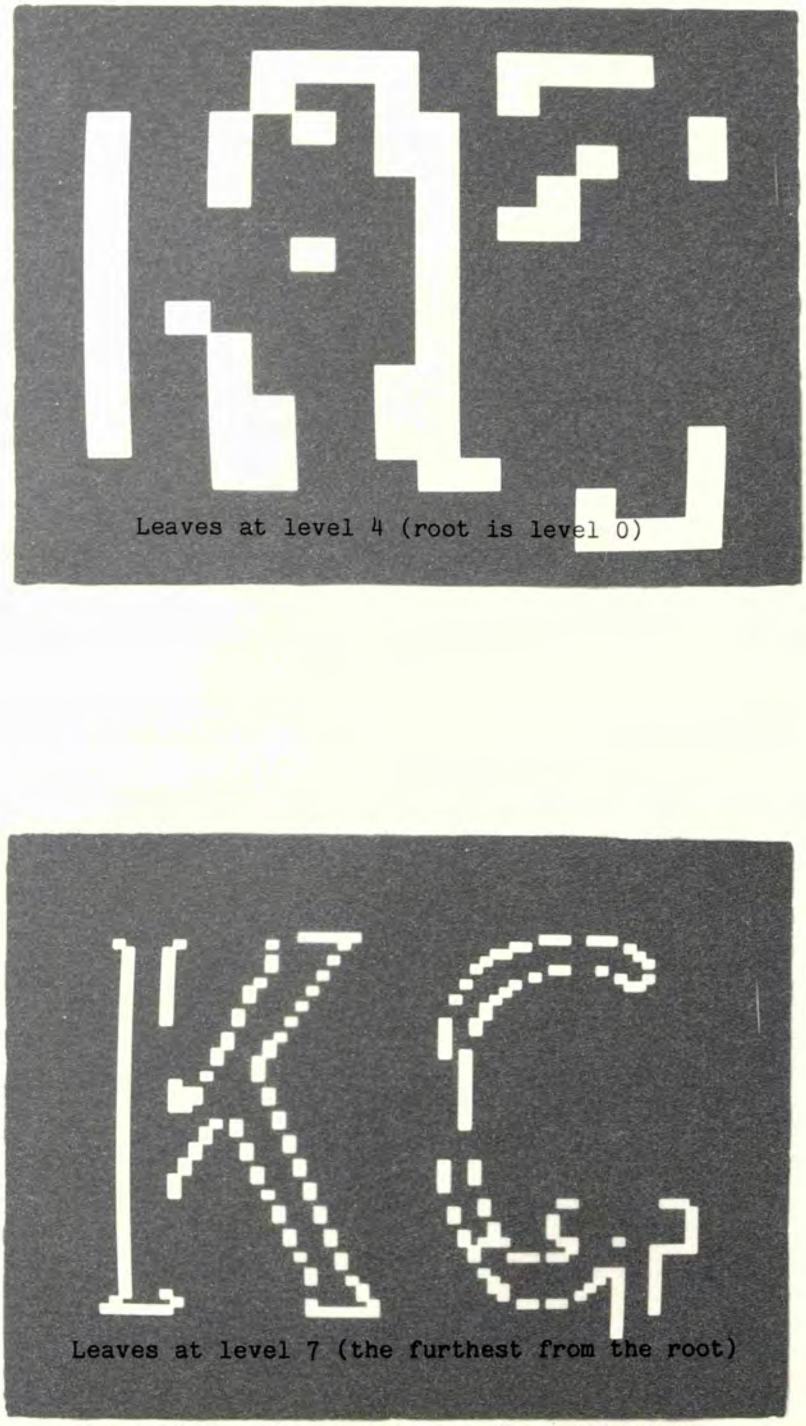


Figure 6.3 Binary Image and its Quadtree Representation

KG

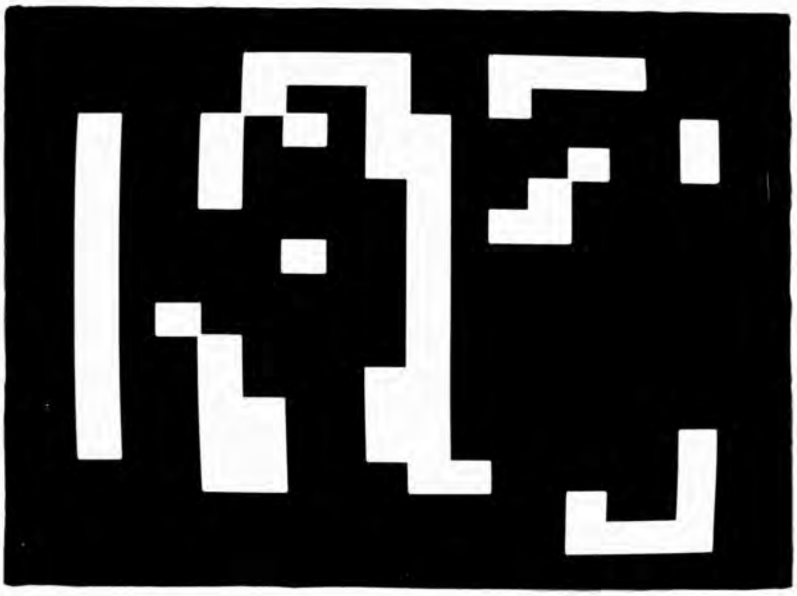

KG

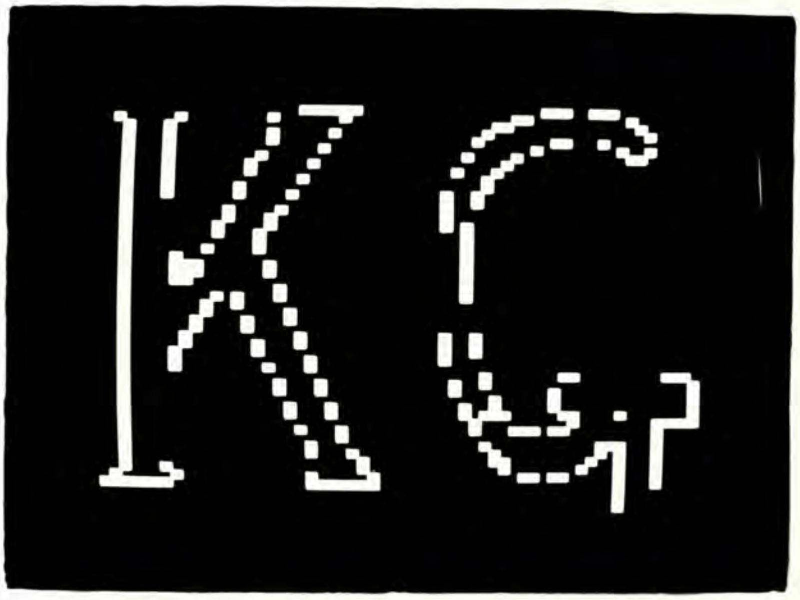


Leaves at level 4 (root is level 0)

Leaves at level 7 (the furthest from the root)

Figure 6.4 Selected levels from the quadtree





entirely single pixel blocks. This results in very little benefit, or even a data increase, and cannot be usefully employed.

Instead of requiring that an area be constant it is necessary to specify that an area contains similar pixel values for a sensible grouping into regions to be made. Any one of a large number of algorithms could be used for testing for similarity. There are conflicting requirements for such algorithms, for example noise rejection versus real object detection. Different algorithms have been suggested (e.g. [36,42]) but for the application in this chapter a very simple test appears to be adequate.

Uniformity, in this application, is determined simply by checking the total range of values in the block; the maximum value minus the minimum value has to be below a given threshold. This method is prone to error due to noise but still achieves satisfactory results without the time penalty of a more complex test. It also has the advantage of guaranteeing a known range of values within the block.

Much the same program as before is used with alteration only of the uniformity test: the grey scale version is shown in figure 6.5. The threshold value for the uniformity test has been set to a maximum range of 20, this value being determined by the lighting conditions and contrast between the objects and the background in the scene. It must be determined experimentally for the arrangements used although its value is not too critical and a few trials with different values will be sufficient.

An example of the result obtained from this program is given in figure 6.6.

```
QUAD:
  { set all to root, initially }
      [[ Q0:=0 ]];

  { LEVEL is the level, 0 = root
    BLKSIZ is the block size, 128 = the whole image
    THRESH is the maximum range of values in a constant block }
      BLKSIZ:=128; THRESH:=20;

  { scan for each level }
      FOR LEVEL FROM 0 TO 6 DO

  { go to top L.H. corner of each block of the specified size }
        FOR I FROM 0 BY BLKSIZ TO 128-BLKSIZ DO
         FOR J FROM 0 BY BLKSIZ TO 128-BLKSIZ DO
          X:=I; Y:=J;

  { if it needs checking - }
          IF Q0>=LEVEL THEN MAX:=MIN:=P0;
  { test all pixels for equality }
                       FOR X FROM I TO I+BLKSIZ-1 DO
                        FOR Y FROM J TO J+BLKSIZ-1 DO
                          MAX:=MAX?>P0;
                          MIN:=MIN?<P0
                                               OD
                                              OD;
  { if not all the same mark block as deeper level to study it
     next time through                                           }
                       IF MAX-MIN > THRESH
                         THEN
                           FOR X FROM I TO I+BLKSIZ-1 DO
                            FOR Y FROM J TO J+BLKSIZ-1 DO
                                       Q0:=LEVEL+1
                                                   OD
                                                  OD
                       FI
          FI
                                     OD
                                    OD;

  { halve the block size for the next level }
        BLKSIZ:=BLKSIZ/2
           OD

      RETURN;
```

Figure 6.5   Grey Scale Quadtree Generation Program

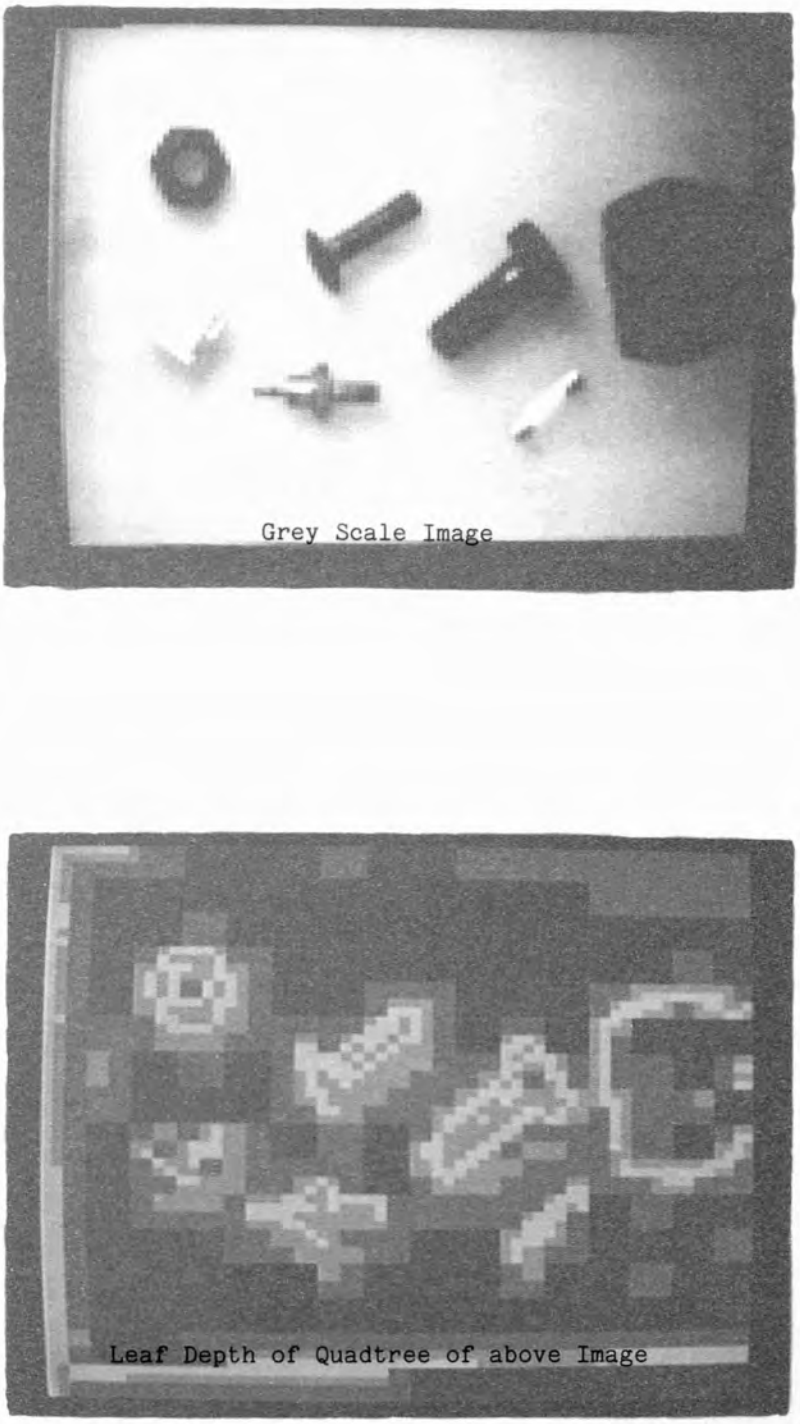
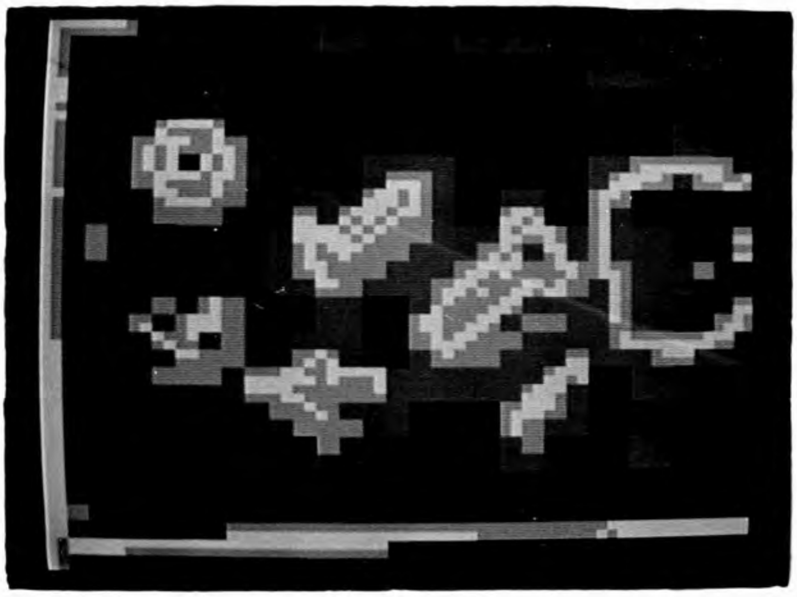


Figure 6.6 Grey Image and its Quadtree Representation



6.5 IMMEDIATELY AVAILABLE USEFUL INFORMATION

Some very useful information can be extracted from the quadtree without the need for any further processing. The larger the area of similar intensity the larger the block that can cover it and hence the nearer the root of the tree it occurs. Conversely, fast changes in intensity of the scene allow only small blocks to be used: these occur further from the root.

Deep leaves, therefore, occur at sharp changes in intensity such as the edges of objects. Unfortunately, simply interpreting deep leaves as edges is not sufficient to guarantee finding the complete edge, since some edge points may occur on the edges of large blocks near the root. It may be possible to use this data to partially remove edges from the scene: this may be useful in determining the properties of the background.

Leaves near the root describe large constant areas which, although they may border upon edges, do not contain them. Regions on large objects or expanses of background can quickly be extracted.

The depth of the leaf determines the size of the block that it describes. The root describes blocks the size of the image, the next level down those of linear dimensions half this size, the next a quarter the size and so on to the lowest level which describes individual pixels.

Since the blocks in the quadtree are fixed in position whilst objects in the scene are not so constrained, it is not certain that an object (even a square object of the size of the quadtree block) will be covered by a block of a particular size - indeed it is very unlikely. It is more probable that a block of the next smaller size or even smaller blocks will be needed to help describe the object accurately.

6.6 BACKGROUND EXTRACTION

As objects will be likely to be split into small blocks for the quadtree description it is not easy to see how objects may be reliably extracted from such a description. However, large blocks near the root of the tree will accurately describe large constant areas, any irregularities causing splitting of the block.

Provided that objects in the scene are fairly small the large blocks can only describe the areas of background between the objects. Areas of the scene that are known to be background can be extracted ignoring the objects within them.

Unlike the normal thresholding techniques of separating foreground and background in a scene, this technique is not affected by non-uniform illumination of the background. Whilst it may be argued that it is quite possible to light a scene uniformly it should be noted that a failed light bulb will alter this situation. It might be embarrassing on a production line to have to stop production to change a light bulb for the sake of the controlling machinery. Inspection by manual labour is currently immune to this type of defect and it is unlikely that an inferior method would be appreciated by the line manager.

6.7 SMALL PART DETECTION

Once a few background areas have been found it should be possible to interpolate between them to give an estimate of the complete background. Objects can then be found by simply detecting differences between the scene and the estimated background. Of course, the estimate will not correspond exactly to the actual background so an error limit must be set. The block uniformity description given in section 6.4 is useful in that it can be used to set such a limit.

6.8 A SMALL PART DETECTION PROGRAM

The ideas suggested above have been incorporated in a program for the detection of small parts. The problem has been split into three parts -

1) detection of known background,

2) interpolation of the background over uncertain areas and

3) separation of parts from the background in the scene.

The scene shown in figure 6.7 has been used as a test scene for this application; it contains a variety of different objects. The objects are all small but otherwise vary in their size and shape. The image is directly from the camera, and has been digitised but not preprocessed in any way. It can be seen that the contrast is not very good and that there is a fair amount of noise in the scene.

6.8.1 Background Detection

Assuming that the objects to be found are small, the background is defined as that area in the scene that can be divided into large blocks of similar pixel values. We need to find leaves near to the root of the quadtree. After construction of the tree these leaves can be removed and used as the known background areas. (Note that at this point we limit the size and spacing of the objects that can be detected - see section 6.10.)

Since it is only the large leaves that are required it is pointless generating the tree beyond them: thus a saving in computational effort can be achieved. It is only necessary to note whether a block is consistent or would have been subdivided if processing had continued.

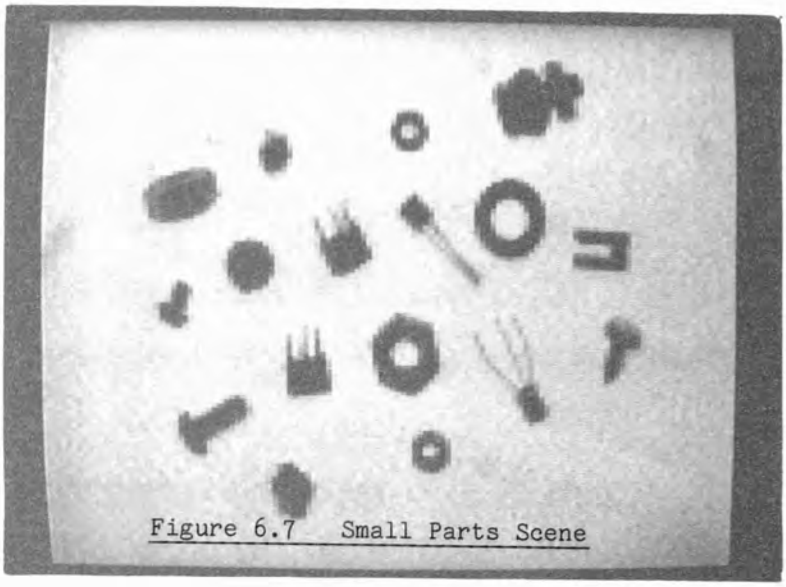


Figure 6.7 Small Parts Scene

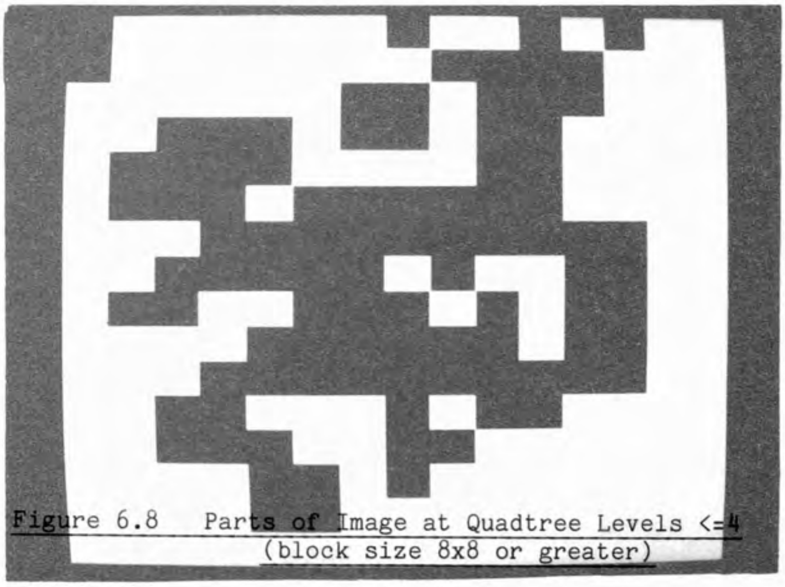
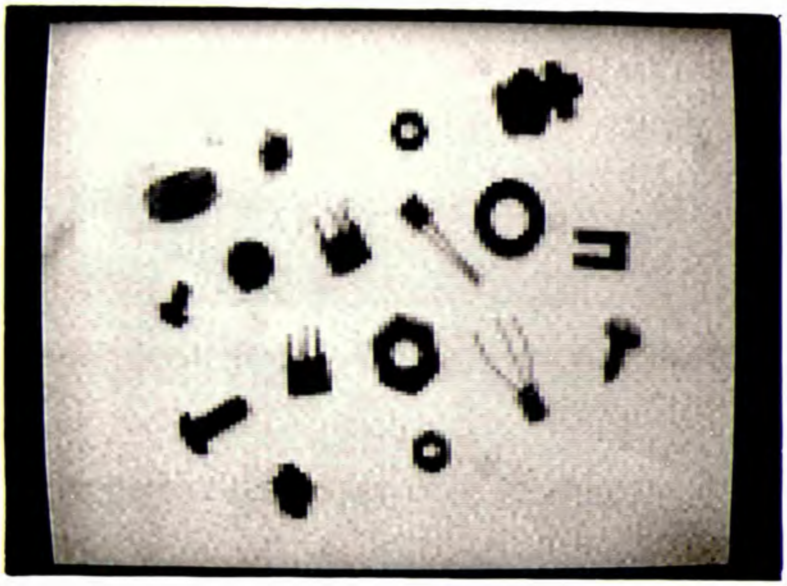
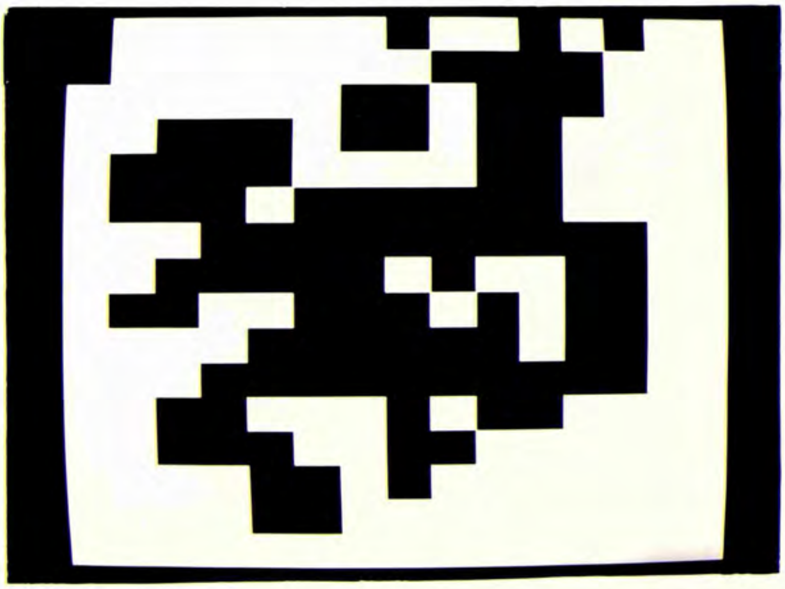


Figure 6.8 Parts of Image at Quadtree Levels <=4 (block size 8x8 or greater)





Indeed, in a program such as that given in figure 6.5 where the tree generation proceeds from the root, it is unnecessary to generate levels nearer the root than that one required. This is so because such a program examines each pixel in each of the different sized blocks: smaller blocks merely repeat the uniformity tests made further up the tree. A complete examination at a particular level will detect all those blocks that would have been placed into higher levels on the tree.

Routine QUAD in figure 6.9 performs the uniformity tests at one level only. The level is determined by the block size set in BLKSIZ. This value should be a power of two to give a correct quadtree subdivision but, as will be seen later, could advantageously be given other values. Image A is given the binary result of the uniformity test whilst image Q is given the value of the average of the maximum and minimum values found in the block. This value will not truly reflect the correct grey level of the block but all pixels within the block are within ± THRESH/2 of this value, a fact which will be useful later.

Areas of the image which are considered to be background in the example image of figure 6.7 are shown as white blocks in figure 6.8.

6.8.2 Background Interpolation

Having determined a few known background values it is necessary to fill in unknown areas with a suitable approximation.

The simplest reasonable approximation is a linear interpolation between known values. More complex algorithms, whilst providing a better approximation, would take rather longer to execute.

Interpolation is performed in two stages within routine INTERP of figure 6.10. First interpolations in the X direction take place

```
QUAD:
  { BLKSIZ is the block size, 128 = the whole image
    THRESH is the maximum range of values in a constant block }
       BLKSIZ:=8; THRESH:=35;

  { go to top L.H. corner of each block }
       FOR I FROM 0 BY BLKSIZ TO 128-BLKSIZ DO
        FOR J FROM 0 BY BLKSIZ TO 128-BLKSIZ DO
         X:=I; Y:=J;
         MAX:=MIN:=P0;
  { test all pixels for equality }
         FOR X FROM I TO I+BLKSIZ-1 DO
          FOR Y FROM J TO J+BLKSIZ-1 DO
           MAX:=MAX?>P0;
           MIN:=MIN?<P0
                                     OD
                                    OD;
         C:= MAX-MIN > THRESH;
         R:=IF C THEN 0 ELSE (MAX+MIN)/2 FI;
         FOR X FROM I TO I+BLKSIZ-1 DO
          FOR Y FROM J TO J+BLKSIZ-1 DO
           Q0:=R; A0:=C
                                     OD
                                    OD
                                             OD
                                            OD
      RETURN;
```

Figure 6.9 Quadtree Derived Background Extraction Program

followed by those in the Y direction; routines XINTER and YINTER.

Unfortunately extrapolation can not be performed with as much confidence (bearing in mind the uneven illumination with, possibly, some very odd effects at the edges) and the areas outside the known values can not be so easily found. For simplicity the program simply notes that it can do nothing in such areas; although in practice it would be possible to attempt to do so.

Figure 6.11 shows the interpolated approximation to the background of figure 6.7.

```
INTERP: @XINTER; @YINTER RETURN;

XINTER: EDGEA:=TRUE;
      FOR Y FROM 0 TO 127 DO
       OK:=FALSE;
       FOR X FROM 0 TO 127 DO
        IF A0 THEN
               IF A5=0 THEN OK:=TRUE; DIST:=1;
                            VAL:=Q5; XX:=X
                       ELSE DIST:=DIST+1
               FI
              ELSE
               IF A5 & OK THEN XXX:=X; DIST:=DIST+1;
                               VAL2:=Q0; OK:=FALSE;
                               FOR X FROM XX TO XXX-1 DO
                                Q0:=(VAL2*(X-XX+1)
                                      +VAL*(XXX-X))/DIST;
                                A0:=FALSE
                                                       OD;
                                X:=XXX
        FI     FI
                            OD
                           OD            RETURN;

YINTER: EDGEA:=TRUE;
     FOR X FROM 0 TO 127 DO
      OK:=FALSE;
      FOR Y FROM 0 TO 127 DO
       IF A0 THEN
              IF A3=0 THEN OK:=TRUE; DIST:=1;
                           VAL:=Q3; YY:=Y
                      ELSE DIST:=DIST+1
              FI
             ELSE
              IF A3 & OK THEN YYY:=Y; DIST:=DIST+1;
                              VAL2:=Q0; OK:=FALSE;
                              FOR Y FROM YY TO YYY-1 DO
                               Q0:=(VAL2*(Y-YY+1)
                                     +VAL*(YYY-Y))/DIST;
                               A0:=FALSE
                                                      OD;
                               Y:=YYY
        FI    FI
                           OD
                          OD             RETURN;
```

Figure 6.10 Background Interpolation Program

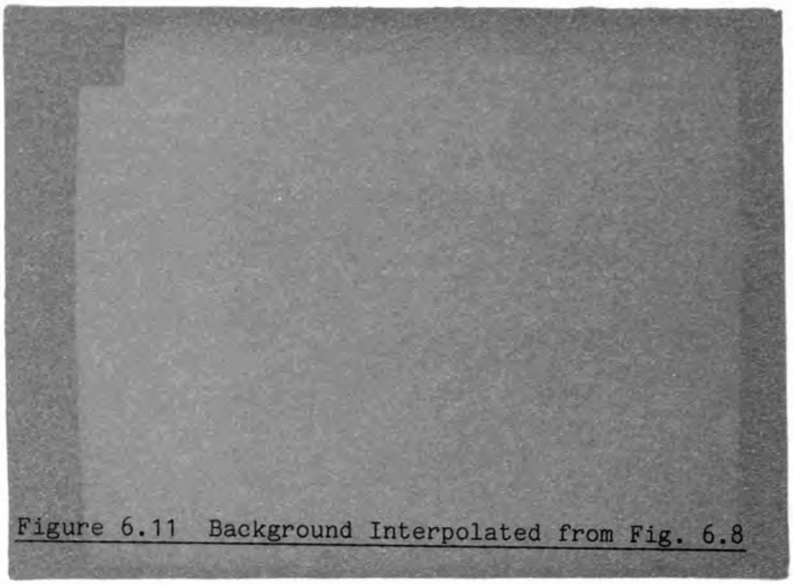


Figure 6.11 Background Interpolated from Fig. 6.8

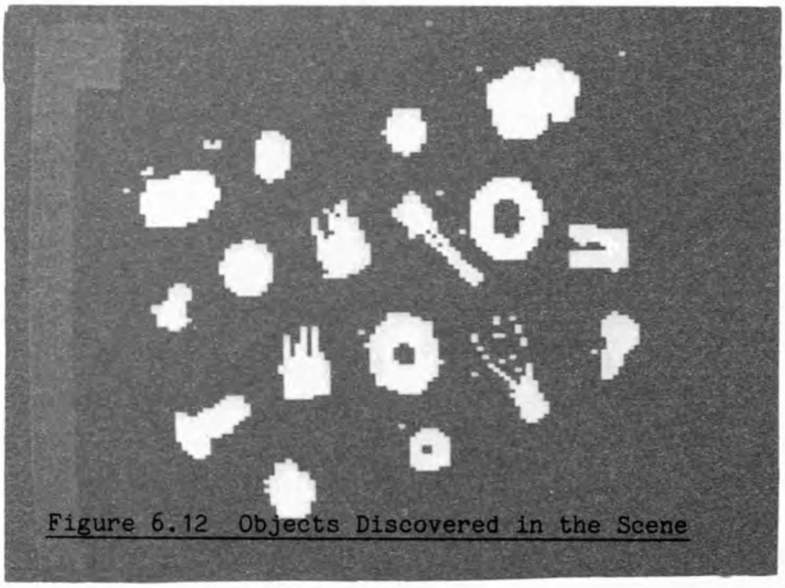
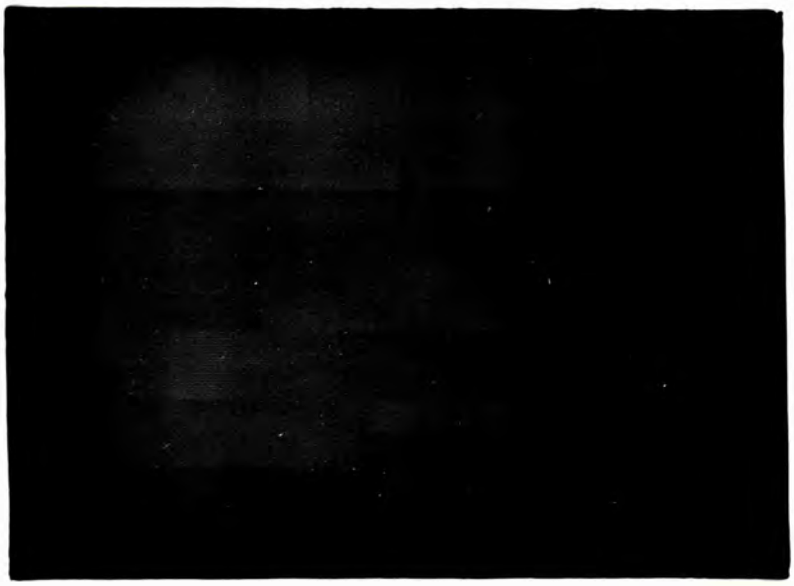
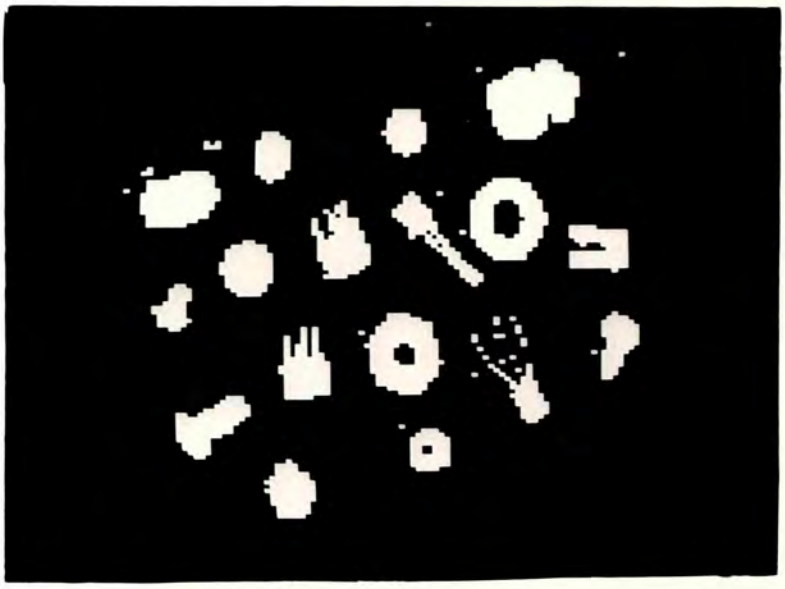


Figure 6.12 Objects Discovered in the Scene





6.8.3 Part Separation

Differences between the scene and the background generated above determine the presence or absence of the parts we are searching for. A small difference suggests that the area in the scene is background while a large one suggests that an object has been found. A suitable limit on the error must be set, above which it is possible to say that this is no longer background.

Here we can make use of the error limit set in the first section of the program and the knowledge that the block values generated there are within half the threshold value of the real background. Hence this value, THRESH/2, is used as the test value in routine PARTS which extracts objects from the scene.

From routine PARTS (figure 6.13) the objects are extracted and seen in figure 6.12, grey shades representing those areas in which an interpolation could not be made and no decision given.

Also seen in figure 6.13 is the calling program FIND to run all the above routines to perform the whole operation of small part detection.

This set of routines in PPL2 compiles and executes in approximately 9 seconds, about half of this being the compilation time.

```
PARTS: [[ R0:=(?+(P0-Q0) > THRESH/2 & A0=0)
             + ((A0#0)&127)                ]] RETURN;

{ Do the whole job }

FIND: @QUAD; @INTERP; @PARTS RETURN;
```

Figure 6.13 Object Location and Display with Overall Program

6.9 UNIFORM THRESHOLD COMPARISON

Commonly used part detection algorithms put a lot of effort into finding a single threshold value to be applied over the whole image. This produces satisfactory results with good lighting but with a scene such as that in figure 6.14 (where the lighting is slightly uneven) proves unusable. A threshold value suitable for most of the objects fails to detect the thin wires from the capacitor on the left (see figure 6.15). A different value which finds the wires also removes most of the rest of the objects, as in figure 6.16. The scheme suggested above, in effect, adjusts the threshold over the image producing the result of figure 6.17 which does not suffer from the problems of the uniform threshold method.

6.10 PARAMETER SELECTION

Two parameters are crucial to the correct operation of this program - the maximum block size and the uniformity test threshold. Both depend on the scenes being viewed.

Maximum block size is determined by the size of the objects to be detected and the amount of background visible between them. The size selected must be sufficiently large that an object cannot totally fill it (or the block would contain a uniform area and be classified as background). It must be small enough to find the areas of background between the objects (if it were too large a block would include part of an object and not be uniform). This places a condition on the scenes that can be tested, namely that the spaces between the objects must be larger than the largest objects; or, only small parts can be correctly extracted.

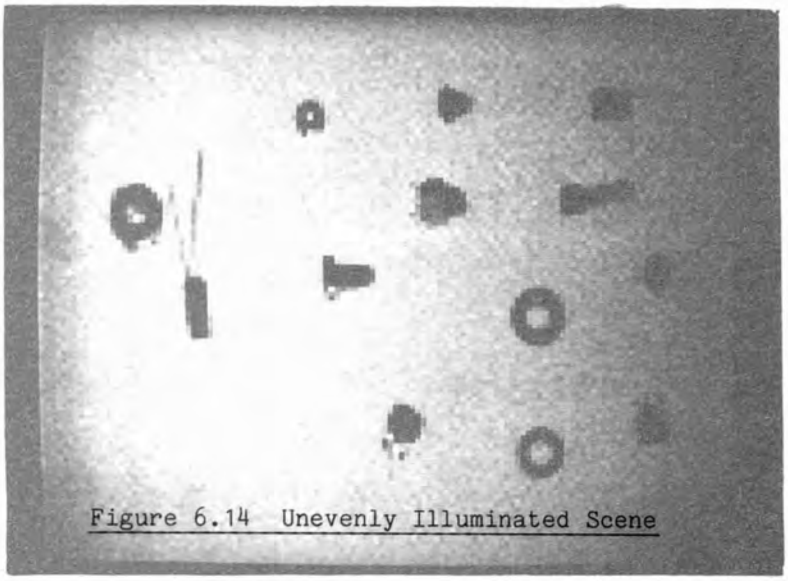


Figure 6.14 Unevenly Illuminated Scene

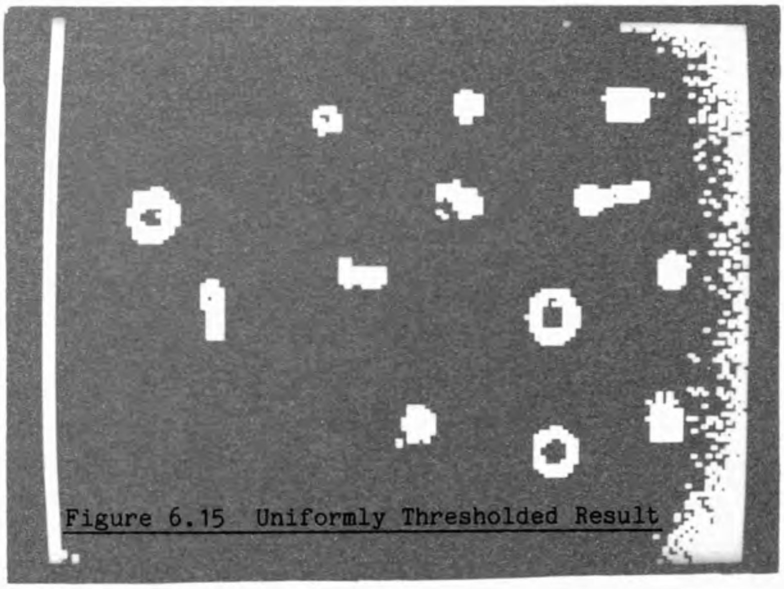
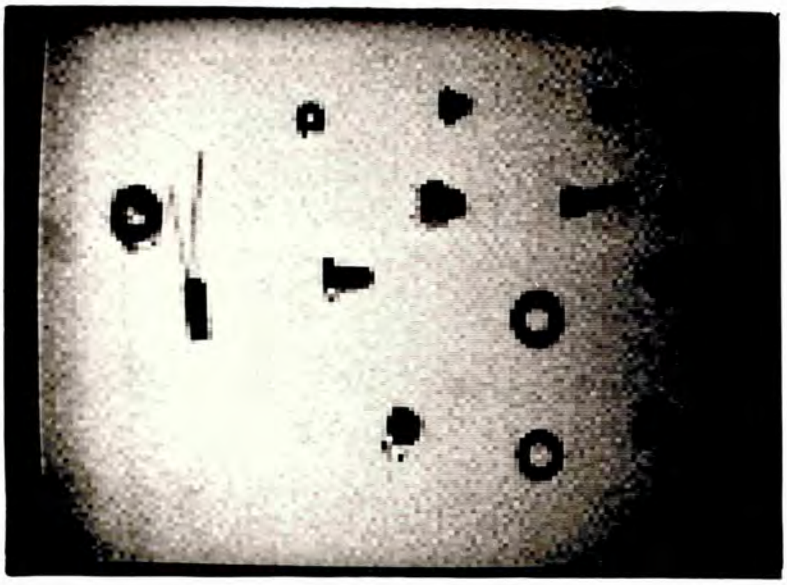
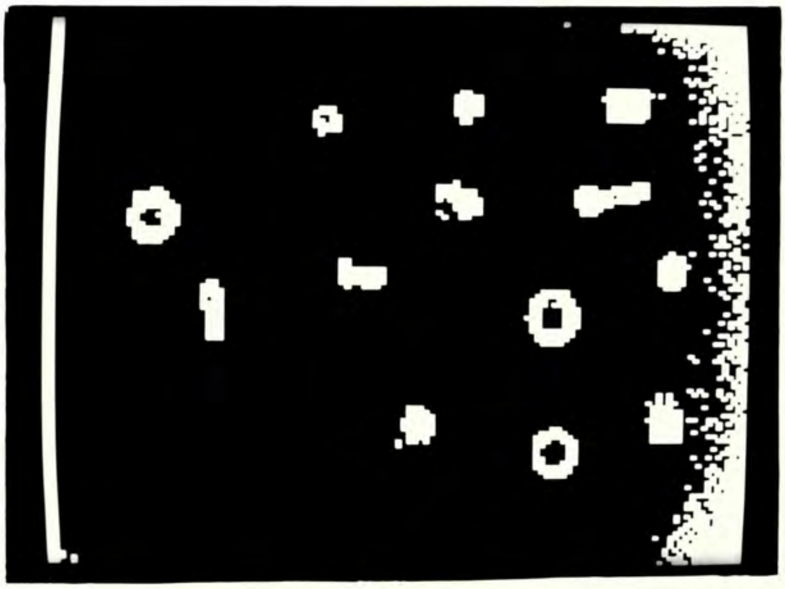


Figure 6.15 Uniformly Thresholded Result





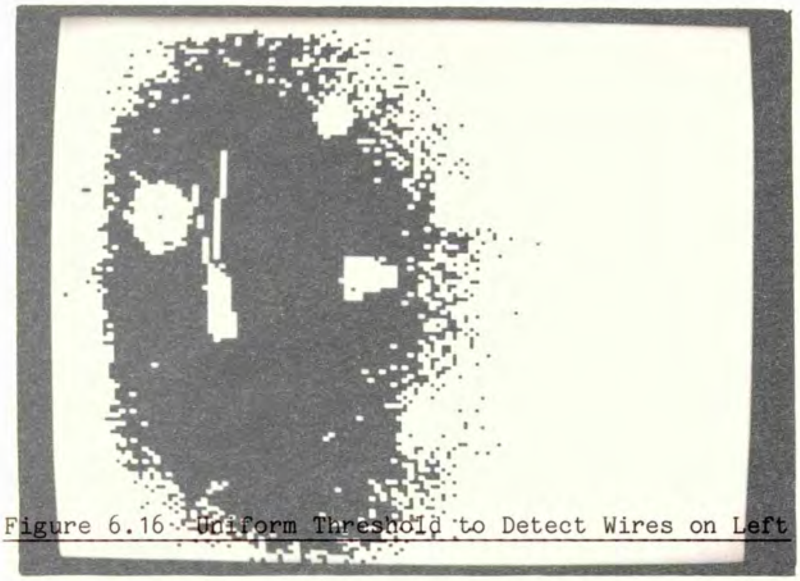


Figure 6.16 Uniform Threshold to Detect Wires on Left

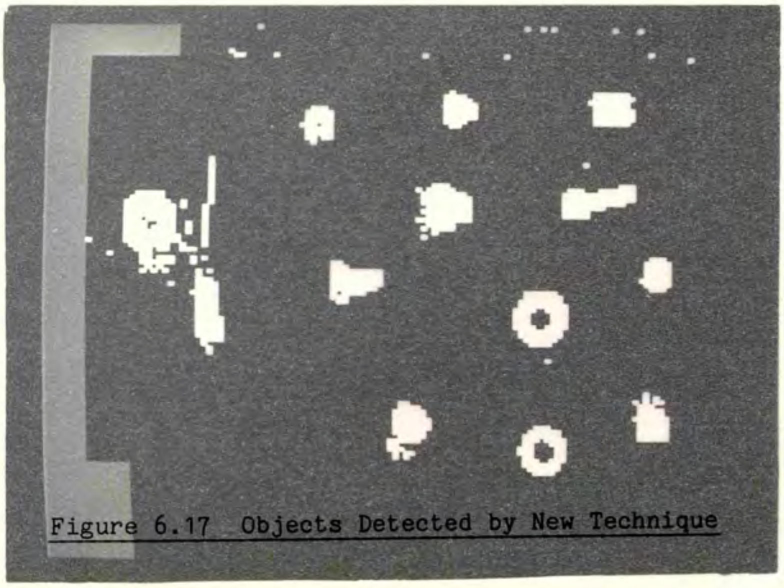
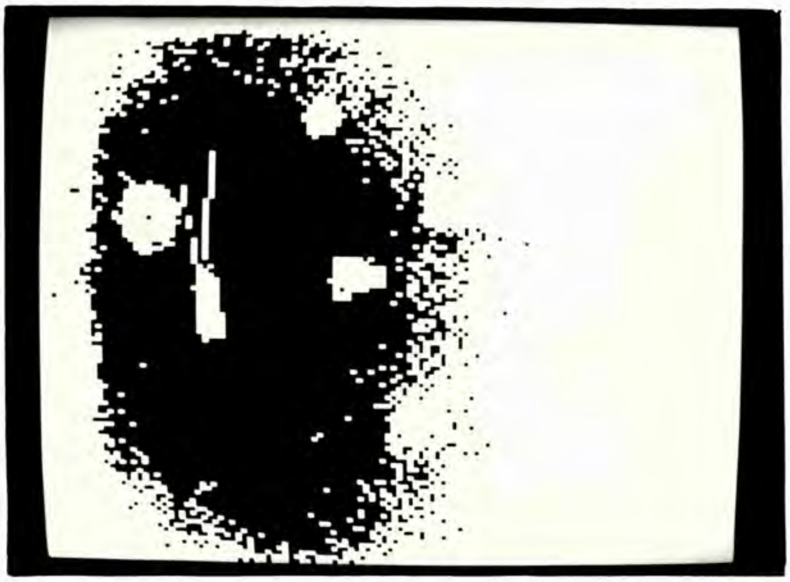
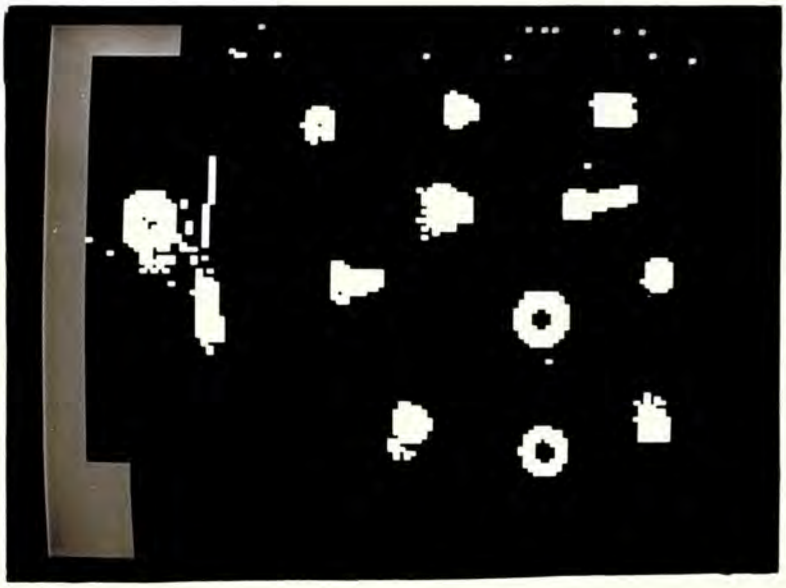


Figure 6.17 Objects Detected by New Technique





Quadtree block sizes all have sides that are a power of two long, this gives only a limited choice of block sizes and hence restricts the scenes that can properly be interpreted. It is not necessary, in the program given, for the blocks to be limited to these sizes. Any block size may be chosen to suit the application and BLKSIZ set accordingly. If there are not an integral number of blocks in the image a border of uncertain area will be given. Allowance for this border can be made in the setting of the field of view of the camera and need not be a problem.

Block positions are fixed by the program and the background regions between objects must coincide with them. Clear spaces need to be twice the length of a block (four times its area) to guarantee satisfactory detection.

Secondly, a suitable threshold for the determination of uniform blocks must be set. This must be such a value that a block covering only background be accepted but one containing any part of an object be rejected. Obviously this depends upon the contrast of the scene and, less obviously on the amount of noise present. The value must be larger than the noise but less than the brightness difference between the objects and the background. It is not necessary to ensure that all background blocks are detected, those missing will be interpolated from those present, so that the threshold may be set low to give errors due to noise rather than to objects.

Noisy environments can benefit from a small amount of noise removal beforehand although this can add a great deal of processing time for relatively little gain. It might be better to change the object location algorithm so that it gives a three-way result, background, object and uncertain, which can be resolved by the recognition system.

6.11 Conclusion

The quadtree representation has been described and shown to lead to a practical method of segmenting an image. There have been difficulties with this approach that would not have been evident in a 'straightforward' method of analysis. On the other hand this method is many times faster than the direct approach. Clearly some more general method is required that retains the speed properties of the quadtree with the ease of application of the direct method.

Ideally tree methods should not only be faster in operation but should implement a more natural rather than a more restricted description of the image.

7. SKELETONISATION

7.1 INTRODUCTION

Skeletons of objects form a compact description of their shape and, providing a suitable skeletonising method is chosen, may be later used to reconstruct the object to any required accuracy [43]. A skeleton shape is characteristic of the object and is suitable for use in a pattern recognition system for identifying the objects (for example [44,45]).

Quadtrees have been suggested for data reduction to speed subsequent processing, and skeletons may be used for a similar purpose. Both allow fast processing on a reduced data set and both can be used to reproduce the original object, either exactly or to a specified, relaxed accuracy. However, there are differences between the two techniques that affect the subsequent processing of the data. Perhaps the most important is the affect of re-positioning the object in the field of view; the skeleton of such an object will be unchanged by such a move but the quadtree may be altered drastically [46,47]. A movement of as little as one pixel will cause a completely new quadtree to be formed and its usefulness in a recognition system consequently reduced. To overcome this problem an algorithm for producing a skeleton from a quadtree has been described [48]. A second difference that is implied by the two techniques is the architecture of the processor that is best suited (fastest, cheapest, etc.) to the technique; this is discussed further in section 8.3.

7.2 SKELETON FINDING ALGORITHMS

Many skeleton finding algorithms have been proposed over the past twenty or so years [44,49,50,51,52] but the approach to their design has been rather ad hoc and each has some property that prevents it being of universal value. The problems of these algorithms have been studied in [1] and a new method proposed that is rigorously derived. This method is guaranteed to produce skeletons to a known accuracy and capable of being used to reconstruct the original image if so desired.

Common to all skeletonisation algorithms is the need to remove the outer layer of pixels from an object to produce a slimmer outline; the procedure is repeated until a single width line is left. Parallel implementations of the skeletonising algorithm are faced with a problem when the object contains a limb of two pixels width. Consider, for example, figure 7.1 which shows a portion of an object with a section 2 pixels wide marked as 'A' and 'B', '+'s denote other points on the object.

```
+ + + + A + + +
+ + + + B + + +
```

Figure 7.1 A Two Pixel Wide Object

Using a 3 x 3 window to process the image produces the problem that for the processor centered at 'A' the object appears to need slimming and point 'A' is removed. At the same time the processor at 'B' comes to the same conclusion and removes 'B' to produce a break in the skeleton. The problem occurs because of a lack of information on the part of each processor, a 3 x 3 window is not sufficient to detect that the object is not more than 2 pixels wide and an error occurs.

Two possible solutions to this problem can be found, the first is to use a window of more than 3 x 3 pixels. The second, which is the more usual, is to divide the slimming procedure into four sections each removing points from only one direction at a time, e.g. from the North, South, West and East sides. The latter solution requires four processes to remove one layer of cells from around the image where the former would need only one; however the 5 x 5 window would take longer to execute for its one application and it is not obvious which would be the faster overall.

In order to investigate the differences in method the Davies-Plummer algorithm of [43] has been implemented in PPL2 with a 3 x 3 and a 5 x 5 window so that the relative performances of each could be tested.

7.3 PPL2 IMPLEMENTATION OF THE DAVIES-PLUMMER ALGORITHM

The algorithm described by Davies and Plummer [43] can be divided into five sub-tasks each of which can be implemented separately -

1. Propagate the distance function,
2. Mark the local maxima,
3. Slim to a connected shape,
4. Thin to a unit width skeleton and
5. Clean off the unit length noise spurs.

It is during step 3 that a 5 x 5 window is employed and the interactive facilities provided by the PPL2 system were used to full advantage in generating the algorithm and checking the results obtained. The object shown in figure 7.4 will be used to demonstrate the workings of this implementation.

7.3.1 Propagation of the Distance Function

Either sequential or parallel algorithms may be used to propagate the distance function: the sequential algorithm is given in figure 7.2 and requires a pair of scans, one forward and one reverse. The parallel version is given in figure 7.3 and requires a number of applications depending on the data. If the largest object is n pixels wide at its widest the propagation function must be applied n/2 times before completion. For a 128 x 128 pixel image the maximum number of

```
PROP: [[ P0:=0 ]]; EDGEP:=0;  { initially distance function is 0 }
                              { A contains the binary image      }
      [[+ IF A0 THEN P0:= (P2 ?< P3 ?< P4 ?< P5)  +1  FI +]];
      [[- IF A0 THEN P0:= ((P1 ?< P2 ?< P3 ?< P4 ?< P5 ?<
                            P6 ?< P7 ?< P8)  +1) ?< P0    FI -]]
      RETURN;
```

Figure 7.2 Distance Function Propagation - Sequential Algorithm

```
PROP: [[ P0:=0 ]]; EDGEP:=0;  { initially distance function is 0 }
                              { A contains the binary image      }
      TO 64                   { 128/2 times is the maximum       }
       DO
        [[ Q0:=IF A0          { process into Q, for parallel     }
                THEN
                     (P0 ?< P1 ?< P2 ?< P3 ?< P4 ?<
                                   P5 ?< P6 ?< P7 ?< P8) + 1
                ELSE
                     P0
               FI
        ]];
        [[ P0:=Q0 ]]          { return to P for next run         }
       OD
      RETURN;
```

Figure 7.3 Distance Function Propagation - Parallel Algorithm

applications needed is 64; this is the number performed in the program of figure 7.3. It would be possible to reduce this number by testing at each application if another is required and thus stopping at less than 64 applications. Each application would take a little longer, however, so the advantages may not be as large as expected. The result obtained is shown in figure 7.5.

7.3.2 Marking of Local Maxima

Little effort is required to discover which points correspond to the local maxima of the distance function, such points are characterized by having a value at least as large as all their

```
                              X X
                              X X X
                            X X X X
                            X X X X
                          X X X X X X
                          X X X X X X
                          X X X X X X
                        X X X X X X X X
                        X X X X X X X X
                        X X X X X X X X
                      X X X X X X X X X X
                      X X X X X X X X X X
                      X X X X X X X X X X
                    X X X X X X X X X X X X
                    X X X X X X X X X X X X
                  X X X X X X X X X X X X X
                  X X X X X X X X X X X X X
                  X X X X X X   X X X X X X X
                X X X X X X X   X X X X X X X
                X X X X X X X     X X X X X X
                X X X X X X       X X X X X X X
              X X X X X X X       X X X X X X X
              X X X X X X X       X X X X X X X
              X X X X X X           X X X X X X
            X X X X X X X           X X X X X X X
            X X X X X X X           X X X X X X X
            X X X X X X X X X   X X X X X X X X X
          X X X X X X X X X X X X X X X X X X X X
          X X X X X X X X X X X X X X X X X X X X X
          X X X X X X X X X X X X X X X X X X X X X
        X X X X X X X X X X X X X X X X X X X X X X
        X X X X X X X X X X X X X X X X X X X X X X X
        X X X X X X X X X X X X X X X X X X X X X X X
      X X X X X X X X X X X X X X X X X X X X X X X X
      X X X X X X X X X X X X X X X   X X X X X X X X
      X X X X X X X                       X X X X X X X
    X X X X X X X X                       X X X X X X X
    X X X X X X X                           X X X X X X
    X X X X X X X                           X X X X X X X
    X X X X X X X                           X X X X X X X
  X X X X X X X                               X X X X X X
  X X X X X X X                               X X X X X X
  X X X X X X X                               X X X X X X X
X X X X X X X                                   X X X X X X
X X X X X X X                                   X X X X X X
X X X X X X X                                   X X X X X X X
X X X X X X X                                     X X X X X X
      X X X                                       X X X X X X
                                                    X X X X
```

Figure 7.4 Original Shape

```
                              1 1
                              1 1 1
                            1 1 2 1
                            1 2 2 1
                          1 1 2 2 1 1
                          1 2 2 2 2 1
                          1 2 3 3 2 1
                        1 1 2 3 3 2 1 1
                        1 2 2 3 3 2 2 1
                        1 2 3 3 3 3 2 1
                      1 1 2 3 4 4 3 2 1 1
                      1 2 2 3 4 4 3 2 2 1
                      1 2 3 3 4 4 3 3 2 1
                    1 1 2 3 4 4 4 4 3 2 1 1
                    1 2 2 3 3 3 3 3 3 2 2 1
                  1 1 2 3 2 2 2 2 2 3 3 2 1
                  1 2 2 3 2 1 1 1 2 3 3 2 1
                  1 2 3 3 2 1   1 2 2 3 2 1 1
                1 1 2 3 2 2 1   1 1 2 3 2 2 1
                1 2 2 3 2 1 1     1 2 3 3 2 1
                1 2 3 3 2 1       1 2 3 3 2 1 1
              1 1 2 3 2 2 1       1 2 2 3 2 2 1
              1 2 2 3 2 1 1       1 1 2 3 3 2 1
              1 2 3 3 2 1           1 2 3 3 2 1
            1 1 2 3 3 2 1           1 2 3 3 2 1 1
            1 2 2 3 3 2 1           1 2 3 3 2 2 1
            1 2 3 3 3 2 1 1 1   1 1 1 2 3 3 3 2 1
          1 1 2 3 4 3 2 2 2 1 1 1 2 2 2 3 4 3 2 1
          1 2 2 3 4 3 3 3 2 2 2 2 2 3 3 3 4 3 2 1 1
          1 2 3 3 4 4 4 3 3 3 3 3 3 3 4 4 4 3 2 2 1
        1 1 2 3 4 4 5 4 4 4 4 4 4 4 4 4 4 4 3 3 2 1
        1 2 2 3 4 4 4 4 4 4 4 3 3 3 3 3 3 3 4 3 2 1 1
        1 2 3 3 3 3 3 3 3 3 3 3 2 2 2 2 2 3 3 3 2 2 1
      1 1 2 3 3 2 2 2 2 2 2 2 2 2 1 1 1 2 2 2 3 3 2 1
      1 2 2 3 3 2 1 1 1 1 1 1 1 1 1   1 1 1 2 3 3 2 1
      1 2 3 3 2 2 1                       1 2 2 3 2 1 1
    1 1 2 3 3 2 1 1                       1 1 2 3 2 2 1
    1 2 2 3 3 2 1                           1 2 3 3 2 1
    1 2 3 3 2 2 1                           1 2 2 3 2 1 1
    1 2 3 3 2 1 1                           1 1 2 3 2 2 1
  1 1 2 3 3 2 1                               1 2 3 3 2 1
  1 2 2 3 2 2 1                               1 2 2 3 2 1
  1 2 3 3 2 1 1                               1 1 2 3 2 1 1
1 1 2 3 3 2 1                                   1 2 3 2 2 1
1 2 2 3 3 2 1                                   1 2 2 3 2 1
1 2 2 2 2 2 1                                   1 1 2 3 2 1 1
1 1 1 1 2 1 1                                     1 2 2 2 2 1
      1 1 1                                       1 1 2 2 1 1
                                                    1 1 1 1
```

Figure 7.5 Propagated Distance Function on figure 7.4

neighbours. A program is shown in figure 7.6 and is equally applicable to a sequential or a parallel machine. A display of the points marked, with their values, is given in figure 7.7; the original image points are shown with dots.

```
MARK: [[ B0:=       { B set if the point is a local max ... }
             A0 &   { ... on the image                      }
             P0 >= ( P1 ?> P2 ?> P3 ?> P4 ?>
                         P5 ?> P6 ?> P7 ?> P8 )  ]]
      RETURN;
```

Figure 7.6 Mark the Local Maxima

7.3.3 Slimming to a Connected Shape

Slimming is achieved by removing, symmetrically, the outer layers of the shape to leave a connected skeleton of 1 or 2 pixels width. The Davies-Plummer algorithm requires that the local maxima marked in the previous step are not removed at this stage. This process has been programmed for 3 x 3 and 5 x 5 windows.

7.3.3.1 3 x 3 Window

As shown in figures 7.8a and 7.8b the program for a 3 x 3 window is quite simple. In each of the four directions of erosion the specific conditions attached to the removal or otherwise of a point are explicitly stated. First, a check is made that the centre point of the window is on an image and that it has not been marked. This is followed by a check that the point is on the correct edge of the image and then that its removal would not break the skeleton. For example, in the case of possible northerly point removal (the first in the program) the centre point must remain if -

```
                              1 1
                              . . .
                            . . 2 .
                            . 2 2 .
                          . . 2 2 . .
                          . . . . . .
                          . . 3 3 . .
                        . . . 3 3 . . .
                        . . . 3 3 . . .
                        . . . . . . . .
                      . . . . 4 4 . . . .
                      . . . . 4 4 . . . .
                      . . . . 4 4 . . . .
                    . . . . 4 4 4 4 . . . .
                    . . . . . . . . . . . .
                  . . . 3 . . . . . 3 3 . .
                  . . . 3 . . . . . 3 3 . .
                  . . 3 3 . .   . . . 3 . . .
                . . . 3 . . .   . . . 3 . . .
                . . . 3 . . .     . . 3 3 . .
                . . 3 3 . .       . . 3 3 . . .
              . . . 3 . . .       . . . 3 . . .
              . . . 3 . . .       . . . 3 3 . .
              . . 3 3 . .           . . 3 3 . .
            . . . 3 3 . .           . . 3 3 . . .
            . . . 3 3 . .           . . 3 3 . . .
            . . . . . . . . .   . . . . . . . . .
          . . . . 4 . . . . . . . . . . . 4 . . .
          . . . . 4 . . . . . . . . . . . 4 . . . .
          . . . . . . . . . . . . . . 4 4 4 . . . .
        . . . . 4 . 5 . 4 4 4 4 4 4 4 4 4 4 . . . .
        . . . . 4 . . . 4 4 4 . . . . . . . 4 . . . .
        . . 3 . . . . . . . . . . . . . . . . . . . .
      . . . 3 3 . . . . . . . . . . . . . . . 3 3 . .
      . . . 3 3 . . . . . . . . . .   . . . . 3 3 . .
      . . 3 3 . . .                       . . . 3 . . .
    . . . 3 3 . . .                       . . . 3 . . .
    . . . 3 3 . .                           . . 3 3 . .
    . . 3 3 . . .                           . . . 3 . . .
    . . 3 3 . . .                           . . . 3 . . .
  . . . 3 3 . .                               . . 3 3 . .
  . . . 3 . . .                               . . . 3 . .
  . . 3 3 . . .                               . . . 3 . . .
. . . 3 3 . .                                   . . 3 . . .
. 2 . 3 3 . .                                   . . . 3 . .
. 2 . . . . .                                   . . . 3 . . .
. . . . 2 . .                                     . . . . 2 .
      . . .                                       . . 2 2 . .
                                                    . . . .
```

Figure 7.7 Local Maxima of the Distance Function
(Original shape marked with dots)

It is not on a northern edge OR it would break the skeleton

```
i.e.     ~A7 ! A3          !  ~A1 & A2   !  ~A5 & A4

           X 0 X              1 X X         X X 1
           X + X              0 + X         X + 0
           X 1 X              X X X         X X X
```

Similar conditions apply for the southern, western and eastern edges which may be illustrated by rotating the above diagrams by multiples of 90 degrees. This is the parallel thinning algorithm employed by Davies and Plummer [43].

The first program of figure 7.8 shows a separated parallel implementation and the second combines processing with the restoring move to execute in about 60% of the time required by the first.

7.3.3.2 5 x 5 Window

Using a window larger than 3 x 3 pixels it is possible to determine whether the shape is less than three cells wide and hence avoid simultaneously removing opposite edges to disconnect the shape. It is possible to remove the edge point of a two cell wide shape if it is known that the opposite edge will not be removed, i.e. it has been marked.

At first sight the required algorithm is extremely simple and obtained as a small extension of the 3 x 3 method above. The centre point of the window may be removed if (in the case of a northerly point) -

```
SLIM:               { A contains the binary image              }
    TO 64           { B contains the local maxima points       }
     DO             { Perform 128/2 times (max needed)         }
      [[ CO:=IF ~BO&AO THEN                                { N }
                ~A7 ! A3 ! ~A1&A2 ! ~A5&A4 ELSE AO FI ]];
      [[ AO:=CO ]];
      [[ CO:=IF ~BO&AO THEN                                { S }
                ~A3 ! A7 ! ~A5&A6 ! ~A1&A8 ELSE AO FI ]];
      [[ AO:=CO ]];
      [[ CO:=IF ~BO&AO THEN                                { E }
                ~A1 ! A5 ! ~A3&A4 ! ~A7&A6 ELSE AO FI ]];
      [[ AO:=CO ]];
      [[ CO:=IF ~BO&AO THEN                                { W }
                ~A5 ! A1 ! ~A7&A8 ! ~A3&A2 ELSE AO FI ]];
      [[ AO:=CO ]]
     OD
    RETURN;         { A now contains the slimmed binary image  }
```

Figure 7.8a NSEW Slimming Program

```
SLIM:
    TO 64
     DO
      [[ CO:=IF ~BO&AO THEN                                { N }
                ~A7 ! A3 ! ~A1&A2 ! ~A5&A4 ELSE AO FI ]];
      [[ AO:=IF ~BO&CO THEN                                { S }
                ~C3 ! C7 ! ~C5&C6 ! ~C1&C8 ELSE CO FI ]];
      [[ CO:=IF ~BO&AO THEN                                { E }
                ~A1 ! A5 ! ~A3&A4 ! ~A7&A6 ELSE AO FI ]];
      [[ AO:=IF ~BO&CO THEN                                { W }
                ~C5 ! C1 ! ~C7&C8 ! ~C3&C2 ELSE CO FI ]]
     OD
    RETURN;
```

Figure 7.8b Faster NSEW Slimming Program

```
                              1 1
                              . 1 .
                            . . 2 .
                            . 2 2 .
                          . . 2 2 . .
                          . . . 2 . .
                          . . 3 3 . .
                        . . . 3 3 . . .
                        . . . 3 3 . . .
                        . . . . 3 . . .
                      . . . . 4 4 . . . .
                      . . . . 4 4 . . . .
                      . . . . 4 4 . . . .
                    . . . . 4 4 4 4 . . . .
                    . . . 3 . . . . 3 . . .
                  . . . 3 . . . . . 3 3 . .
                  . . . 3 . . . . . 3 3 . .
                  . . 3 3 . .   . . . 3 . . .
                . . . 3 . . .   . . . 3 . . .
                . . . 3 . . .     . . 3 3 . .
                . . 3 3 . .       . . 3 3 . . .
              . . . 3 . . .       . . . 3 . . .
              . . . 3 . . .       . . . 3 3 . .
              . . 3 3 . .           . . 3 3 . .
            . . . 3 3 . .           . . 3 3 . . .
            . . . 3 3 . .           . . 3 3 . . .
            . . . . 3 . . . .   . . . . . 3 . . .
          . . . . 4 . . . . . . . . . . . 4 . . .
          . . . . 4 . . . . . . . . . . . 4 . . . .
          . . . . 4 . . . . . . . . . 4 4 4 . . . .
        . . . . 4 . 5 4 4 4 4 4 4 4 4 4 4 4 . . . .
        . . . . 4 . . . 4 4 4 . . . . . . . 4 . . . . .
        . . 3 3 . . . . . . . . . . . . . . . 3 . . .
      . . . 3 3 . . . . . . . . . . . . . . . 3 3 . .
      . . . 3 3 . . . . . . . . . .   . . . . 3 3 . .
      . . 3 3 . . .                       . . . 3 . . .
    . . . 3 3 . . .                       . . . 3 . . .
    . . . 3 3 . .                           . . 3 3 . .
    . . 3 3 . . .                           . . . 3 . . .
    . . 3 3 . . .                           . . . 3 . . .
  . . . 3 3 . .                               . . 3 3 . .
  . . . 3 . . .                               . . . 3 . .
  . . 3 3 . . .                               . . . 3 . . .
. . . 3 3 . .                                   . . 3 . . .
. 2 2 3 3 . .                                   . . . 3 . .
. 2 . . 2 . .                                   . . . 3 . . .
. . . . 2 . .                                     . . . 2 2 .
      . . .                                         . . 2 2 . .
                                                    . . . .
```

Figure 7.9 Slimmed Shape - 3 x 3 Window

```
· It is on the northern edge AND removal would not break the skeleton
  AND ( the shape is >= 3 deep
        OR the shape is >=2 deep
           AND marked           )

i.e. ~A3 & A7 & (A22 ! B7) &     ~(  ~A5 & A4     !   ~A1 & A2  )

        X X X X X                    X X X X X        X X X X X
        X X 0 X X                    X 1 X X X        X X X 1 X
        X X + X X                    X 0 + X X        X X + 0 X
        X X 1 X X                    X X X X X        X X X X X
        X X 1 X X                    X X X X X        X X X X X
```

While this condition provides the major requirement of the slimming algorithm, that North and South points on opposite sides of a shape less than 3 cells wide are not simultaneously removed, it is not sufficient for a complete algorithm. Further restraints on the removal of points have to be added to fully prevent the breaking of a skeleton.

Consider the portion of a shape shown below; zeros represent cells which are not part of the shape, other characters cells belonging to it. The two cells marked 'S' and 'N' are both found to be removable during the same pass; 'S' as a southerly point and 'N' as a northerly point, by the algorithm proposed above.

```
1 1 1 1 0 0 0 0
1 1 1 1 0 0 0 0
1 1 1 S N 1 1 1
0 0 0 0 1 1 1 1
0 0 0 0 1 1 1 1
```

However, if their simultaneous removal is attempted the skeleton is broken. The problem does not arise in the case of the 3 x 3 window as the two points are tested on different passes so that one of them is removed, causing the shape to change and the other is then not allowed to disappear.

Prevention of this case is fairly simple but requires a considerable computational effort relative to the detection algorithm above. It is necessary to detect the existence of adjacent cells which are edges in the opposite direction to that at the centre point; that is the algorithm must prevent removal of a point

X N S X   or   X S N X

While removal is allowed for -

X S S X   or   X N N X

The presence of an adjacent edge point in an orthogonal direction (E or W) does not matter. The scheme suggested here prevents the removal of either point where one of them may have been safely removed; the program thus errs on the side of safety.

The discussion above referred to the removal of northerly and southerly points from a shape; the same argument may be applied to the westerly and easterly points by rotating all diagrams through 90 degrees.

Parallel removal of points in the North-South and West-East directions causes a further interaction that can also break the skeleton. An example of the problem is shown below; zeros represent cells not on the shape, while ones, ´W´s, ´S´s and ´*´s represent cells on the shape.

```
0 0 W 1 1
0 0 W 1 1
0 0 * S S
0 0 1 0 0
1 1 0 0 0
```

Cell ´*´ on the corner is both a Westerly and Southerly point and is thus clearly suitable for removal. A problem occurs if this point AND that next to it on its right (a Southerly point) are both removed, the skeleton is broken. It is possible to detect this specific condition and to prevent the removal of points such as that marked ´*´ above by looking for patterns such as

$$\begin{matrix}0&X&X\\1&+&X\\0&X&X\end{matrix} \quad \text{or} \quad \begin{matrix}X&X&0\\X&+&1\\X&X&0\end{matrix} \quad \text{or} \quad \begin{matrix}0&1&0\\X&+&X\\X&X&X\end{matrix} \quad \text{or} \quad \begin{matrix}X&X&X\\X&+&X\\0&1&0\end{matrix}$$

to inhibit removal of the centre point ('+').

Implementation of this algorithm is simple but fairly long in PPL2, as is shown in figure 7.10. It is possible to speed up this program by algebraically simplifying the equations, at the expense of clarity; the clearer version is given here. The slimmed shape resulting from this algorithm is shown in figure 7.11. Comparison with the shape produced by the 3 x 3 window program (see figure 7.9) reveals some minor differences caused by the sequential erosion of the edge of the shape by the 3 x 3 window technique. Both skeletons are, however, acceptable under the requirements of the Davies-Plummer algorithm.

7.3.3.3 Performance Comparison

In order to reduce the execution times of the algorithms to a minimum they were both modified to stop after completing a scan (5 x 5) or set of scans (3 x 3); the resultant programs are reproduced in Appendix A. The programs were then executed for a number of images, each image being slimmed by each of the algorithms to enable a comparison to be made. Figure 7.12 tabulates the resulting execution times for these trials; the images used are shown in figure 7.20 at the end of the chapter. Since PPL2 compiles image scan operations each time they are required this compilation time has to be removed from the total elapsed time to give the execution time of the algorithms.

Additional scans by the 5 x 5 window algorithm over the number required by the 3 x 3 algorithm are caused by the caution in point removal built in to the 5 x 5 slim. Despite the extra scans required and apparent extra complexity of the programs it is seen that the larger window executes in less time than the smaller. The difference

```
SLIM: EDGEA::=0;            { shape surrounded by nothing           }
      TO 64                 { enough times to be sure of finishing  }
       DO
        [[ C0:=             { results --> C for parallel            }
            IF ~B0 & A0     { remove only if not marked object point}
             THEN
              NS:= (A4!~A5!A6) & (A2!~A1!A8);  { thin connection }

              N:= ~A3&A7&(A22!B7) & ~(~A5&A4 ! ~A1&A2);{ N point }
              NW:= ~A6&A4 & (A18!~A19);            { S point on W }
              NE:= ~A8&A2 & (A10!~A9);             { S point on E }

              S:= ~A7&A3&(A14!B3) & ~(~A5&A6 ! ~A1&A8);{ S point }
              SW:= ~A4&A6 & (A18!~A17);            { N point on W }
              SE:= ~A2&A8 & (A10!~A11);            { N point on E }

              WE:= (A2!~A3!A4) & (A6!~A7!A8);  { thin connection }

              W:= ~A5&A1&(A10!B1) & ~(~A3&A4 ! ~A7&A6);{ W point }
              WN:= ~A2&A4 & (A14!~A13);            { E point on N }
              WS:= ~A8&A6 & (A22!~A23);            { E point on S }

              E:= ~A1&A5&(A18!B5) & ~(~A3&A2 ! ~A7&A8);{ E point }
              EN:= ~A4&A2 & (A14!~A15);            { W point on N }
              ES:= ~A6&A8 & (A22!~A21);            { W point on S }
                                           { remove it if we can }
              ~(   ( ~(NW ! NE) & N   !  ~(SW ! SE) & S ) & NS
                !  ( ~(WN ! WS) & W   !  ~(EN ! ES) & E ) & WE
                 )
             ELSE
              A0         { background and marked points unchanged }
            FI
       ]];
      [[ A0:=C0 ]]             { results back --> A for parallel }
     OD
     RETURN;
```

Figure 7.10 5 x 5 Window Parallel Slim

```
                              1 1
                              . 1 .
                            . . 2 .
                            . 2 2 .
                          . . 2 2 . .
                          . . 2 2 . .
                          . . 3 3 . .
                        . . . 3 3 . . .
                        . . . 3 3 . . .
                        . . . 3 3 . . .
                      . . . . 4 4 . . . .
                      . . . . 4 4 . . . .
                      . . . . 4 4 . . . .
                    . . . . 4 4 4 4 . . . .
                    . . . 3 . . . 3 . . . .
                  . . . 3 2 . . . . 3 3 . .
                  . . . 3 . . . . . 3 3 . .
                  . . 3 3 . .   . . . 3 . . .
                . . . 3 . . .   . . . 3 . . .
                . . . 3 . . .     . . 3 3 . .
                . . 3 3 . .       . . 3 3 . . .
              . . . 3 . . .       . . . 3 . . .
              . . . 3 . . .       . . . 3 3 . .
              . . 3 3 . .           . . 3 3 . .
            . . . 3 3 . .           . . 3 3 . . .
            . . . 3 3 . .           . . 3 3 . . .
            . . . 3 . . . . .   . . . . . 3 . . .
          . . . . 4 . . . . . . . . . . . 4 . . .
          . . . . 4 3 . . . . . . . . . . 4 . . . .
          . . . . . 4 4 . . . . . . . 4 4 4 . . . .
        . . . . 4 4 5 . 4 4 4 4 4 4 4 4 4 4 . . . .
        . . . . 4 . . . 4 4 4 . . . . . . 3 4 . . . .
        . . 3 3 3 . . . . . . . . . . . . . 3 . . . .
      . . . 3 3 . . . . . . . . . . . . . . . 3 3 . .
      . . . 3 3 . . . . . . . . . .   . . . . 3 3 . .
      . . 3 3 . . .                       . . . 3 . . .
    . . . 3 3 . . .                       . . . 3 . . .
    . . . 3 3 . .                           . . 3 3 . .
    . . 3 3 . . .                           . . . 3 . . .
    . . 3 3 . . .                           . . . 3 . . .
  . . . 3 3 . .                               . . 3 3 . .
  . . . 3 . . .                               . . . 3 . .
  . . 3 3 . . .                               . . . 3 . . .
. . . 3 3 . .                                   . . 3 . . .
. 2 2 3 3 . .                                   . . . 3 . .
. 2 . . 2 . .                                   . . . 3 . . .
. . . . 2 . .                                     . . . 2 2 .
      . . .                                       . . 2 2 . .
                                                    . . . .
```

Figure 7.11 Slimmed Shape - 5 x 5 Window

	3 x 3 Window				5 x 5 Window				Ratio
I	Ns	Tt (s)	Tc (s)	Te (s)	Ns	Tt (s)	Tc (s)	Te (s)	Te(3x3)/ Te(5x5)
a	11	38.50	7.26	31.24	19	48.20	20.33	27.87	1.12
b	6	20.31	3.96	16.35	7	15.53	7.49	8.04	2.03
c	5	17.02	3.30	13.72	9	20.12	9.63	10.49	1.31
d	11	38.05	7.26	30.79	19	45.79	20.33	25.46	1.21
e	11	38.46	7.26	31.20	19	47.97	20.33	27.64	1.13
f	7	24.30	4.62	19.68	9	21.92	9.63	12.29	1.60
g	6	20.66	3.96	16.70	11	25.36	11.77	13.59	1.23
h	7	24.73	4.62	20.11	11	28.39	11.77	16.62	1.21
i	8	27.09	5.28	21.81	8	17.31	8.56	8.75	2.49
j	6	20.33	3.96	16.37	6	13.05	6.42	6.63	2.47
k	22	80.35	14.52	65.83	31	96.26	33.17	63.09	1.04
l	4	13.63	2.64	10.99	6	13.28	6.42	6.86	1.60
	Compile time per scan 0.66s				Compile time per scan 1.07s				

I: Image, see figure 7.20.

Ns: Number of scans.

Tt: Total time taken.

Tc: Compile time = Ns * Compile time per scan.

Te: Execution time = Ttot - Tcom.

Figure 7.12 Performance Comparison of Slimming Algorithms

is highly data dependant and hence the ratio of execution times is seen to vary from only just greater than 1 to nearly 2.5.

Worthwhile speed increases are possible by using the larger window for slimming of the shapes. The improvement in execution speed is not, however, so great that this technique should be used at all costs. It is quite likely that a processor that lacks the image access hardware built for this project and described in chapter 5 would lose the speed advantage of a larger window in the time required to access the increased number of image points.

Different skeletons are produced by the two algorithms and while both fulfil the accuracy requirements of the Davies-Plummer algorithm they can differ somewhat in detail. Figure 7.13 shows two skeletons of part of a key which has been slimmed by both the methods described above; the original being indicated by '.'s and the skeleton by 'X's It can be seen that the second upward spur from the left is rather different in the two cases. The cause is the improved symmetry of the 5 x 5 window algorithm as opposed to the bias introduced by the sequential choice of directions of erosion for the 3 x 3 window algorithm.

7.3.4 Thinning to a Skeleton

Prior to this step the original shape could be recreated from the slimmed shape and the corresponding values of the distance function. However, the shape is not a unit width skeleton and must be further thinned. At no point on the slimmed shape is the width more than two pixels in the thinnest direction so that it is necessary to remove no more than one pixel from the edge to produce the skeleton.

Production of a unit width skeleton from the slimmed shape

```
...........
...........
...........
...........
...........
.....X.....
.....X......    ...
.....X...........X.
.....X...........X.
.....X..........X.......
.....X.........X.........     XX               ...
.....XX.......X............  ..X....       .....X....
......X......X.................X.....     ......X......
......X.....X..................X....... ........X......
.......XXXXX....................X...............X.......
.......XXXXXXXXXXX..............X..............XX........
.............XXXXXXXXX..........X...............X........
....................XXXXXXXXXXXX.XX.........XXXXXXX.......
.................................XXXXXXXXXXXXX.....XXX..X.
.....................................................XXXX.
..........................................................
.........................................................
.........................................................
```

3 x 3 slimmed

```
...........
...........
...........
...........
...........
.....X.....
.....X......    ...
.....X...........X.
.....X..........X..
.....X..........X.......
.....X..........X........     XX               ...
.....XXXX.......X..........  ..X....       .....X....
........X.......X..............X.....     ......X......
.........X......X..............X....... ........X......
.......XXXXX....X..............XX...............X.......
.......XXXXXXXXXXX.............X...............XX........
.............XXXXXXXXX.........XX..............XX........
....................XXXXXXXXXXXX.XX.........XXXXXXXXX.....
.................................XXXXXXXXXXXXX.....X.XXXX.
.....................................................X..X.
..........................................................
.........................................................
.........................................................
```

5 x 5 Slimmed

Figure 7.13 Comparison of Slimmed Shapes

requires an algorithm that is permitted to remove the marked points in the image. This can be achieved with the slimming algorithms given previously by removing the references to the image plane containing the marked points. Since it is necessary to delete points from a two cell wide shape the 3 x 3 window program which erodes from each direction in sequence is required.

Removal of these points renders it impossible to exactly re-create the original image from the skeleton; however, an approximation may be generated. Providing care is taken in this stage to preserve the end points of lines, as they are no longer protected by marking, the original may be re-created to within one pixel. In accord with the theory in [43] the shape generated from the thinned skeleton will lie within the original shape. A suitable program is given in figure 7.14a and the thinned skeleton formed in figure 7.15. The information contained in this skeleton has been used to re-create an approximation to the original shape and this is shown overlaid with the original in figure 7.16.

Use of a thinning algorithm that does not preserve line ends may be used at this stage with only slightly worse results. Since the 'thin' will be applied only a single time, at most one point will be removed from each line end. This will introduce an error in the re-created shape which most often will be a difference of one pixel; it should be noted that the same error is introduced elsewhere by this skeletonising method.

Occasionally (with the set of images tested so far, very rarely) the error will worsen to two pixels when a condition such as that shown in figure 7.17 occurs. The topmost point can be removed without introducing an error of more than one pixel but that at the right-hand side introduces an error of two pixels, as shown in the figure. This

```
THIN: [[ C0:=A0 & ( @SIGMA=1 ! ~A7 ! A3 ! ~A1&A2 ! ~A5&A4 ) ]]; {N}
      [[ A0:=C0 ]];
      [[ C0:=A0 & ( @SIGMA=1 ! ~A3 ! A7 ! ~A5&A6 ! ~A1&A8 ) ]]; {S}
      [[ A0:=C0 ]];
      [[ C0:=A0 & ( @SIGMA=1 ! ~A1 ! A5 ! ~A3&A4 ! ~A7&A6 ) ]]; {W}
      [[ A0:=C0 ]];
      [[ C0:=A0 & ( @SIGMA=1 ! ~A5 ! A1 ! ~A7&A8 ! ~A3&A2 ) ]]; {E}
      [[ A0:=C0 ]]
      RETURN;

SIGMA: A1+A2+A3+A4+A5+A6+A7+A8  RETURN; { find the number of
                                              neighbouring points }
                                        { @SIGMA=1 is a line end  }
```

a) Thinning Program Preserving Line Ends

```
THIN: [[ C0:=A0&(~A7!A3!~A1&A2!~A5&A4) ]];
      [[ A0:=C0&(~C3!C7!~C5&C6!~C1&C8) ]];
      [[ C0:=A0&(~A1!A5!~A3&A4!~A7&A6) ]];
      [[ A0:=C0&(~C5!C1!~C7&C8!~C3&C2) ]]  RETURN;
```

b) Thinning Program Removing Line Ends

Figure 7.14 Programs for the Final Thinning Stage

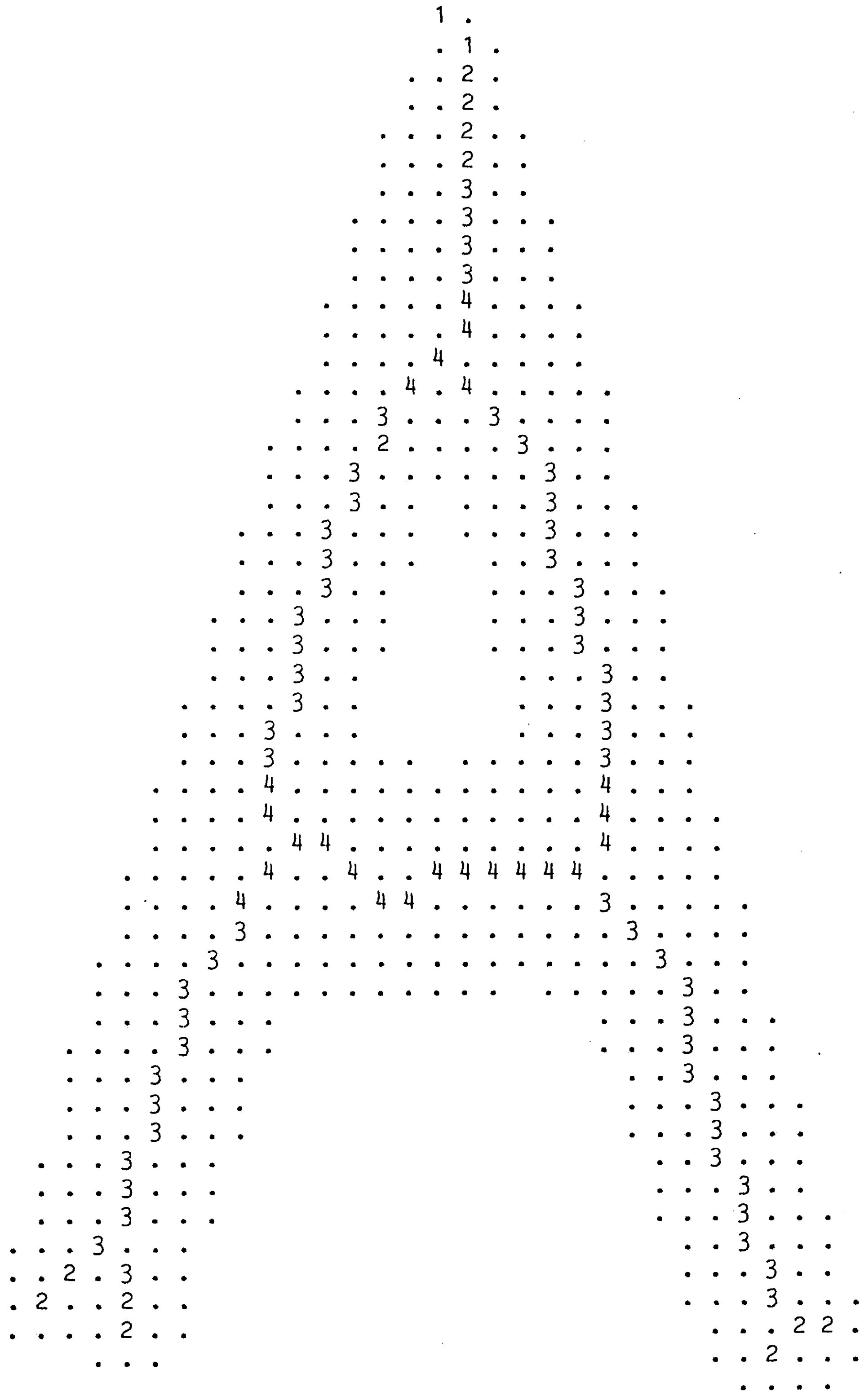


Figure 7.15 Thinned Skeleton

```
                              X .
                              X X X
                            . X X X
                            . X X X
                          . X X X X X
                          . X X X X X
                          . X X X X X
                        . X X X X X X X
                        . X X X X X X X
                        X X X X X X X X
                      X X X X X X X X X .
                      X X X X X X X X X .
                      X X X X X X X X X .
                    . X X X X X X X X X X .
                    X X X X X X X X X X X X
                  . X X X X X X X X X X X X
                  X X X X X X X X X X X X X
                  X X X X X X   X X X X X X .
                . X X X X X X   . X X X X X X
                X X X X X X X     X X X X X X
                X X X X X X       X X X X X X .
              . X X X X X X       X X X X X X X
              . X X X X X X       . X X X X X X
              X X X X X X           X X X X X X
            X X X X X X X           X X X X X X X
            X X X X X X X           X X X X X X X
            X X X X X X X X X   . . X X X X X X X
          . X X X X X X X X X X X X X X X X X X X
          X X X X X X X X X X X X X X X X X X X X .
          X X X X X X X X X X X X X X X X X X X X .
        . X X X X X X X X X X X X X X X X X X X X .
        . X X X X X X X X X X X X X X X X X X X X X .
        X X X X X X X X X X X X X X X X X X X X X X X
      . X X X X X X X X X X X X X X X X X X X X X X X
      . X X X X X X X X X X X X X X   . X X X X X X X
      X X X X X X X                       X X X X X X .
    . X X X X X X .                       . X X X X X X
    . X X X X X X                           X X X X X X
    X X X X X X X                           X X X X X X .
    X X X X X X .                           X X X X X X X
  . X X X X X X                               X X X X X X
  X X X X X X X                               X X X X X X
  X X X X X X .                               X X X X X X X
. X X X X X X                                   X X X X X X
X X X X X X X                                   X X X X X X
X X X X X X X                                   X X X X X X X
X X X X X X X                                     X X X X X X
      X X X                                       X X X X X X
                                                    X X X .
```

Figure 7.16 Re-created Shape ('X's) Over Original Shape ('.'s)

occurs because the region described by that point extends to the edge of the figure, but on its removal the distance function at the new edge can re-create to no better than within two points of the original edge. A suitable program is given in figure 7.14b.

7.3.5 Removal of Noise Spurs

Roughness on the edges of the shape will cause small spurs on the skeleton which may be detrimental to the recognition of the object from its skeleton. Such roughness is often caused by noise on the image and could, with a suitable algorithm, be removed without altering the validity of the skeleton as a description of the object.

Spurs caused by noise will be characterised by being small and may be easily removed from the skeleton. It is suggested in [43] that unit length spurs may be removed. Points that are line ends next to points that branch fulfill this criterion and can be removed. A program for this operation is given in figure 7.18 and the resultant skeleton in figure 7.19.

7.4 CONCLUSION

Skeletonisation of shapes provides a compact way of representing them and, by employing the algorithm implemented here, of re-creating the original to a good accuracy. The precise choice of thinning program is determined by a number of features and, as has been shown, the choice of window size is one of importance. In this, as in all image processing tasks there is a great need for facilities to carefully monitor the program in development and to test it on a number of examples; these were all provided by the PPL2 system.

Although the execution times given here are rather long in real terms they give an indication of the best approach to use on faster

```
    <     . . . . . . .
    <     . . . . . . . . .
    <   . . . . . . . . . .
etc.<   . . . . 4 . . . . .
    <   . . . . 4 . . . . .
    < . . . . 4 4 5 . . . .
    < . . . . 4 . . . . . .
    L v v v v v v v v v v v
            etc.
```

Thinned Skeleton on Original Points

```
    <   X X X X X X X
    <   X X X X X X X X X
    < X X X X X X X X X X
etc.< X X X X 4 X X X X X
    < X X X X 4 X X X X X
    < X X X 4 4 5 X X X X
    < X X X 4 X X X X X X
    L v v v v v v v v v v
           etc.
```

Re-created Shape -
Skeleton Ends Preserved

```
    <   . . . . . . .
    <   X X X X X X X . .
    < X X X X X X X X . .
etc.< X X X X X X X X . .
    < X X X X 4 X X X . .
    < X X X 4 4 X X X . .
    < X X X 4 X X X X . .
    L v v v v v v v v v v
           etc.
```

Re-created Shape -
Skeleton Ends Deleted

Figure 7.17 Shape Re-construction from Thinned Skeleton

equipment. The choice of suitable processing hardware to implement the algorithms is aided by being able to test them beforehand on a system such as this and to compare methods. Given a suitable machine a speed improvement of several orders of magnitude could be achieved.

```
CLEAN:                      { Form plane of number of neighbours }
   [[ Q0:=A1+A2+A3+A4+A5+A6+A7+A8 ]];
                            { Use it for spur removal test       }
   [[ IF Q0=1                          { Line end                }
        & Q1+Q2+Q3+Q4+Q5+Q6+Q7+Q8>2    { Next to branching point }
       THEN
          A0:=0             { So remove the point                }
      FI
     ]] RETURN;
```

Figure 7.18 Noise Spur Removal Program

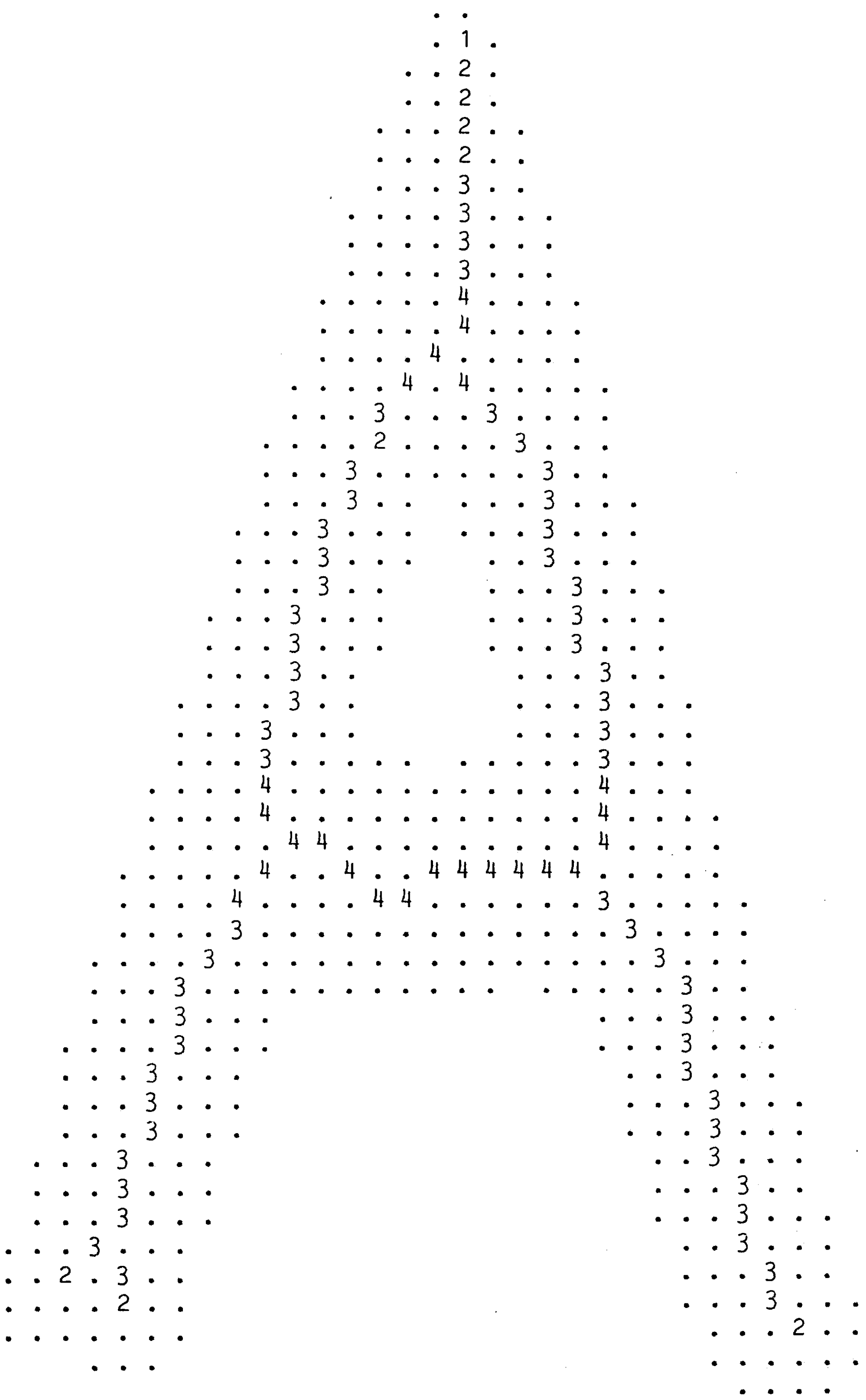
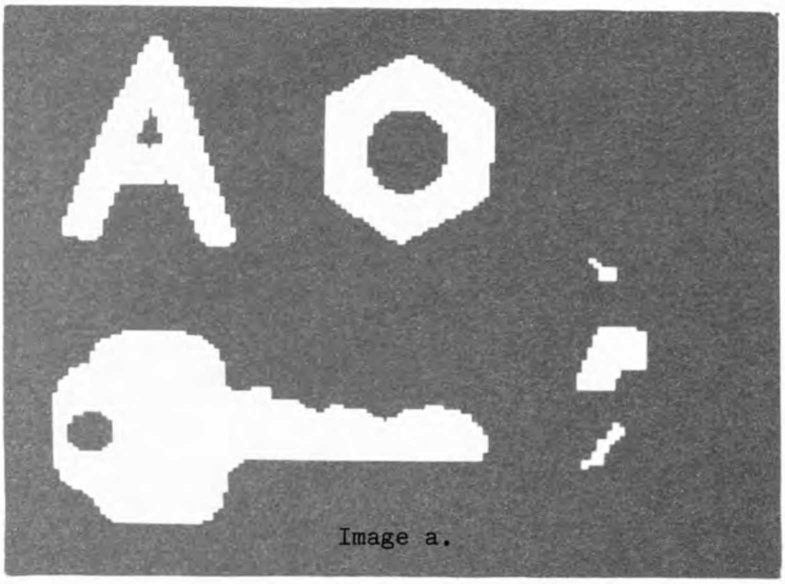


Figure 7.19 Skeleton After Removal of Noise Spurs



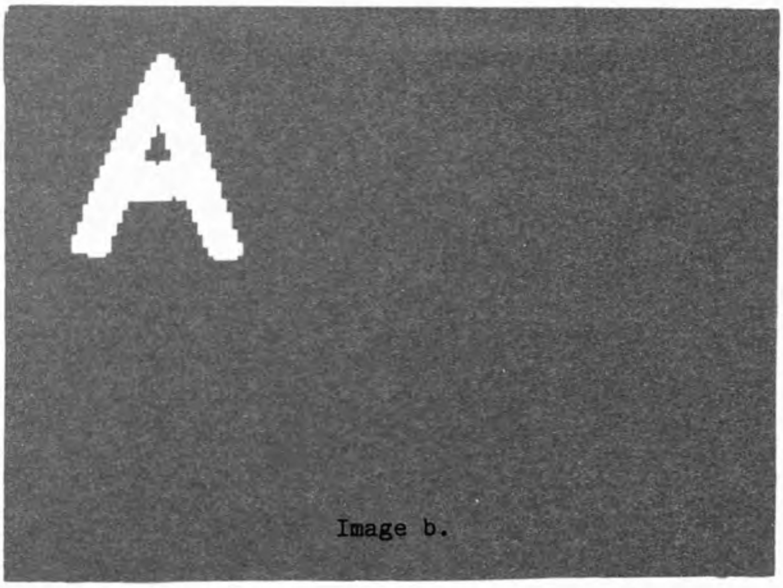
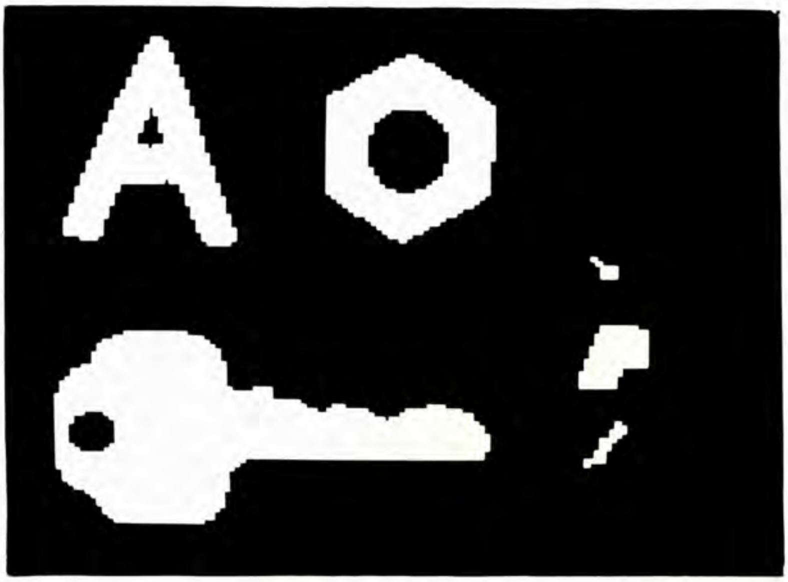
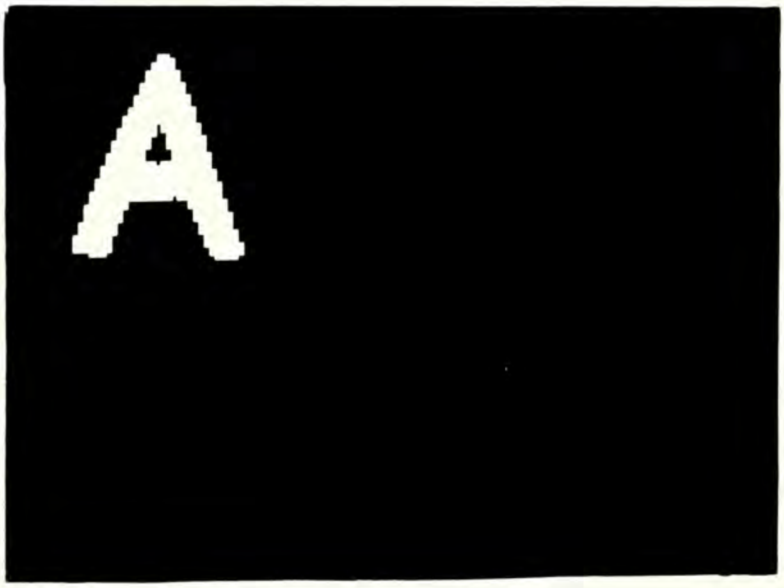
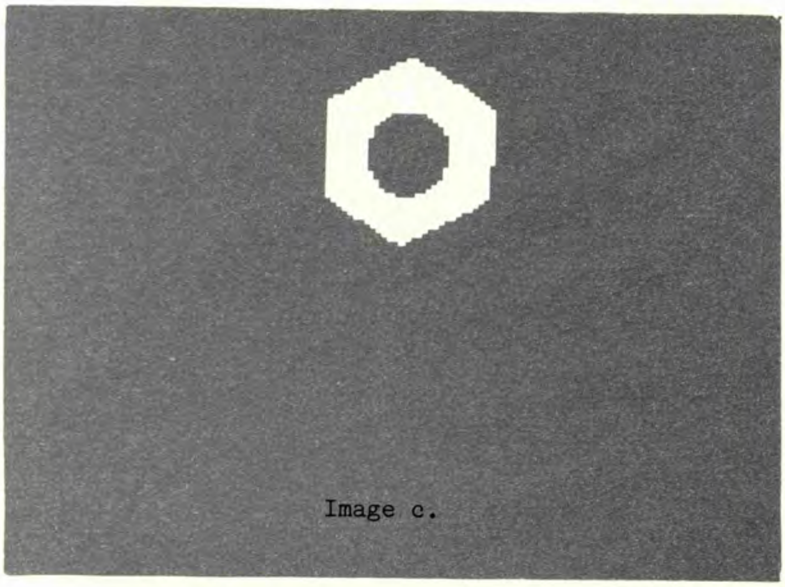


Figure 7.20 Images for Speed Comparison







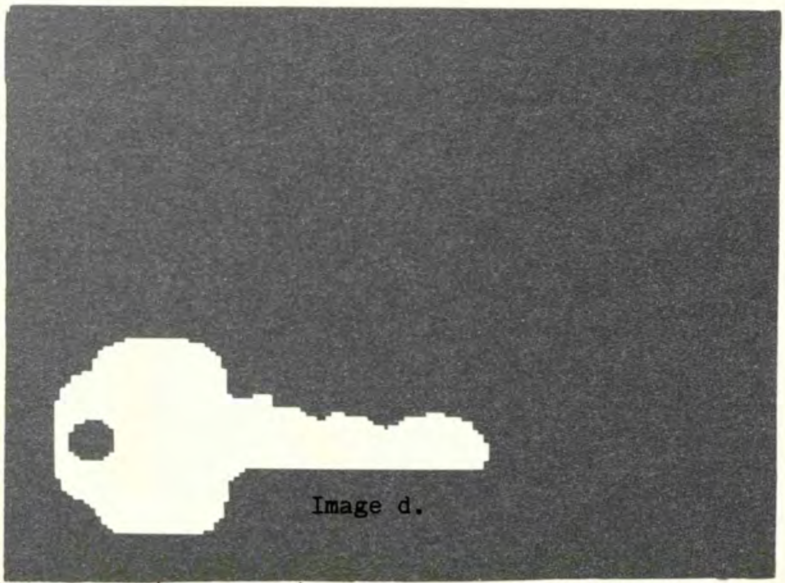
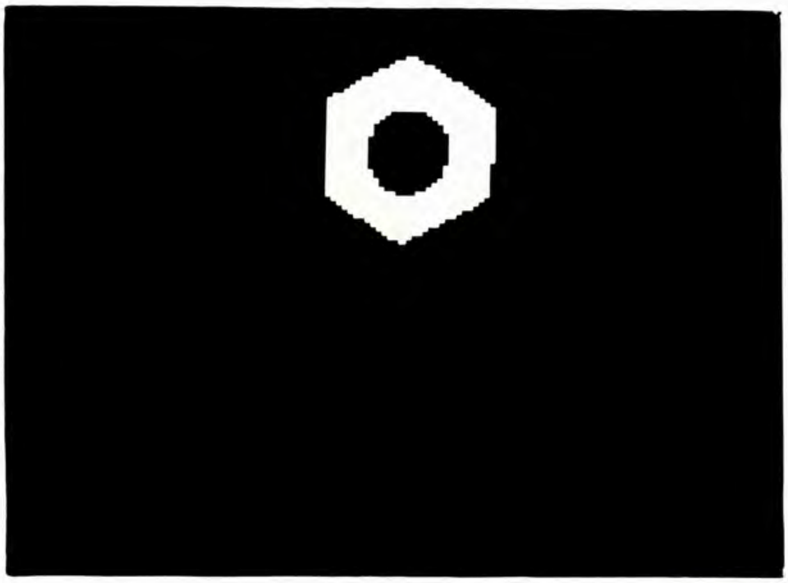
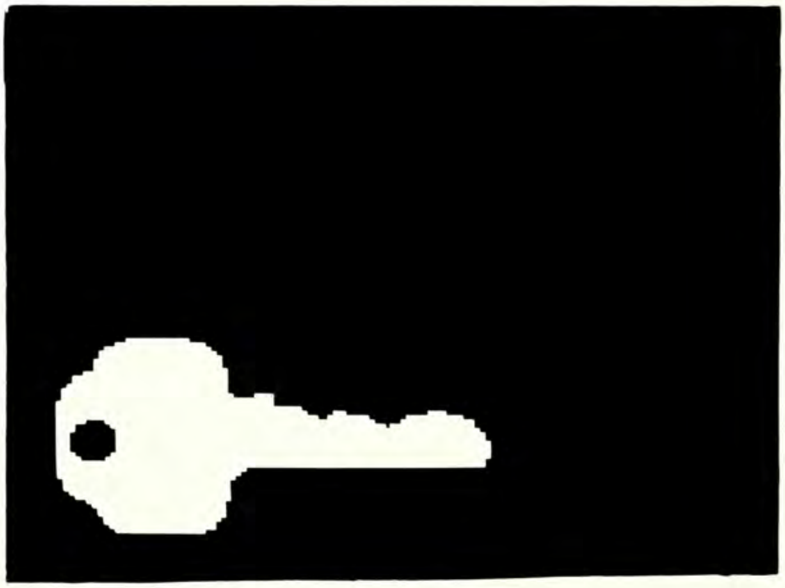
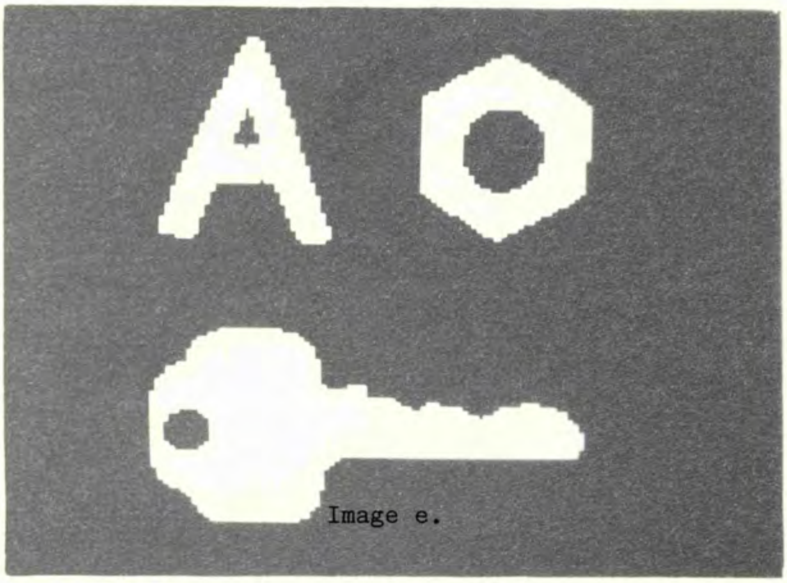


Figure 7.20(continued) Images for Speed Comparison







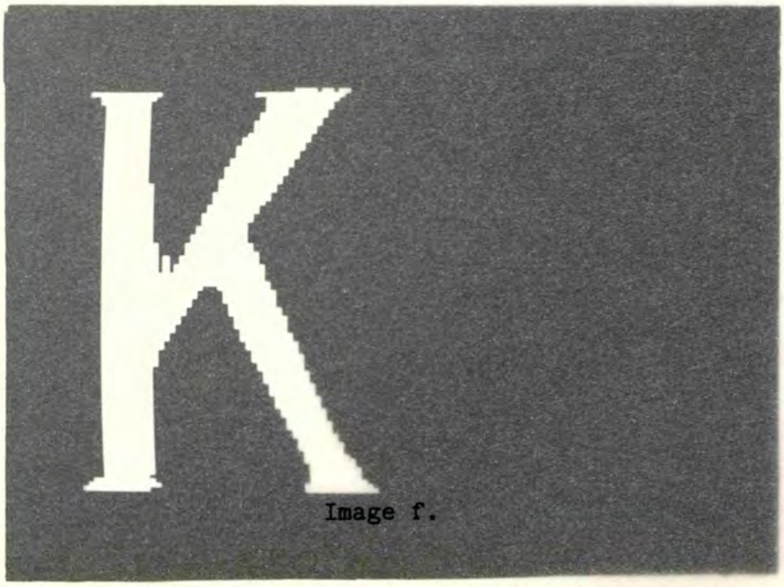
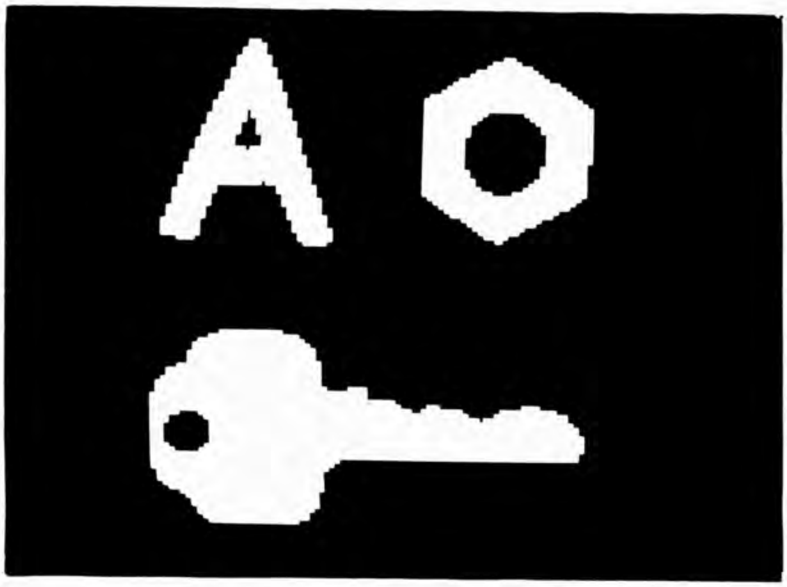
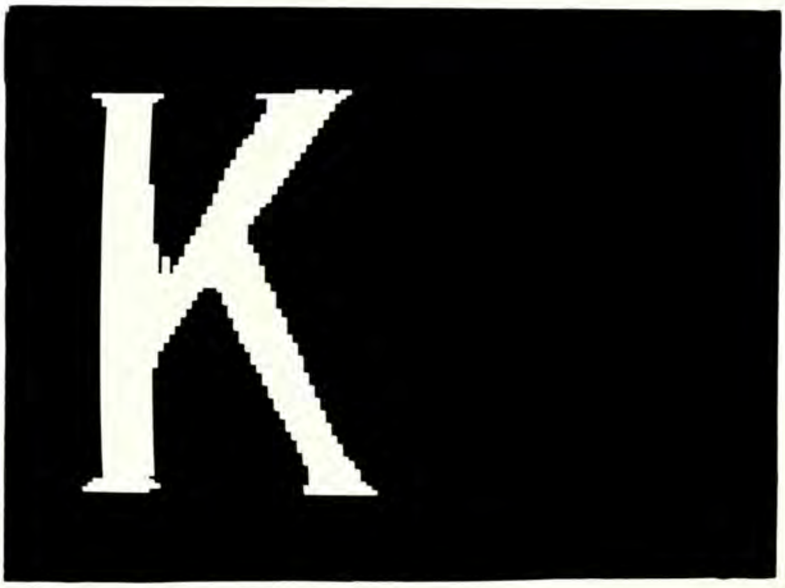


Figure 7.20(continued) Images for Speed Comparison


A


K

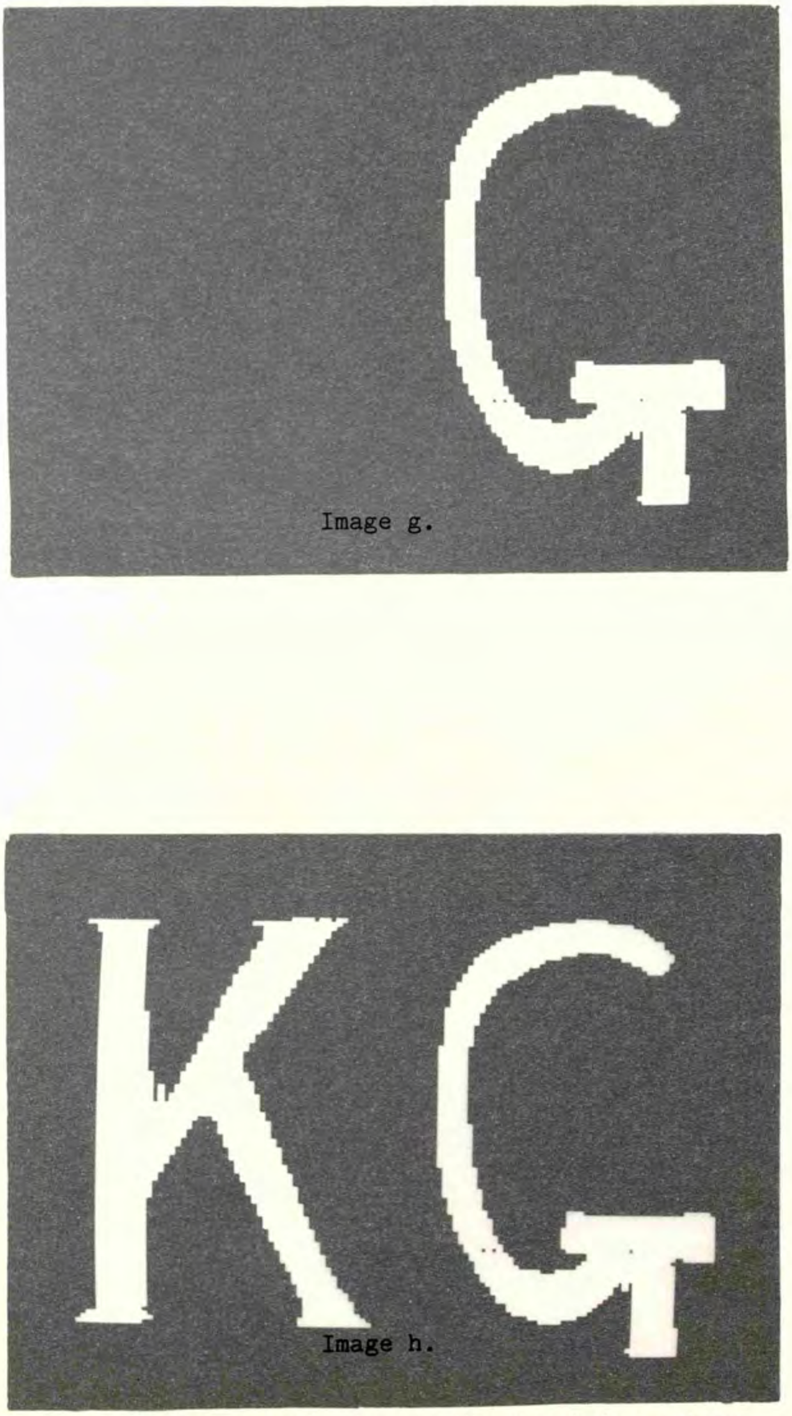
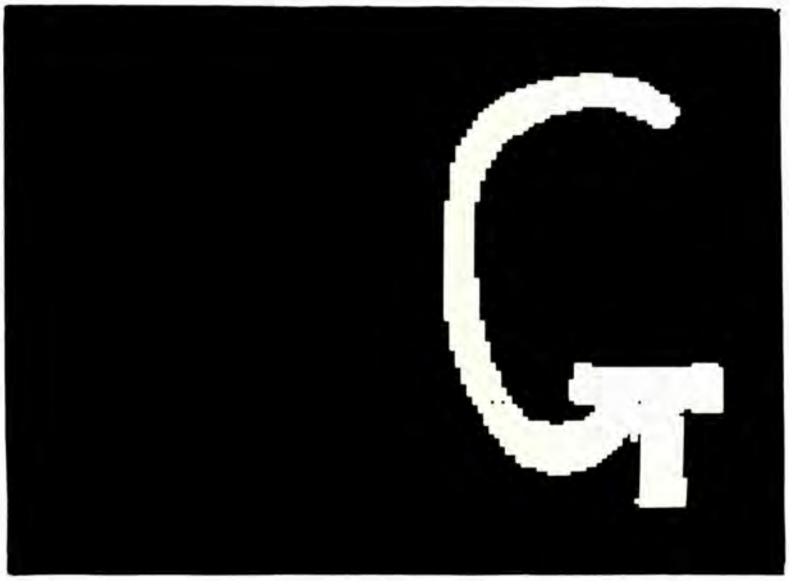
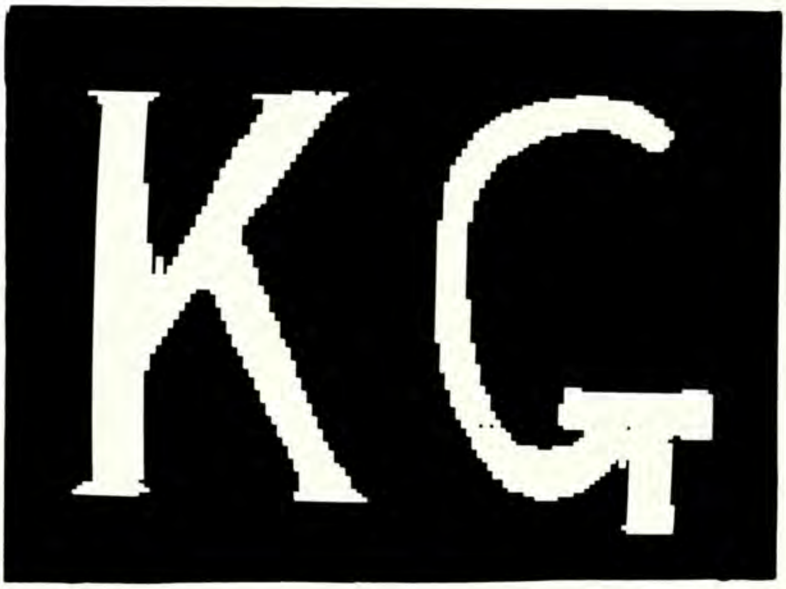


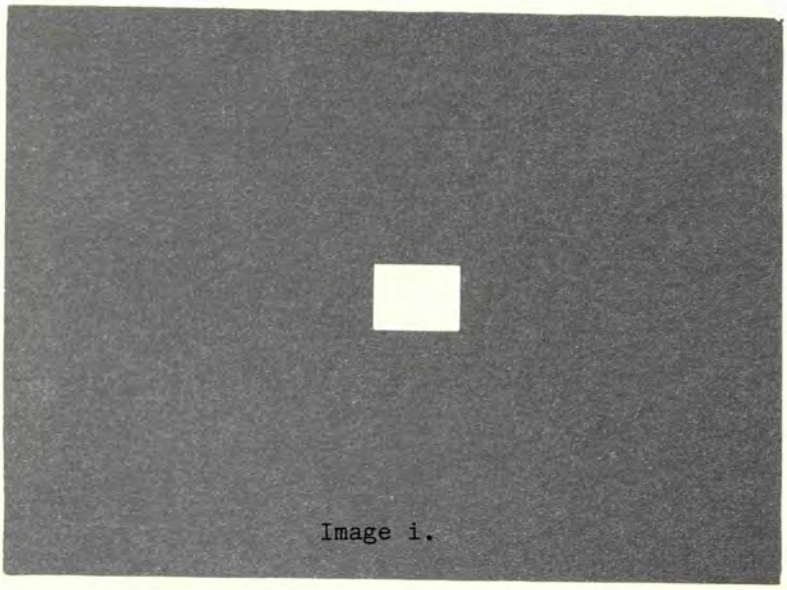
Image g.

Image h.

Figure 7.20(continued) Images for Speed Comparison


G


KG



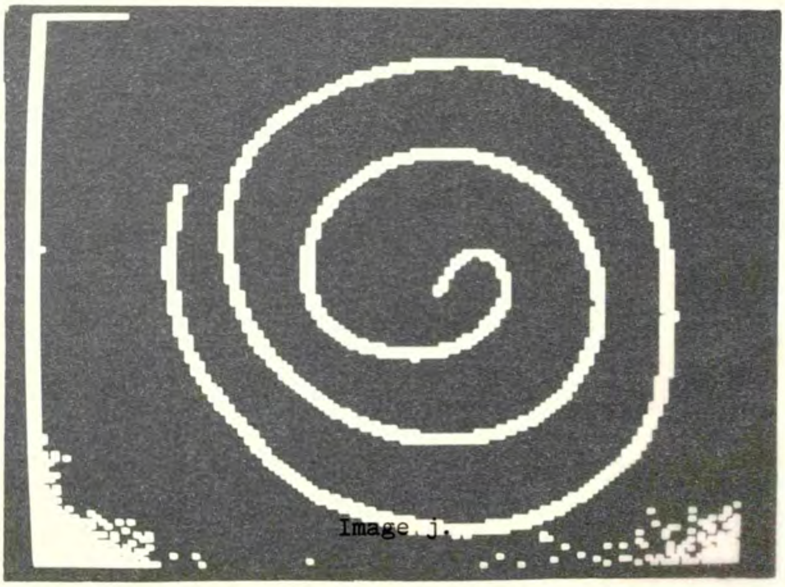
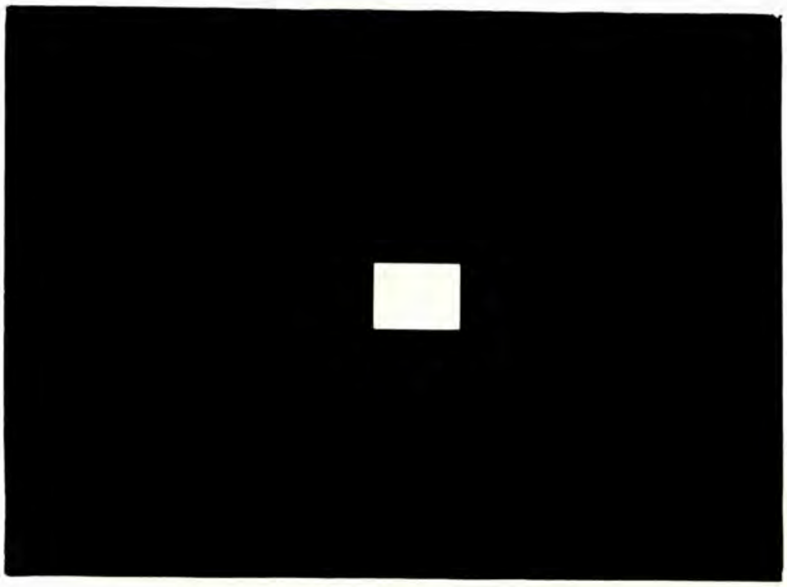
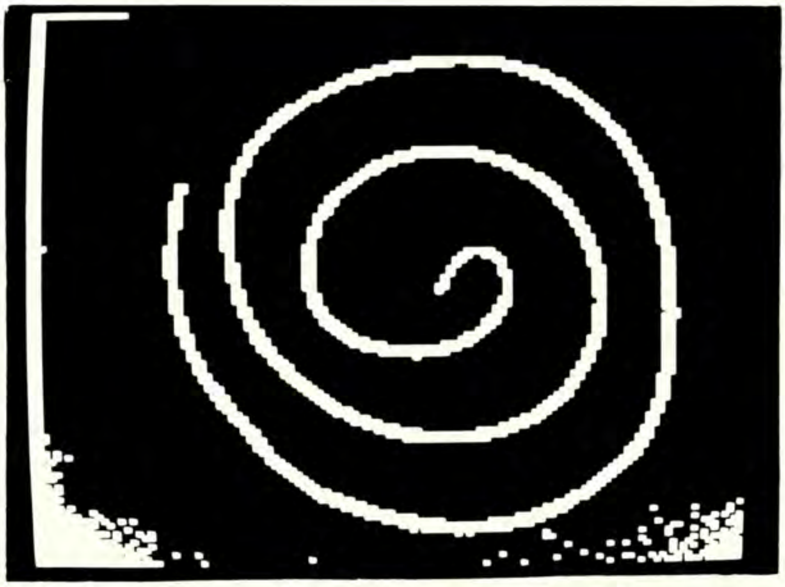


Figure 7.20(continued) Images for Speed Comparison





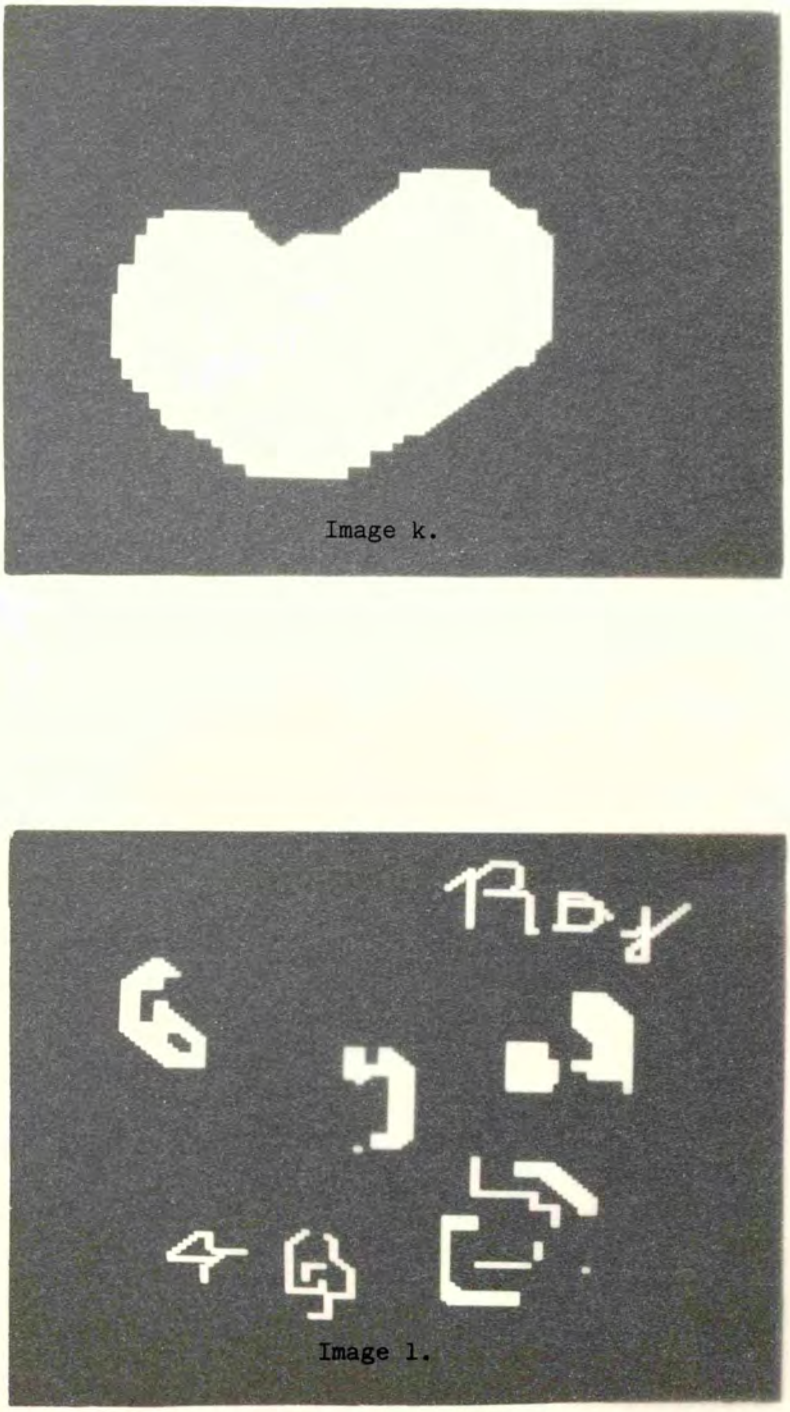
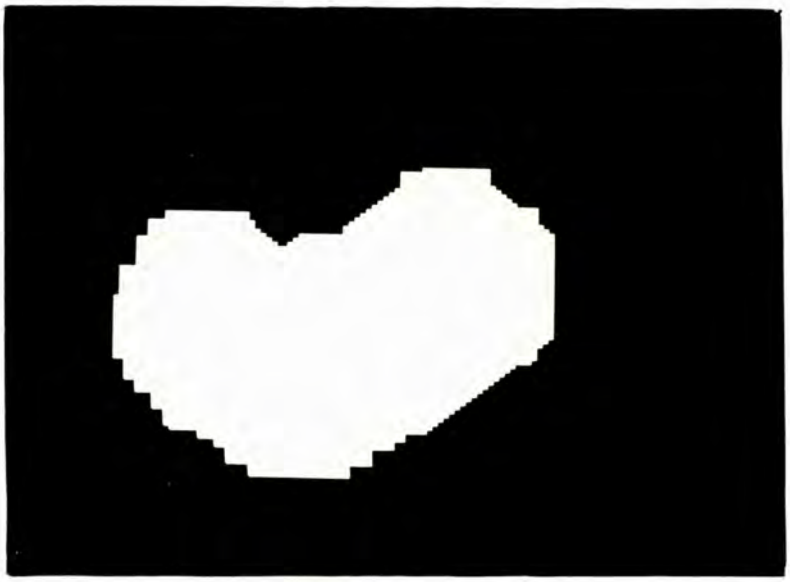
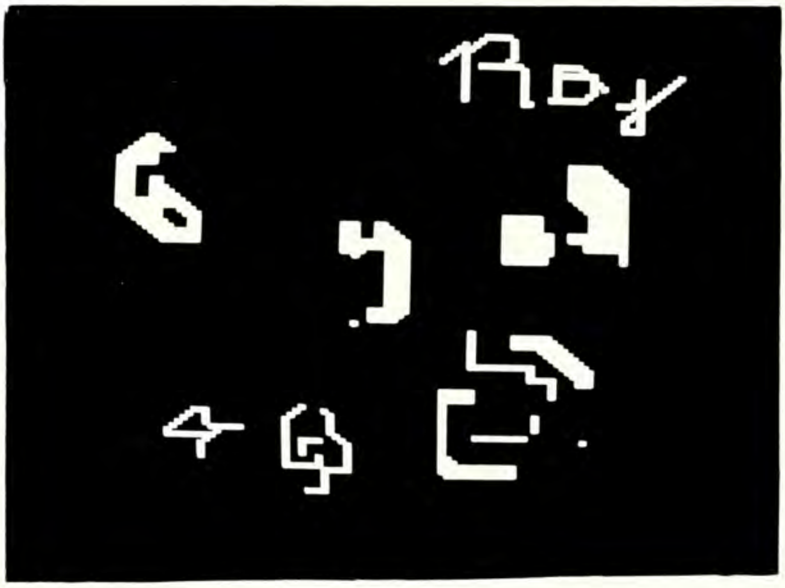


Image k.

Image l.

Figure 7.20(continued) Images for Speed Comparison





8. CONCLUSIONS

8.1 INTRODUCTION

Systems used for image processing fall into two distinct categories, those used for research and development of techniques and those on the factory floor performing a real job. The factory environment requires nothing more than a black box that produces signals to control the process to which it is attached. A development system, on the other hand, must yield a great many intermediate results for detailed examination and checking.

The development system described in this thesis is summarized in section 8.2. A number of improvements to the PPL2 language have suggested themselves in the course of its use and are described in section 8.3.

Industrial systems have rather different requirements from the development system, perhaps the most important being a much greater speed. The system described in this thesis has produced results that indicate the type of architecture that is required in industry. Section 8.4 discusses these indications.

Implications of the use of different processing techniques are briefly discussed in section 8.5.

8.2 THE DEVELOPMENT SYSTEM

PPL2 as described in this thesis is intended to fulfil the requirements of a development system and provide facilities that assist in the design of image processing algorithms. As a system PPL2 has not been found to lack any feature that would significantly affect its usefulness. Perhaps the single most important feature of PPL2 is the fast interaction it provides. Images can be quickly stored on and retrieved from disc storage, input from camera, displayed on a monitor, printed, dumped in character form and displayed as numeric values. Programs may be created, listed, retrieved, saved, edited (with the aid of a screen editor) and run. It is possible to create, test and modify programs very quickly and to observe their effects on a large number of test images both off camera and from a disc based library. Whilst speed is not of utmost importance when developing algorithms it is relevant in that to fully test the program it must be run a number of times with different data. A slow system or one that is inconvenient to use will discourage thorough test procedures and may cause an error to be missed.

Transfer from this development system to a run-time system is made easy by the use of a high-level, machine independent language, PPL2. It must be remembered, when developing algorithms, that the run-time hardware may not be able to support certain classes of algorithms, for example, sequential or parallel procedures may be required.

8.2.1 The PPL2 Language

PPL2 was designed to be used for image processing and thus provides constructs that are of particular use in this application. Particularly useful short-hand notations include the double square bracket scan indicator and the fundamental operators MAX, MIN and RANGE. Other frequently used operators, for example - the logical and arithmetic operators, have been given short, often single character, symbols to ease programming.

By using a high-level language, concepts rather than detailed data manipulation are programmed, thus avoiding an intricate program which cannot be understood without a detailed explanation. When bit manipulation is required it can be accomplished within the language without the usual recourse to accessing special routines or writing special operators which may not be particularly fast in execution. As a high-level language PPL2 is compact and can in principle be transported between computers with ease.

8.2.2 The Image Storage System

Only large computers have enough memory to contain one or more images so that it is commonly necessary to process an image in stages, each on a small sub-image. Partitioning of an image requires an overhead in "stitching" together the various parts to produce the whole which contributes to the execution time. By removing the image to an external memory unit with a suitable access method this problem can be avoided. Further benefits can be obtained in the provision of fast, hardware, window mappers and edge of image detectors which improve the system performance.

The image storage hardware described in chapter 5 provides these functions very successfully and consumes only 1/32 of the machine´s memory space for a total image storage capability which would otherwise require the whole machine. The window mapping scheme provides a large reduction of computational effort at run-time resulting in very acceptable execution times.

8.3 IMPROVEMENTS TO PPL2

PPL2, in common with most other computer languages, has been found to contain a number of ambiguities that could cause differing results to be obtained from different implementations. To be fully useful as a communications language this should not occur. Most of these problems are a direct reflection of the computer used to execute PPL2 programs, an event that should not occur in a (supposedly) machine independent high-level language.

8.3.1 Data Type Definitions

During the design of PPL2 it was felt that it would only be necessary to work with integers as provided by the PDP11 with pixels forming a sub-range of values. The pixels are represented by 8-bit unsigned values while 16-bit signed integers are also supported by the machine. After a number of odd and inconsistent results had been obtained it became clear that 16-bit working storage is occasionally inadequate. This occurs when more than just a few pixel values are combined and was soon identified as an overflow problem. Only 127 pixels of 8-bits each can be summed without running the risk (it does not always happen) of overflow, this corresponding to just an 11 x 11 window. The problem is further aggravated when the values are weighted and the restriction on window size has to be further tightened. Squaring a pixel value immediately runs the risk of overflow. With care the restriction may not be fatal and PPL2 has nonetheless continued to be extremely useful provided that the problem is considered when designing an algorithm. Failure to note the effects of overflow can lead to very odd results and even extensive alteration of an algorithm in an attempt to correct them. Using long integers on a machine designed to support in hardware only 16-bit precision requires the use of a small program for each operation and is considerably slower than directly using the machine's instruction set. The problem is much more serious when using 8-bit architecture microprocessors. Ideally the computer should support 24- or 32-bit arithmetic within its basic instruction set, something found only on the more expensive mini-computers and latest generation of micro-processors.

There have been occasions when integer arithmetic could not provide an accurate enough result for meaningful comparisons to theory to be made and real number arithmetic is then required. Unfortunately this tends to be slow relative to integer arithmetic unless special processors are used. It would seem likely that real number arithmetic should be avoided in an application program on the grounds of speed but there is obviously a need to provide it for research.

Several different types of data are required at different times in a program and the simple answer of providing a single large type (e.g. high precision real numbers) is not really satisfactory. It requires a large overhead in space and time on the many occasions in which it is not needed. The obvious answer is to adopt a system of declaring program variables as being of a type appropriate to the section of program concerned. There are, however, problems in using this approach for an interactive language, for example - ensuring that when a part of the program is called upon to be executed the appropriate declarations have been previously executed. Interactive systems such as PPL2 are very valuable for fast evaluation of a large proportion of image processing algorithms but in a few cases it would seem to be necessary to move to a semi- or non-interactive system.

Even those languages that do support multi-type variables (such as FORTRAN) rarely provide an explicit declaration of their precision in the program. The only information generally provided on the precision of operations occurs within the language or machine manual: this is of little help when transporting programs from machine to machine. Languages must provide a clear definition of the precision used either as part of the language definition or in the variable declarations themselves.

8.3.2 Overflow

If it is at all likely that an overflow will occur it is necessary to detect it and take an appropriate action or a whole range of odd results may appear. Many floating-point processors (for example the FP11 in a PDP11) can detect and trap, with an interrupt, overflow or other error conditions. Integer overflows generally do no more than set a flag which can be tested if required (with consequent run-time overheads). It is quite likely that an image overflow does not correspond to a hardware overflow in the computer - for example the case of 8-bit pixels on a 16-bit machine. Some means of detecting and flagging overflows from a user defined precision of arithmetic would be desirable; it is possible that such a mechanism could be incorporated in the image storage device. An appropriate set of instructions in the programming language would be required to control this mechanism and make clearly visible the permissible range of values and error conditions. Without such a facility programs lose a degree of portability between machines and become restricted in their applicability.

An interim partial solution in PPL2 is the provision of an operator that tests a pixel value and clips it to range limits if it is not within them. This greatly reduces the number of apparently ridiculous results but is still far from an ideal solution.

8.3.3 PPL3

In order to incorporate the ideas expressed above into an improved image processing environment a new language, PPL3, is being developed. It is intended to be used as a stand alone compiler or within a command loop with an editor to provide semi-interactive facilities; the program can be interactively edited but must be compiled and run as a whole.

8.4 INDUSTRIAL SYSTEMS

Single sequential processors when used for image processing are adequately fast for research use but appear to be far too slow to be used in an industrial application. The benchmark timings given in appendix A show execution times that are too long by two or three orders of magnitude when the application is, for example, inspection of parts on a conveyor belt.

To provide an industrially useful system a large increase in speed has to be achieved. The processor used for these timings has a cycle time of about 1uS which is reduced by only about one order of magnitude in large main frame computers. The alternative to faster processors is a larger number of them, in other words a multi-processor system. To a first approximation a system of n processors should be able to handle n times as much information in a given time as one processor. This is only accurate if the processors can work on data that is independent of the results from other processors; in a network of sequential processors on a single image (the problem we are trying to speed up) this is unlikely to be true. The full benefit of such a system is likely to be realised only when the image can be divided into independent sub-images each operated upon by its own processor.

Collections of processors in which the dependence of the data is recognised and catered for can be built and are seen in the form of vector or array processors. The vector processor can be used to operate upon part or all of one line of an image at a time while the array processor uses the whole image as its data base. Unfortunately such processors are, at the present, very expensive and thus relatively unattractive so that alternative methods will be of interest for some time.

Computer designers have continually strived for greater speeds in their machines and a number of techniques have been developed to achieve this. One such method that is of particular interest in image processing is called pipelining. This stems from the fact that many of the basic machine operations can be divided into several smaller stages, each using a physically separate piece of hardware. For the execution of a basic operation the data passes through the sub-operations in turn. Whilst the data is occupying a sub-stage the rest are idle; pipelining uses these idle times to process other data. When the first datum has passed to the second sub-operation another datum enters the first. Each datum takes the usual time to pass completely through the system but the total data throughput is increased many-fold. Processors specifically designed for image processing have made use of this technique to increase their speed, for example [53].

Image operations can be divided into many stages and it is often not the absolute processing time but the rate at which images can be accepted that is the limiting factor. For example, a system that is employed for industrial inspection of parts on a conveyor belt must handle a large number of images per second, but it does not matter too much if it takes a couple of seconds to produce a decision on a given part. The only thing that will have happened in the meantime is that the parts will have moved along the belt a little; indeed, in this application, the mechanical layout may dictate a delay from the inspection to the routing stations!

8.4.1 The Shift Register Processor

Shift register processors are ideally suited for use as sub-operations in the complete image processing task. Although the time required to process an image is the time required to completely scan that image (20ms for a television signal) data from the process is available after only two line periods (128us), for a 3 x 3 window, and can be passed to the next stage. This delay depends only upon the window size (for a n x n window the delay is n-1 lines plus n-1 pixel points) and the speed required from the processor is dictated by the resolution required. If a throughput of 10 images per second is required, with the television standard above, it would be possible to process the image through about 800 steps - this is rather more than any program I have yet needed to write! With this scheme the processors are idle 4/5 of the time and by making use of this fact it would be possible to use only 140 processors and still achieve the throughput above.

Cheap processors for a 3 x 3 window and binary operations can be produced rather more cheaply than array processors for the same performance and it seems likely that these can produce an effective system where binary images are employed. Unfortunately, for some applications, binary images cannot be directly used thus requiring either faster grey-scale shift register processing elements or a fast grey to binary conversion.

8.5 ALGORITHM DEVELOPMENT

Algorithms for image processing may be written for either a parallel or sequential architecture depending on the intended hardware to be used for implementation. Sequential algorithms can be executed efficiently only on a machine with sequential architecture - simulation on a parallel machine is an expensive way to go slowly. Parallel algorithms can be simulated on a sequential machine, albeit slowly, provided that more than one image storage area is available; this provides a development system at moderate cost (for example the PPL2 system described earlier). Shift register processors can be used to implement either a parallel algorithm or one direction of a sequential algorithm.

Architectures that execute parallel algorithms would seem to be the most likely to be used in a situation where speed of operation is a requirement; this has long been recognised [54]. It seems reasonable, then, to concentrate development work in parallel algorithms. (Even if the problem appears to be easily soluble by a sequential process algorithms should be re-implemented for a parallel machine to ensure universal application.) This is particularly important as the number of parallel machines being developed is increasing.

8.5.1 Window Size

Image operations may be performed with a variety of different sized windows, the choice being influenced by two major factors, namely speed of execution and ease of implementation. Ideally the former should take priority but in practice it is often the latter that does so.

Finding the skeleton of an image has been used as a task that can be implemented with more than one window size (see chapter 7). It has been found that the use of a window larger than the usual 3 x 3 pixels can bring speed benefits in parts of the process. Heavy use was made in this task of the facilities provided by the PPL2 system, possibly explaining why ease of implementation is often a criterion for the choice of window size.

It would appear to be worthwhile to investigate the use of windows of various sizes for a given task to find the optimum in terms of performance. It must be remembered, however, that the hardware used to implement the algorithm will impose its own restrictions on execution time. Differences of a factor 2, as found in the case of skeletonisation, may well be masked by the different operating environment, or may be increased; only a final test on the complete system will prove the case.

8.6 FUTURE

Image processing and pattern recognition will clearly be of major importance in the future and it is necessary to provide the appropriate machinery for the development and execution of suitable techniques. There are two closely linked aspects to this aim: firstly the construction of suitable hardware for the storage and processing of image data, and secondly the provision of a suitable software environment for the specification of algorithms.

Digital computers of the type most commonly found are not at all well suited to the task of fast image processing. Some improvement can be made by the addition of image storage and access hardware such as that described in chapter 5. Although this still does not provide an industrially satisfactory machine it is well suited to a research environment. The ideal machine will have to wait for specialised integrated circuits that can be interconnected to form fast processors for image type data; machines such as the CLIP series indicate that such an approach is possible.

Software for the development and description of image processing algorithms is, as yet, relatively unavailable. The only generally available computer languages were not designed for the type of tasks required in image processing and hence do not provide a very convenient system. PPL2 was designed to provide the facilities required in a clear and easy to use manner, both in terms of the language for image processing and of the peripheral aids to image analysis. PPL2 has, on the whole, proved itself to be extremely valuable but as has been noted above is rather weak in some areas. There is a great deal of scope for the design of much improved languages for image processing both to make use of the available image-computers and to incorporate the structures required for pattern recognition.

ACKNOWLEDGEMENTS

I would like to thank Dr.E.R.Davies for his supervision of this research and for the many helpful comments and suggestions that he has made during the preparation of this thesis.

My thanks are due to the Science and Engineering Research Council for providing the grant necessary to allow me to undertake this work.

REFERENCES

[1] Hall E.L., "Computer Image Processing and Recognition",
Academic Press, Chapter 1 (1979).

[2] Mayer M., "Investigations into Trainable Learning Machines",
PhD. Thesis, London University (1982).

[3] Deutsch E.S., "Thinning Algorithms on Rectangular, Hexagonal, and
Triangular Arrays",
CACM, vol 15, no.9, 827-837 (September 1972).

[4] Batchelor B.G. and Brumfitt P.J., "Command Language for
Interactive Image Analysis",
Proc IEE, vol 127, Part E, no.5, 203-218 (September 1980).

[5] Plummer A.P.N., "Structural Analysis and Classification of
Patterns",
PhD. Thesis, London University (1980).

[6] Nevatia R., "Characterization and Requirements of Computer Vision
Systems",
in "Computer Vision Systems"
Eds. Hanson A.R. and Riseman E.M.
81-87 (1978)

[7] Paul G., "Large Scale Vector/Array Processors",
in "Advances in Digital Image Processing"
Ed. Stucki P.
277-300 (1979).

[8] Kruse B., "A Parallel Picture Processing Machine",
IEEE Trans Comput, vol C-22, no. 12, 1075-1087 (Dec 1973).

[9] Unger S.H., "A Computer Oriented Toward Spatial Problems",
Proc IRE, vol 46, no. 10, 1744-1750 (1958).

[10] Unger S.H., "Pattern Detection and Recognition",
Proc IRE, vol 47, no. 10, 1737-1752 (1959).

[11] McCormick B.H., "The Illinois Pattern Recognition Computer -
ILLIAC III",
IEEE Trans Comput, 719-813 (December 1963).

[12] Barnes G.H. et al., "The ILLIAC IV computer",
IEEE Trans Comput, vol C-17, no. 8, 746-757 (Aug 1968).

[13] Duff M.J.B. et al., "A Cellular Logic Array for Image Processing",
Pattern Recognition, vol 5, 229-247 (1973)

[14] Reddaway S.F., "The DAP Approach",
Infotech State of the Art Report on Supercomputers,
2, 309-329 (1979).

[15] Hunt D.J., "The ICL DAP and its Application to Image Processing",
in "Languages and Architectures for Image Processing"
Eds. Duff M.J.B. and Levialdi S.
275-282 (1981).

[16] Fountain T.J. and Goetcherian V., "CLIP 4 Parallel Processing System",
Proc IEE, 127, Part E (no 5), 219-224 (1980).

[17] Duff M.J.B., "CLIP 4: A Large Scale Integrated Circuit Array Parallel Processor",
Proc 3rd Int Joint Conf on Pattern Recognition, Coronado, California, November 8-11, 728-733 (1976).

[18] Reynolds D.E. and Otto G.P., "Software Tools for CLIP 4",
Report no. 82/1, Image Processing Group, University College, LONDON (Jan 1982).

[19] Fountain D.J., "CLIP 4: A Progress Report",
in "Languages and Architectures for Image Processing"
Eds. Duff M.J.B. and Levialdi S.
283-291 (1981).

[20] Kashner T., Wu A.Y. and Rosenfeld A., "Image Processing on MPP:1",
Pattern Recognition vol 15, no.3, 121-130 (1982).

[21] Brooks F.P. Jr., "The Mythical Man-Month",
Addison-Wesley, Chapter 8 (1975).

[22] Wood A., "The Interaction Between Hardware, Software and Algorithms",
in "Languages and Architectures for Image Processing"
Eds. Duff M.J.B. and Levialdi S.
1-11 (1981).

[23] Maggiolo-Schettini A., "Comparing some High-level Languages for Image Processing",
in "Languages and Architectures for Image Processing"
Eds. Duff M.J.B. and Levialdi S.
157-164 (1981).

[24] Levialdi S., Maggiolo-Schettini A., Napoli M. and Uccella G.,
"Pixal: A High-level Language for Image Processing",
in "Real-Time/Parallel Computing",
Eds. Onoe M., Preston K. and Rosenfeld A.
131-143 (1981).

[25] Preston K., "Languages for Parallel Processing of Images",
in "Real-Time/Parallel Computing",
Eds. Onoe M., Preston K. and Rosenfeld A.
145-158 (1981).

[26] Woodward P.M. and Bond S.G., "ALGOL 68-R Users Guide",
HMSO, London (1975).

[27] Jensen K. and Wirth N., "PASCAL User Manual and Report", Springer-Verlag, New York, Second Edition (1975).

[28] Aho A.V. and Ullman J.D., "Principles of Compiler Design", Addison-Wesley (1979).

[29] Gries D., "Compiler Construction for Digital Computers", Wiley (1971).

[30] Ferranti Semiconductors, "ZNA134J: CCIR/EIA TV Synchronizing Pulse Generator",
(1977).

[31] Digital Equipment Corporation, "PDP11 Bus Handbook",
(1979)

[32] Klinger A., "Data Structures and Pattern Recognition", Proc. 1IJCPR, 497-498 (1973).

[33] Uhr L., "Layered ′Recognition cone′ Networks that Preprocess, Classify and Describe",
IEEE Trans. Computers 21, 758-768 (1972).

[34] Tanimoto S.L. and Pavlidis T., "A Hierarchical Data Structure for Picture Processing",
Computer Graphics Image Processing 4, 104-119 (1975).

[35] Tanimoto S.L., "Regular Hierarchical Image and Processing Structures in Machine Vision",
in "Computer Vision Systems",
Eds. Hanson A.R. and Riseman E.M.
Academic Press, N.Y., 165-174 (1978).

[36] Rosenfeld A., "Quadtrees and Pyramids for Pattern Recognition and Image Processing",
Proc 5ICPR Florida, 802-811 (1980).

[37] Wong R.Y. and Hall E.L., "Sequential Hierarchical Scene Matching",
IEE Trans Comput, vol C-27, no.4, 359-366 (April 1978).

[38] Samet H. and Rosenfeld A., "Quadtree Representations of Binary Images",
Proc 5ICPR Florida, 815-818 (1980).

[39] Samet H., "Connected Component Labelling Using Quadtrees",
Computer Science Centre TR-756, University of Maryland,
College Park, MD (April 1979).

[40] Shneier M., "Extracting Linear Features from Images Using Quadtrees",
Computer Science Centre TR-855, University of Maryland,
College Park, MD (January 1980).

[41] Ranade S., "Use of Quadtrees for Edge Enhancement",
Computer Science Centre TR-862, University of Maryland,
College Park, MD (February 1980).

[42] Wu A.Y., Tsai-Hong H. and Rosenfeld A., "Threshold Selection Using Quadtrees",
Computer Science Centre TR-886, University of Maryland,
College Park, MD (March 1980).

[43] Davies E.R. and Plummer A.P.N., "Thinning Algorithms: a Critique and a New Methodology",
Pattern Recognition, Vol 14, Nos 1-6, 53-63 (1981).

[44] Beun M., "A Flexible Method for Automatic Reading of Handwritten Numerals",
Phillips Tech. Rev. Vol 33, 89-101 and Vol 33, 130-137 (1973).

[45] Cox C.H., Coueignoux P., Blesser B. and Eden M., "Skeletons: A link Between theoretical and Physical Letter Descriptions",
Pattern Recognition, Vol 15, no.1, 11-22 (1982).

[46] Ranade S., Rosenfeld A. and Samet H., "Shape Approximation Using Quadtrees",
Pattern Recognition, Vol 15, no.1, 31-40 (1982).

[47] Tanimoto S.L., "Pictorial Feature Distortion in a Pyramid", Computer Graphics and Image Processing Vol 5, 333-352 (1976).

[48] Samet H., "A Quadtree Medial Axis Transform", TR-803, Computer Science Dept., University of Maryland (1979).

[49] Sherman H., "A Quasi-topological Method for the Recognition of Line Patterns", Proc UNESCO conf., 232-238 (1959).

[50] Hilditch C.J., "Linear Skeletons From Square Cupboards", Machine Intelligence Vol 4, eds Meltzer B. and Michie D., 403-420, Edinburgh University Press, Edinburgh (1969).

[51] Rutovitz D., "Pattern Recognition", J R Statist Soc, Vol 129, 504-530 (1966).

[52] Jones R.N. and Fairhurst M.C., "Skeletonisation of Binary Patterns: a Heuristic Approach", Electron Lett Vol 14, 265-266 (1978).

[53] Gerritsen F.A. and Aardema L.G., "Design and Use of DIP-1: A Fast, Flexible and Dynamically Microprogrammable Pipelined Image Processor", Pattern Recognition, Vol 14, nos 1-6, 319-330 (1981).

[54] Cordella L., Duff M.J.B. and Levialdi S., "Comparing Sequential and Parallel Processing of Pictures", Proc 3rd Int Joint conf on Pattern Recognition, Coronado, California, 703-707 (November 1976).

Appendix A

BENCHMARK TESTS

The Abingdon Workshop on Multi-computers for Image Processing suggested a number of tasks that may be used to provide benchmark timings for image processing systems. A number of these tasks and some others have been programmed on the PPL2 system and the results are presented below. The programs, where appropriate, are given on the following pages. It should be noted that the times given are for a complete task from the command to start to its completion for a 128 by 128 pixel image. Since PPL2 is an interactive system all programs are compiled and executed after the task is initiated, the times are given as a total run time and, in brackets, the compile time. Faster execution could be achieved by re-writing the programs in machine code and carefully optimizing them.

Computer	PDP11/34A	with 128K bytes MOS memory
Operating system	RT11XM V04.00G	
Mass storage	RL02 (2 off)	10.4M byte removable cartridge
Language	PPL2	

A.1 IMAGE INPUT

Camera input is digitised by the image storage hardware in real time but input from disc storage takes place via the main processor.

A.1.1 Camera Input

Minimum	20ms	i.e. 1 to 2 frame scans for a 625-line TV standard,
Maximum	40ms	frame rate 50Hz.

A.1.2 Disc Input

Approximately 1.5s including directory access.

A.2 IMAGE OUTPUT

Output can be to a television monitor for immediate viewing or to disc for storage. Hard copy can be obtained from the monitor output via a Tektronix 4634 Image Forming Module.

A.2.1 Monitor Output

Output time 0s the contents of the working memory are continuously displayed, changing to another working area for display requires one machine instruction of approximately 1us.

A.2.2 Disc Output

Approximately 1.5s including directory access.

A.2.3 Hard Copy Output

Approximately 25s.

A.3 MEDIAN FILTER

7 to 11s (2.3s) run time is data dependent.

A.4 HISTOGRAM

0.87s (0.16s) data collection only.

A.5 THRESHOLD

0.55s (0.04s) Against a given value.

2.03s (0.68s) Threshold derived from histogram (includes histogram and threshold selection.

A.6 SIMPLE OPERATIONS BETWEEN TWO IMAGES

Logical

AND	0.58s	(0.04s)
OR	0.51s	(0.04s)
EX OR	0.54s	(0.04s)

Arithmetic

ADD	0.52s	(0.04s)
SUBTRACT	0.51s	(0.04s)
MULTIPLY	0.64s	(0.04s)
DIVIDE	0.71s	(0.04s)

A.7 PROPAGATION OF DISTANCE FUNCTION

A.7.1 Sequential Algorithm

1.7s (0.23s)

A.7.2 Parallel Algorithm

81.61s (10.4s) for objects up to 128 pixels wide

A.8 SMALL PART LOCATION

7.65s (1.75s) See chapter 6.

A.9 Skeletonisation

15s to 120s Very highly data dependant, see chapter 7.

MEDIAN FILTER

```
{ Replace centre point with median value of 3 x 3 window

  Perform partial bubble-sort to get median value;
  Add value by bubbling up list until in correct position.
  Bubble is performed long-hand as this is faster than a loop.       }

MEDIAN: [[
{ P0,P1 }  IF P0>P1 THEN A:=P0; B:=P1 ELSE A:=P1; B:=P0 FI;
{ P2 }     IF P2<=B THEN C:=P2 ELSE C:=B;
            IF P2<=A THEN B:=P2 ELSE B:=A; A:=P2 FI FI;
{ P3 }     IF P3<=C THEN D:=P3 ELSE D:=C;
            IF P3<=B THEN C:=P3 ELSE C:=B;
             IF P3<=A THEN B:=P3 ELSE B:=A; A:=P3 FI FI FI;
{ P4 }     IF P4<=D THEN E:=P4 ELSE E:=D;
            IF P4<=C THEN D:=P4 ELSE D:=C;
             IF P4<=B THEN C:=P4 ELSE C:=B;
              IF P4<=A THEN B:=P4 ELSE B:=A; A:=P4 FI FI FI FI;
{ P5 }     IF P5<=E THEN F:=P5 ELSE F:=E;
            IF P5<=D THEN E:=P5 ELSE E:=D;
             IF P5<=C THEN D:=P5 ELSE D:=C;
              IF P5<=B THEN C:=P5 ELSE C:=B;
               IF P5<=A THEN B:=P5 ELSE B:=A FI FI FI FI FI;
{ P6 }     IF P6<=F THEN G:=P6 ELSE G:=F;
            IF P6<=E THEN F:=P6 ELSE F:=E;
             IF P6<=D THEN E:=P6 ELSE E:=D;
              IF P6<=C THEN D:=P6 ELSE D:=C;
               IF P6<=B THEN C:=P6 ELSE C:=B FI FI FI FI FI;
{ P7 }     IF P7<=G THEN H:=P7 ELSE H:=G;
            IF P7<=F THEN G:=P7 ELSE G:=F;
             IF P7<=E THEN F:=P7 ELSE F:=E;
              IF P7<=D THEN E:=P7 ELSE E:=D;
               IF P7<=C THEN D:=P7 ELSE D:=C FI FI FI FI FI;
{ P8 }     IF P8<=H THEN I:=P8 ELSE I:=H;
            IF P8<=G THEN H:=P8 ELSE H:=G;
             IF P8<=F THEN G:=P8 ELSE G:=F;
              IF P8<=E THEN F:=P8 ELSE F:=E;
               IF P8<=D THEN E:=P8 ELSE E:=D FI FI FI FI FI;
{ MEDIAN } Q0:=E
           ]]  RETURN;
```

HISTOGRAM

```
{ histogram is in ARRAY }

HIST: [[ FOR I FROM 0 TO 255 DO ARRAY[I]:=0 OD; EXIT ]];
      [[ ARRAY[PO]:=ARRAY[PO]+1 ]]  RETURN;
```

THRESHOLD

```
{ TH is the threshold value }

THRESH: [[ PO:=PO > TH ]] RETURN;
```

THRESHOLD SELECTED FROM HISTOGRAM

```
AUTO:
{ histogram }
   [[ FOR I FROM 0 TO 255 DO ARRAY[I]:=0 OD; EXIT ]];
   [[ ARRAY[PO]:=ARRAY[PO]+1 ]];

{ smooth histogram to avoid false peaks & troughs }
   [[ FOR I FROM 5 TO 250 DO T:=0;
                            FOR J FROM -5 TO 5 DO T:=T+ARRAY[I+J] OD;
                            ARRAY1[I]:=T/11
                         OD;

{ find the first peak (foreground) }
      FOR I FROM 8 TO 247 WHILE A:=ARRAY1[I-3]>=ARRAY1[I];
                                B:=ARRAY1[I+3]>=ARRAY1[I];
                                A!B                        DO PK:=I OD;

{ find the trough }
      FOR I FROM PK TO 247 WHILE A:=ARRAY1[I-3]<=ARRAY1[I];
                                 B:=ARRAY1[I+3]<=ARRAY1[I];
                                 A!B                       DO TH:=I OD;
             EXIT ]];

{ threshold at the trough value }
   [[ QO:=PO<TH ]]
   RETURN;
```

SMALL PART LOCATION

```
FIND: @QUAD; @NINT; @OBJ RETURN;    { Find the parts }

QUAD: OUT:=1;
           { to force compilation only - }   [[
{   BLKSIZ is the block size, 128 = the whole image
    THRESH is the maximum range of values in a constant block }
       BLKSIZ:=8; THRESH:=35;

  { go to top L.H. corner of each block }
       FOR I FROM 0 BY BLKSIZ TO 128-BLKSIZ DO
        FOR J FROM 0 BY BLKSIZ TO 128-BLKSIZ DO
         X:=I; Y:=J;
         MAX:=MIN:=P0;
  { test all pixels for equality }
         FOR X FROM I TO I+BLKSIZ-1 DO
          FOR Y FROM J TO J+BLKSIZ-1 DO
           MAX:=MAX?>P0;
           MIN:=MIN?<P0
                                     OD
                                    OD;
         C:= MAX-MIN > THRESH;
         R:=IF C THEN 0 ELSE (MAX+MIN)/2 FI;
         FOR X FROM I TO I+BLKSIZ-1 DO
          FOR Y FROM J TO J+BLKSIZ-1 DO
           Q0:=R; A0:=C
                                     OD
                                    OD
                                             OD
                                            OD

           { end of compiled section }   ;EXIT ]]
      RETURN;

NINTX:                   { Interpolate background in X,         }
      EDGEA:=TRUE;                  { Use linear interpolation }
      FOR Y FROM 0 TO 127 DO
       OK:=FALSE;
       FOR X FROM 0 TO 127 DO
        IF A0 THEN
               IF A5=0 THEN OK:=TRUE; DIST:=1; VAL:=Q5; XX:=X
                       ELSE DIST:=DIST+1 & DIST#0
               FI
              ELSE
              IF A5&OK THEN XXX:=X;DIST:=DIST+1;VAL2:=Q0;OK:=FALSE;
                              FOR X FROM XX TO XXX-1 DO
                              Q0:=(VAL2*(X-XX+1)+VAL*(XXX-X))/DIST;
                              A0:=FALSE
                                                     OD; X:=XXX
              FI
        FI
                            OD
                           OD
      RETURN;
```

```
NINTY:                   { Interpolate background in Y           }
     EDGEA:=TRUE;
     FOR X FROM 0 TO 127 DO
      OK:=FALSE;
      FOR Y FROM 0 TO 127 DO
       IF A0 THEN
              IF A3=0 THEN OK:=TRUE; DIST:=1; VAL:=Q3; YY:=Y
                      ELSE DIST:=DIST+1
              FI
             ELSE
             IF A3&OK THEN YYY:=Y;DIST:=DIST+1;VAL2:=Q0;OK:=FALSE;
                             FOR Y FROM YY TO YYY-1 DO
                             Q0:=(VAL2*(Y-YY+1)+VAL*(YYY-Y))/DIST;
                             A0:=FALSE
                                                    OD; Y:=YYY
             FI
        FI
                           OD
                          OD
      RETURN;

NINT: [[ @NINTX; @NINTY; EXIT ]] RETURN;{ Interpolate background }

OBJ:  { Discriminate the objects }
     OUT:=2; [[ R0:=(?+(P0-Q0) > THRESH/2 & A0=0)+((A0#0)&127) ]]
     RETURN;
```

DISTANCE FUNCTION PROPAGATION (Sequential Algorithm)

```
PROP: [[ Q0:=0 ]];  { initially all is 0 }
      [[+ IF P0
            THEN Q0:=(Q2?<Q3?<Q4?<Q5)+1 FI +]];
      [[- IF P0
            THEN Q0:=(Q1?<Q2?<Q3?<Q4?<Q5?<Q6?<Q7?<Q8)+1 FI -]]
      RETURN;
```

DISTANCE FUNCTION PROPAGATION (Parallel Algorithm)

```
PROP: [[ P0:=0 ]]; EDGEP:=0;  { initially distance function is 0 }
                              { A contains the binary image      }
      TO 64                   { 128/2 times is the maximum       }
       DO
        [[ Q0:=IF A0          { process into Q, for parallel     }
                THEN
                     (P0 ?< P1 ?< P2 ?< P3 ?< P4 ?<
                                   P5 ?< P6 ?< P7 ?< P8) + 1
                ELSE
                     P0
               FI
        ]];
        [[ P0:=Q0 ]]          { return to P for next run         }
       OD
      RETURN;
```

SKELETONISATION

```
SKEL: @PROP;     { Propagate the distance function }
      @MARK;     { Mark the local maxima }
      @SLIM5;    { Slim to a connected shape, 5x5 here or 3x3 }
      @THIN;     { Thin to unit width skeleton }
      @CLEAN     { Clean off noise spurs }
      RETURN;

PROP: [[ P0:=0 ]]; EDGEP:=0;  { initially distance function is 0 }
                              { A contains the binary image      }
      [[+ IF A0 THEN P0:= (P2 ?< P3 ?< P4 ?< P5)  +1  FI +]];
      [[- IF A0 THEN P0:= ((P1 ?< P2 ?< P3 ?< P4 ?< P5 ?<
                            P6 ?< P7 ?< P8)  +1) ?< P0    FI -]]
      RETURN;

MARK: [[ B0:=       { B set if the point is a local max ... }
             A0 &   { ... on the image                      }
             P0 >= ( P1 ?> P2 ?> P3 ?> P4 ?>
                         P5 ?> P6 ?> P7 ?> P8 )  ]]
      RETURN;

SLIM3:
    I:=1;
    WHILE
     F:=1;
      [[ C0:=IF ~B0&A0 THEN                                 { N }
         F:=F&(R:=~A7 ! A3 ! ~A1&A2 ! ~A5&A4); R ELSE A0 FI ]];
      [[ A0:=IF ~B0&C0 THEN                                 { S }
         F:=F&(R:=~C3 ! C7 ! ~C5&C6 ! ~C1&C8); R ELSE C0 FI ]];
      [[ C0:=IF ~B0&A0 THEN                                 { E }
         F:=F&(R:=~A1 ! A5 ! ~A3&A4 ! ~A7&A6); R ELSE A0 FI ]];
      [[ A0:=IF ~B0&C0 THEN                                 { W }
         F:=F&(R:=~C5 ! C1 ! ~C7&C8 ! ~C3&C2); R ELSE C0 FI ]];
      F=0
     DO
      I:=I+1
     OD;
    WRITE "(' 3x3 - ',I3,' Passes')",I
    RETURN;
```

```
SLIM5: EDGEA:=0;
      I:=1;
      WHILE
      FLAG:=1;
      [[ C0:=IF ~B0 & A0
          THEN

           NSZ:= (A4!~A5!A6) & (A2!~A1!A8);

           ZN:= ~A3&A7&(A22!B7) & ~(~A5&A4 ! ~A1&A2);
           ZNSW:= ~A6&A4 & (A18!~A19);
           ZNSE:= ~A8&A2 & (A10!~A9);

           ZS:= ~A7&A3&(A14!B3) & ~(~A5&A6 ! ~A1&A8);
           ZSNW:= ~A4&A6 & (A18!~A17);
           ZSNE:= ~A2&A8 & (A10!~A11);

           WEZ:= (A2!~A3!A4) & (A6!~A7!A8);

           ZW:= ~A5&A1&(A10!B1) & ~(~A3&A4 ! ~A7&A6);
           ZWEN:= ~A2&A4 & (A14!~A13);
           ZWES:= ~A8&A6 & (A22!~A23);

           ZE:= ~A1&A5&(A18!B5) & ~(~A3&A2 ! ~A7&A8);
           ZEWN:= ~A4&A2 & (A14!~A15);
           ZEWS:= ~A6&A8 & (A22!~A21);

           FLAG:=FLAG&(R:=
           ~(
             ( ~(ZNSW!ZNSE)&ZN ! ~(ZSNW!ZSNE)&ZS ) & NSZ !
             ( ~(ZWEN!ZWES)&ZW ! ~(ZEWN!ZEWS)&ZE ) & WEZ
            )
                      );R
          ELSE
              A0
             FI
        ]];
      [[ A0:=C0 ]];
      FLAG=0 DO
            I:=I+1
           OD;
    WRITE "(' 5x5 ',I3,' Passes')",I
    RETURN;

THIN: [[ C0:=A0 & ( @SIGMA=1 ! ~A7 ! A3 ! ~A1&A2 ! ~A5&A4 ) ]];
      [[ A0:=C0 ]];
      [[ C0:=A0 & ( @SIGMA=1 ! ~A3 ! A7 ! ~A5&A6 ! ~A1&A8 ) ]];
      [[ A0:=C0 ]];
      [[ C0:=A0 & ( @SIGMA=1 ! ~A1 ! A5 ! ~A3&A4 ! ~A7&A6 ) ]];
      [[ A0:=C0 ]];
      [[ C0:=A0 & ( @SIGMA=1 ! ~A5 ! A1 ! ~A7&A8 ! ~A3&A2 ) ]];
      [[ A0:=C0 ]]
      RETURN;

SIGMA: A1+A2+A3+A4+A5+A6+A7+A8  RETURN;
```

```
CLEAN: [[ Q0:=@SIGMA ]];
       [[ IF Q0=1 & Q1+Q2+Q3+Q4+Q5+Q6+Q7+Q8>2 THEN A0:=0 FI ]] RETURN;
```

Appendix B

An Example Terminal Session

A short annotated terminal session with the PPL2 sub-system is given to demonstrate the ease with which it can be used. A full description of the facilities available is given in appendix C, the users guide.

The text that appears on the terminal is shown on the left of the page and explanatory notes appear on the right within curly brackets.

```
.@PPL                                { Initiate the sub-system from RT11 }

FRIDAY        30-JUL-1982     10:52:43      { PPL gives the date and  }
                                            { time for record keeping }
Good morning. Welcome to PPL.
                                     { A warning that nothing          }
THERE ARE NO DEFINED VARIABLES       {    is defined (yet)             }
? .OUT:=0                            { Immediate command to display P  }
    0.27 SECONDS ELAPSED             { Time taken to execute           }
? PGET HOUSE,P                       { Get image from file HOUSE into P  }
THERE ARE NO DEFINED VARIABLES
? GET DEMO                           { Get a program from file DEMO      }
UNABLE TO OPEN FILE SY:DEMO.PPL      !  DOES IT EXIST ?
THERE ARE NO DEFINED VARIABLES       { It wasn't found                   }
? DIR DEMO                           { Directory to check                }
   0 FILES,     0 BLOCKS             { It isn't there, so ...            }
THERE ARE NO DEFINED VARIABLES
? NEW DEMO                           { Create a new file using the screen}
THERE ARE NO DEFINED VARIABLES       {  editor, then return              }
? LIST                               { List it to the terminal           }


DEM1: [[ PO:=255-PO ]] RETURN;

DEM2: [[ IF PO<127 THEN PO:=255-PO FI ]] RETURN;


THERE ARE NO DEFINED VARIABLES
? SAVE                               { Save it on disc                 }
AS SY :DEMO  .PPL ? Y                { PPL checks the file-name to use }
THERE ARE NO DEFINED VARIABLES
? @DEM1                              { Execute DEM1                    }
    0.65 SECONDS ELAPSED
? @DEM2                              { And DEM2                        }
    0.90 SECONDS ELAPSED
? PGET HOUSE,Q                       { Get the image into Q            }
? .DO OUT:=1-OUT; TO 40 DO SKIP OD OD    { compare it to P             }
```

```
PROGRAM STOPPED MANUALLY           { Break out by pressing STOP key    }
   31.07 SECONDS ELAPSED
? EDIT                             { Edit the program on screen        }
THERE ARE NO DEFINED VARIABLES     {  return from edit                 }
? @DEM2                            { Run DEM2 again                    }
    0.64 SECONDS ELAPSED
? SAVE                             { Save the new program              }
AS SY :DEMO  .PPL ? Y              { But it already exists, query it   }
SY:DEMO.PPL    ALREADY EXISTS.  DO YOU MEAN TO REPLACE IT ? Y
THERE ARE NO DEFINED VARIABLES     { Replaced                          }
? END                              { finish session, return to RT11    }

.                                  { RT11 prompt                       }
```

APPENDIX C

```
PPPP    PPPP    L        222
P   P   P   P   L       2   2
PPPP    PPPP    L          2
P       P       L         2
P       P       LLLLL   22222
```

USER GUIDE

BARRY M. COOK

PATTERN RECOGNITION GROUP

ROYAL HOLLOWAY COLLEGE

UNIVERSITY OF LONDON

OCTOBER 1981

CONTENTS

C.1 INTRODUCTION

The language PPL2 is intended as a versatile means of describing image-processing algorithms. In order to increase its usefulness a sub-system to create, modify and run programs written in PPL2 is provided. The sub-system is highly interactive to allow the rapid design and testing of image-processing algorithms: a number of special functions have been provided to facilitate this.

The aim of this manual is to describe the sub-system and PPL2 language in enough detail to allow a person to write and develop image-processing algorithms in PPL2.

C.2 ENVIRONMENT

The system to be described was first written at the beginning of 1981 and after a period of improvement is now considered a complete product.

The machine currently in use to run this system is a Digital Equipment Corporation PDP11/34A with 128K bytes of store and running the RT11XM V4.0 operating system.

C.3 STARTING UP

The PPL2 sub-system is started automatically at bootstrap. To start up at any other time the user should reply to RT11´s prompt (a full stop) with:

```
@PPL<cr>
```

C.4 FILENAMES

Filenames within the sub-system follow normal RT11 conventions, having a device name, file name and file type. When specifying a filename the user may give all three parts or only a subset of them; any part missing causes a default action:

Part	Default Action
Device name	SY: substituted (the device the system was bootstrapped from).
File name	The user is asked to supply a name.
File type	The tag .PPL is substituted for program files and .PIC for picture files.

C.5 KEYBOARD COMMANDS

When it is ready to accept a command from the user the sub-system will prompt the user by displaying a question mark and ringing the bell on the terminal. Any of the commands available on the system may then be entered; execution is initiated by pressing the RETURN key. Any command not recognised will produce an appropriate comment in response.

The purpose of ringing the bell as well as printing the question mark is to call the user back to the terminal if he was away waiting for a long program to complete.

The following commands are available in the system:

NEW	Create a new program.
GET	Obtain an existing program from disc.
EDIT	Edit the program.
SAVE	Save the program on disc.
LIST	List the current text.
PGET	Get a picture from disc.
PSAVE	Save a picture on disc.
PRINT	Print a picture on the terminal.
LOOK	Study a picture in detail.
OPT	List the options currently switched on.
ON	Switch options on.
OFF	Switch options off.
DIR	Directory of files on disc.
END	Return to RT11.
\	Print the values of all variables.
.	Indicates a PPL2 program to be executed.
@	Call a PPL2 sub-program to be executed.

C.5.1 NEW

This command clears out the text buffer and enters EDIT mode to enable the user to type in a new program. When the user is satisfied the END-EDIT key will return to the control loop.

The command may have one of two forms:

```
   NEW
or NEW <filename>
```

If a filename is given (second form) this will be remembered for future use in the SAVE command.

e.g. NEW FILE1 creates a new program SY:FILE1.PPL

C.5.2 GET

Files already existing on disc can be read into the text area by this command. It has two forms:

```
    GET
or  GET <filename>
```

e.g. GET FILE2.XYZ gets the file SY:FILE2.XYZ from disc.

C.5.3 EDIT

The editor is invoked by this command enabling the user to change his program. A large range of text manipulating commands is available on the keyboard and auxillary keypad; a complete list is given in section C.6.

C.5.4 SAVE

This command permits programs to be stored on disc for future use. It has two forms:

```
    SAVE
or  SAVE <filename>
```

The action in the first case will depend on whether a file name was previously given in a GET or NEW command; if it wasn´t then the user will be asked for a filename but if it was the user will be asked if this is the filename to be used by printing:

```
AS <filename>?
```

The user must reply "Y" for yes and any other character for no.

If a file of the same name as that specified already exists the user will be asked if he intends to overwrite it by printing:
<filename> ALREADY EXISTS. DO YOU MEAN TO REPLACE IT ?

The reply should be "Y" if he does and anything else if not. This action prevents the accidental destruction of files.
e.g. SAVE DL1:FILE3 saves the program as FILE3.PPL on disc DL1.

C.5.5 LIST

The program is printed on the terminal. There is only one form of this command:

```
    LIST
```

C.5.6 PGET

This is similar to GET, it gets a specified picture from disc into the video interface. It has three forms:

```
    PGET
or  PGET <filename>
or  PGET <filename>,<picture space>
```

In the first case the filename and picture space will be asked for; in the second case only the picture space will be asked for and in the third case the action will proceed without further ado. The picture space consists of a single letter from A, B, C, ..., Z.

e.g. PGET IMAGE,P gets picture SY:IMAGE.PIC into P.

C.5.7 PSAVE

This stores a picture from the video interface onto disc; like PGET it has three forms:

```
    PSAVE
```

or PSAVE <filename>

or PSAVE <filename>,<picture space>

In the same way as SAVE a check is made that no file of the same name already exists; if it does the user is asked if he wants to overwrite it.

C.5.8 PRINT

A picture is interpreted as ASCII characters and is printed on the terminal; only printable characters are output, all others are converted to spaces. There are two forms of this command:

PRINT

or PRINT <picture space>

In the first case the user is asked for the picture space to be printed.

C.5.9 LOOK

Pictures are displayed by LOOK on the VDU as numerical values. A 15 x 15 window on the picture is transferred to the VDU and displayed as an array of 3-digit octal values. The position of the window can be changed by the use of the joystick. Small movements of the stick move the window in 1-pixel increments and large movements of the stick in 8-pixel increments. Also displayed are the co-ordinates of the top left of the window, the picture number being displayed, the edge value of that picture and the mode (rectangular or hexagonal) of the interface.

Next to the joystick is a switch which, when raised, returns control to the PPL2 sub-system.

There are two versions of this command:

```
LOOK
or LOOK <picture space>
```

In the first version the user is asked for the picture space to be viewed.

C.5.10 OPT

Informs the user which options are currently switched on, there is only one form of this command:

```
OPT
```

The options which are controllable by the user are:

T When on prints the time taken by a program.

R When on displays the resultant value of the program, permitting use as a calculator.

C.5.11 ON

Used to switch options on, its form is:

```
ON <list of options to be switched on>
```

where the list of options is one or more of the key letters for the options.

e.g. ON T switches option T on.

C.5.12 OFF

Used to switch options off, its form is:

```
OFF <list of options to be switched off>
```

where the list of options is one or more of the key letters for the options.

e.g. OFF XYZ switches options X, Y and Z off.

C.5.13 DIR

Enables the user to print a directory of files existing on the machine; the user supplies details of the files to be listed in the form of a sample filename. Characters in the sample filename may be replaced by a percent sign to indicate that any character will fit, or by an asterisk to indicate that the rest of the field may be any character. Omission of a device name implies the device the system was booted from (SY:). Omission of the name or type field indicates that any name or type will do.

```
e.g.  DIR .PPL   prints all .PPL files on the bootstrap disc
      DIR        prints all files on SY:
      DIR DL1:   prints all files on DL1:
      DIR B*     prints all files beginning with the letter B
                 on the bootstrap disc.
```

C.5.14 END

This returns the user to RT11, removing the PPL2 sub-system from memory. To restart PPL2 the user should type:

```
@PPL<cr>
```

C.5.15 \

The names and values of all currently defined PPL2 variables are printed on the terminal, values are given in decimal (followed by a decimal point) and in octal. This is useful for error diagnosis. Note that getting a new file or editing the current one will invalidate the values of values.

C.5.16 .

A full stop as the first character on the line indicates that the rest of the line contains a PPL2 program that the user wishes to be executed. Immediate mode programs may be entered in this way.

e.g. .[[PO:=0]] { clears image P to 0 }

C.5.17 @

The ′at′ sign at the beginning of a line indicates that the characters following it are a PPL2 sub-program name which is to be immediately called and executed. This call may be followed by a semi-colon and further PPL2 program also to be executed in immediate mode. The sub-program called must exist in the text space or an error message will be printed.

C.6 THE EDITOR

The PPL2 sub-system provides a screen editor to allow rapid changes, additions and deletions to be made to the current program. It is invoked by typing EDIT<cr> in response to the system prompt ('?') and exited by using the END-EDIT key.

The special editing keys are described in the following pages.

Auxillary keypad editing keys:

```
+ - - - - - - - + - - - - - - - + - - - - - - - + - - - - - - - +
|               |               |               |               |
|  INSERT       |  DELETE       |   INSERT      |   DELETE      |
|  CHARACTER    |  CHARACTER    |   LINE        |   LINE        |
|               |               |               |               |
+ - - - - - - - + - - - - - - - + - - - - - - - + - - - - - - - +
|               |               |               |               |
|  START OF     |      UP       |      UP       |  DELETE TO    |
|  TEXT         |               |     PAGE      | END OF LINE   |
|               |               |               |               |
+ - - - - - - - + - - - - - - - + - - - - - - - + - - - - - - - +
|               |               |               |               |
|     LEFT      |  TOP LEFT     |   RIGHT       |    --N/U--    |
|               |  OF PAGE      |               |               |
|               |               |               |               |
+ - - - - - - - + - - - - - - - + - - - - - - - + - - - - - - - +
|               |               |               |               |
|     LAST      |     DOWN      |     DOWN      |               |
|     PAGE      |               |     PAGE      |               |
|               |               |               |    INSERT     |
+ - - - - - - - + - - - - - - - + - - - - - - - +    MODE       |
|                               |               |    OFF        |
|       INSERT MODE ON          |     END       |               |
|                               |     EDIT      |               |
|                               |               |               |
+ - - - - - - - - - - - - - - - + - - - - - - - + - - - - - - - +
```

Key	Function
INSERT CHARACTER	The text under and to the right of the cursor is moved one place to the right and a space generated for the insertion of one character. Any text falling off the right-hand end of the screen is lost.
DELETE CHARACTER	The character under the cursor is deleted and the text to its right is moved one place left. The far right character becomes a space.
INSERT LINE	A blank line is inserted above the line containing the cursor. The cursor is placed at the beginning of the new line.
DELETE LINE	The line containing the cursor is deleted.
START OF TEXT	The cursor is placed on the first character of the text.

Key	Function
TOP LEFT OF PAGE	The cursor is moved to the top left-hand corner of the screen.
LAST PAGE	The last 24 lines of text are displayed. The cursor is positioned on the top left-hand corner of the screen.
DELETE TO END OF LINE	Delete the character under the cursor and all characters on the line to its right.
DOWN PAGE	Move down 24 lines, the line at the top of the screen is the line that was just below the bottom of the screen. The cursor is positioned at the top left-hand corner of the screen.
UP PAGE	Move up 24 lines, the line at the bottom of the screen is the line that was just above the top of the screen. The cursor is positioned at the top left-hand corner of the screen.

Key	Function
UP	Move the cursor up one line.
DOWN	Move the cursor down one line.
LEFT	Move the cursor one place to the left; if the cursor moves off the left-hand edge of the screen it will appear at the end of the previous line.
RIGHT	Move the cursor one place to the right; if the cursor moves off the right-hand edge of the screen it will appear at the beginning of the next line.
INSERT MODE ON	Switch the insert mode on. In this mode all text typed is inserted before the cursor; if necessary the line is at a suitable word boundary.
INSERT MODE OFF	Switch off the insert mode.

N.B. Any editing key except DELETE and INSERT MODE ON will switch off the insert mode.

Key	Function
END EDIT	Finish editing and return to the main PPL2 control program.

Main keyboard editing keys:

```
< < < < <  - - - - + - - - - - - + - - - - - - +
                   |    LOCAL    |    LOCAL    |
 etc.              |    FORM     |    LINE     |
                   |    FEED     |    FEED     |
                   |             |             |
< < < < <  - - - - + - - - - - - + - - - - - - + - - - - - - +
                   |             |             |             |
 etc.              |     ~       |   REPEAT    |    BREAK    |
                   |     `       |             |             |
                   |             |             |             |
< < < < <  - - - + + - - - - + - + - - - + - - + - - + - - - +
                 |           |           |           |
 etc.            |    }      |           |   DELETE  |
                 |    ]      |           |           |
                 |           |           |           |
< < < < <  - - - + + - - - - +           + - - - - - +
                   |                     |           |
 etc.              |      RETURN         |     |     |
                   |                     |     \     |
                   |                     |           |
< < < < <  + - - - + - - - - - - - + - - + - - - + - +
           |                       |             |
 etc.      |      SHIFT            |   LINE      |
           |                       |   FEED      |
           |                       |             |
< < < < <  + - - - - - - - - - - - + - - - - - - +
```

Key	Function
LINE FEED	Move the cursor down one line.
REPEAT	Repeat the last cursor move, this key has auto-repeat if held down.
DELETE	In normal edit mode this key moves the cursor left one place. If insert mode is on, the last character typed is deleted.
RETURN	Moves the cursor to the beginning of the next line. (in effect, carriage-return and line-feed).

All other keys have the labelled effect. A character key places that character at the cursor position; if the cursor is at the right-hand end of the screen it will not move further and characters overwrite each other.

C.7 THE LANGUAGE PPL2

This section of the user guide describes the language PPL2 as implemented on the PDP11/34A in the pattern recognition group at the Royal Holloway College and includes notes on the special features found at this establishment.

This document is not intended to be a text book but merely a reference for the programmer who is already competent in another language and wishes to learn what PPL2 has to offer.

C.7.1 Comments

Comments in PPL2 are enclosed within curly brackets. These brackets must not be nested. The comment may extend over more than one line if required. Any characters except curly brackets may be used in a comment.

e.g. { this is a comment in PPL2 }

C.7.2 Constants

Constants may be decimal, octal or the ASCII value of a character.

A decimal constant is a string of digits, each one in the range 0 to 9.

e.g. 3906

An octal constant is a string of digits, each one in the range 0 to 7, preceeded by the up-arrow character (shift-6).

e.g. ^377

A character constant is any character enclosed within prime symbols (´); The 7-bit ASCII value of the character is used.

e.g. ´A´

C.7.3 Variables

Variable names may be between one and six characters long, the first character must be a letter and the others may be letters or digits. Only upper-case letters may be used. The name generated may not be the same as a reserved word. Special care should be taken not to use names that may be confused with picture-points, i.e. one letter followed by digits.

C.7.4 Limits of Constants and Variables

Values are stored within the machine as 16-bit integers, the top bit is used as a sign bit; the range of values that may be stored is from -32767 to 32767. Care should be taken not to exceed this range as no checks are made; it is particularly easy to exceed this limit when summing a number of picture points (128 points should be taken as an upper limit when 8-bit picture values are used). The result of overflow is undefined.

Within the video interface there are both byte and bit picture planes. The bit planes hold only the values 0 and 1; if a value outside this range is written to the interface only the least significant bit of the value is stored. The byte planes contain 8 bits holding values from 0 to 255, always positive; only the 8 least significant bits of a value sent to the interface are stored. (Future hardware will permit storage of 16-bit signed integer values).

C.7.5 Logical Values

When generated by one of the comparison operators (=,#,>,<,>=,<=) logical FALSE is returned as a word of zero bits (0) and logical TRUE as a word of one bits (-1).

A conditional statement tests for zero to indicate FALSE and non-zero to indicate TRUE.

C.7.6 Reserved Words

Certain words have special meanings within PPL2 and may only be used as intended and not as variable names. A list of the reserved words and their meanings is given in table C.1.

C.7.7 Assignment

Variables are assigned values by using the assignment operator ":=", they must be assigned a value produced by a unitary clause.

e.g. TIGGER:=6

C.7.8 Other Operators

See table C.2 for a complete list of operators available.

X Y IN OUT EDGEA to EDGEZ MODE
RGB256 JSTICK RETURN MODULO IF THEN ELSF
ELSE FI FOR FROM BY TO WHILE
DO OD MDC512 READ WRITE GOTO ARRAY
ARRAY0 ARRAY1 ARRAY2 ARRAY3 TRUE FALSE
Single letter followed by digits = Picture-point

X	The X co-ordinate of the centre of the window.
Y	The Y co-ordinate of the centre of the window.
IN	The number of the picture to be read from the camera (0 <= IN <= 15).
OUT	The number of the picture to be displayed (0 <= OUT <= 15).
EDGEA to EDGEZ	The edge value of the picture.
MODE	Interface mode, 0=Rectangular, 1=Hexagonal.
RGB256	256 x 256 x 8 bits video output.
JSTICK	Joystick.
RETURN	Return from sub-program to calling program.
MODULO	A MODULO B = remainder after dividing A by B.
IF	Part of conditional statement.
THEN	...ditto...
ELSF	...ditto...
ELSE	...ditto...
FI	...ditto...
FOR	Part of looping statement.
FROM	...ditto...
BY	...ditto...
TO	...ditto...
WHILE	...ditto...
DO	...ditto...
OD	...ditto...
MDC512	512 x 512 binary video output.
READ	Read statement.
WRITE	Write statement.
GOTO	Unconditional branch.
ARRAY	A 1024 word array overlaying ARRAY0 to ARRAY3
ARRAY0	A 256 word array (subscript range 0 to 255)
ARRAY1	...ditto...
ARRAY2	...ditto...
ARRAY3	...ditto...
TRUE	Logical .TRUE. (-1)
FALSE	Logical .FALSE. (0)

C.1 Reserved words

Op	Priority	Description
~	12	.NOT. Unary op.
-	12	Negate Unary op.
?+	12	Abs Unary op ?+ A means IF A < 0 THEN -A ELSE A FI
?¦	12	Range Unary op ?¦ A means IF A < 0 THEN 0 ELSF A > 255 THEN 255 ELSE A FI
*	11	Multiply
/	11	Divide
+	10	Add
-	10	Subtract
?>	9	Max A ?> B means IF A > B THEN A ELSE B FI
?<	9	Min A ?< B means IF A < B THEN A ELSE B FI
=	8	Equals A = B means IF A = B THEN -1 ELSE 0 FI
#	8	Not equal { similarly }
>	8	Greater than { similarly }
<	8	Less than { similarly }
>=	8	Greater than or equal { similarly }
<=	8	Less than or equal { similarly }
&	7	Bit by bit logical AND of operands
!	6	Bit by bit logical OR of operands
$	5	Bit by bit logical EXCLUSIVE OR of operands
:=	4	Assign This is executed from right to left
,	3	List separator (for READ and WRITE)
;	2	Statement separator

Operators are executed in priority order; if two operators of equal priority appear they will be executed from left to right (except :=).

The order of execution may be altered by the use of round brackets.

E.g. A+B*C first multiplies B by C and then adds A
(A+B)*C first adds A to B and then multiplies by C

C.2 Operators

C.7.9 Unitary and Serial Clauses

Each self-contained program step is known as a unitary clause, an example being an assignment statement. A program may be as short as one unitary clause or may consist of a sequence of such clauses separated by semi-colons. Two or more unitary clauses separated by semi-colons are known as a serial clause, i.e. unitary clauses to be performed one after another.

Examples:

Unitary clause: Z:=67

Serial clause: A:=5; B:=9; C:=456

If a unitary clause is required where a number of steps are to be performed they may be enclosed within round brackets to form a self-contained program step which is a unitary clause.

Example:

Serial clause: VAR:=56; VAR2:=67

Unitary clause: (VAR:=56; VAR2:=67)

Notice that there is no semi-colon before the closing bracket as there is no statement needing to be separated.

C.7.10 Results delivered by Unitary and Serial Clauses

A unitary clause may or may not return a result; those in which the last thing performed generates a value return that value whilst those which do not do so return no value.

Examples - unitary clauses returning a value:

```
5
A+7
JOHN+JEAN*3
A:=B
IF ... THEN <value> ELSE <value> FI
GEORGE
```

Examples - unitary clauses returning no value:

```
DO ... OD

FOR ... DO ... OD

IF ... THEN ... FI

IF ... THEN <no value> ELSE ... FI

IF ... THEN ... ELSE <no value> FI
```

The result of a serial clause is the result of the last unitary clause within it to be performed.

Examples:

```
A:=5; B:=8           returns the value 8

A:=2; B:=3; A+B      returns the value 5

READ A; A>6          returns either TRUE or FALSE depending on
                       the value read for A
```

It is thus possible to meaningfully write such statements as:

```
EEYORE:= IF A>B THEN 3 ELSE 7 FI

IF READ A; A=3 THEN ... ELSE ... FI
```

C.7.11 Program Sequencing - IF Statements

The IF statement allows decisions to be made at run time about which part of the program to execute next; such decisions depend upon values generated during the course of the run.

The statement starts with the word IF and ends with the word FI; there must also be a THEN part, other parts being optional unless a value is required to be returned; in such cases an ELSE part is also required.

The general form is:

The first two parts must always both occur.

```
  IF <clause returning a value>
THEN <clause, a value need only be returned if the
        whole statement is expected to return a value>
```

The next two parts are optional; if required they must both occur.

```
ELSF <clause returning a value>
THEN <clause, as above after THEN>
```

This part is optional.

```
ELSE <clause, as above after THEN>
       this part must occur if a result is to be
        returned by the IF-statement
```

The final part must occur

```
  FI
```

Note - the result of the clause after IF or ELSF is interpreted as logical FALSE if it has value zero and TRUE if non-zero.

Examples:

```
IF X=6 THEN J:=8 ELSE N:=N+1 FI
IF Z>3 THEN S:=6 FI
```

```
  IF A=0 THEN B:=8
ELSF A=1 THEN J:=J-9
ELSF A=5 THEN C:=0
         ELSE Z:=88
 FI
```

C.7.12 Program Looping - DO Statements

All parts of the DO statement except DO and OD are optional, default actions being taken on their omission. Those parts which do occur must do so in the order set out below.

The general form is:

```
   FOR <variable name>
  FROM <clause returning a value, the initial value>
    BY <clause returning a value, the step size>
    TO <clause returning a value, the final value>
 WHILE <clause returning a value, the continuation condition>
    DO <clause, the part to be repeated>
    OD
```

The step size may be negative in which case the final value will be less than the initial value.

The step size, initial and final values are evaluated once only; at the beginning of the loop, they cannot be altered from within the loop.

Default actions:

Part	Default	Comments
FOR	-none-	There is no control variable
FROM	1	
BY	1	
TO	infinity	There is no final value
WHILE	-none-	No condition is tested
DO	MUST NOT BE OMITTED	
OD	MUST NOT BE OMITTED	

Examples:

```
FOR A FROM 5 TO 20 DO ... OD

TO 50 DO ... OD

WHILE Z=T DO ... OD

FOR I FROM 0 BY 5 TO 300 WHILE X#5 DO ... OD
```

The PASCAL construction:

```
REPEAT <action> UNTIL <condition>;
```

can be written in PPL2 as:

```
WHILE <action>; ~<condition> DO SKIP OD;
```

Note that the condition is negated as a TRUE is needed to continue as opposed to PASCAL's TRUE to stop and also that a dummy statement is used between DO and OD; anything would do here but SKIP is fairly descriptive and the normal word in ALGOL68.

C.7.13 Sub-programs

The sub-program in PPL2 is very similar to a macro in assemblers in that a section of text is marked and then pulled into a program when required. This generates longer but faster object code than calling and returning from genuine sub-routines; a vital consideration in image processing

The start of the text section is given a label and the end is marked by a RETURN statement. A label is simply a standard variable name followed by a colon.

Example:

```
TEXT: {this is the section of text to be treated as
       a sub-program. When it is finished we} RETURN;
```

To include this program section we write:

```
@TEXT
```

and all the text between the colon of the label and the R of RETURN is substituted for @TEXT.

C.7.14 Arrays

At present there is available either a single array of 1024 words or four arrays of 256 words each. They are named:

ARRAY	for the 1024 word array
ARRAY0	for a 256 word array
ARRAY1	...ditto...
ARRAY2	...ditto...
ARRAY3	...ditto...

Elements of the arrays are accessed by indexing; the index values for the 1024 word array ARRAY are 0 to 1023; the range for the 256 word arrays ARRAY0 to ARRAY3 are 0 to 255.

C.7.15 Indexing

To access elements of an array we must give the name of the array and an index value to specify the offset into the array. The array is assumed to be consecutive words in memory and the index value simply indicates the offset from the first word. An index value is given by enclosing it in single square brackets after the array name. The item within the index brackets may be a unitary or serial clause provided it returns a value.

i.e.

ARRAY [34]

Picture points may also be indexed - see section C.7.16

C.7.16 Picture Spaces

At present there are eight picture areas available on the video interface These consist of (a) four spaces containing 8 bits per pixel known as byte spaces and referred to as P, Q, R and S and (b) four spaces having only one bit per pixel called bit spaces and referred to as A, B, C and D.

The centre points are A0, B0, C0, D0, P0, Q0, R0 and S0; the neighbours are, for example, P1, P2, etc.. A complete list of neighbour numbers is given in table C.3.

The elements of the window may be considered as an array of values and indexing used to access them.

P0 is the same as P0 [0]

P1 is the same as P0 [1]

Pn is the same as P0 [n]

C.7.17 Picture Operations

Window operators are enclosed within pairs of square brackets to indicate that they should be performed over the whole picture:

forward raster scan:

```
    [[  <window operator>  ]]
or  [[+ <window operator> +]]
```

reverse raster scan:

```
    [[- <window operator> -]].
```

A parallel operation may be effected by ensuring that the

Table C.3
RECTANGULAR ARRAY 15x15 WINDOW POINT NUMBERING

```
 X: -7   -6   -5   -4   -3   -2   -1    0    1    2    3    4    5    6    7
 Y
-7: 196  195  194  193  192  191  190  189  188  187  186  185  184  183  182 -7

-6: 197  144  143  142  141  140  139  138  137  136  135  134  133  132  181 -6

-5: 198  145  100   99   98   97   96   95   94   93   92   91   90  131  180 -5

-4: 199  146  101   64   63   62   61   60   59   58   57   56   89  130  179 -4

-3: 200  147  102   65   36   35   34   33   32   31   30   55   88  129  178 -3

-2: 201  148  103   66   37   16   15   14   13   12   29   54   87  128  177 -2

-1: 202  149  104   67   38   17    4    3    2   11   28   53   86  127  176 -1

 0: 203  150  105   68   39   18    5    0    1   10   27   52   85  126  175  0

 1: 204  151  106   69   40   19    6    7    8    9   26   51   84  125  174  1

 2: 205  152  107   70   41   20   21   22   23   24   25   50   83  124  173  2

 3: 206  153  108   71   42   43   44   45   46   47   48   49   82  123  172  3

 4: 207  154  109   72   73   74   75   76   77   78   79   80   81  122  171  4

 5: 208  155  110  111  112  113  114  115  116  117  118  119  120  121  170  5

 6: 209  156  157  158  159  160  161  162  163  164  165  166  167  168  169  6

 7: 210  211  212  213  214  215  216  217  218  219  220  221  222  223  224  7

    -7   -6   -5   -4   -3   -2   -1    0    1    2    3    4    5    6    7

RING        X,Y       POINTS     NO.POINTS  CUM.NO.  NO.OUTSIDE
NO.     FROM  TO    FROM  TO               POINTS

0          0   0       0   0        1         1          224
1         -1   1       1   8        8         9          216
2         -2   2       9  24       16        25          200
3         -3   3      25  48       24        49          176
4         -4   4      49  80       32        81          144
5         -5   5      81 120       40       121          104
6         -6   6     121 168       48       169           56
7         -7   7     169 224       56       225            0
```

result of an operator does not corrupt the input picture space i.e. the result is put into a picture space which is not used for input data.

C.7.18 GOTO

Control may be passed to labelled parts of the program by the use of GOTO. This statement is usually frowned upon by the advocates of structured programming and should be used sparingly. The only occasion when it is notably useful is to deal with error conditions as the quick way out of a program.

Example of its use:

```
................ GOTO HELP; ......................
HELP: {error routine}
```

C.7.19 EXIT

This enables a picture scan to be terminated, control passing to the statement after the closing double square brackets.

It is also useful for forcing compilation of a section of code; the section is enclosed within picture scan brackets but only one execution allowed by using EXIT to prevent repetition.

Example:

```
[[ FOR I TO 10000 DO ... OD; EXIT ]].
```

C.7.20 Input and Output

Information may be exchanged between the program and the terminal by the use of the READ and WRITE statements. I/O may be formatted or free format, free format is useful for input from the terminal while formatting is useful for output.

Free format read:

```
      READ A,B,C;
```

Formatted read:

```
      READ "<FORTRAN format including ( and )>",A,B,C,D;
```

as in READ "(2I6)",I,J;

Free format write:

```
      WRITE A,B,3+5*C,D+H,E;
```

Formatted write:

```
      WRITE "(´ The values of X and Y are -´,2O3)",X,Y;
or    WRITE "(´ Give the value for JOHN´,$)";
```

Note that there must be a variable list for read statements and unformatted writes but formatted writes need have none. The output list in a write statement may include clauses returning a value but that in a read must be variable names. Formats may not continue past the end of the line.

C.7.21 Spaces in the Source Program

Usually spaces are ignored and may or may not occur as the programmer sees fit; they may thus be used to aid the clear setting out of the program. There are some places where they must occur and some where they must not:

Spaces MUST occur:

To separate keywords from variable names.

Spaces MUST NOT occur:

Within a keyword or variable name,

Within a decimal or octal constant,

Within the primes of an ASCII constant, unless it is the character required,

Between the ^ and digits of an octal number, Between the @ symbol and sub-program name,

Between the characters of a multi-character symbol

(e.g. [[+ ?> <= etc.).

C.7.22 Construction Occurrence

Some of the constructions in PPL2 have restrictions on the positions in a program at which they may occur. Table C.4 details these restrictions.

	WITHIN [[...]]	OUTSIDE [[...]]
IF ... FI	YES	YES
FOR ... DO ... OD	YES	YES
Arrays	YES	YES
Operators in table 2	YES	YES
RETURN	NO	YES
Sub-program call (@)	YES	YES
READ	NO	YES
WRITE	NO	YES
GOTO	NO	YES
Labels	NO	YES
Indexing	YES	YES

Table C.4 - Construction Occurrence

C.7.23 Example Programs

```
{ MEDIAN FILTER - replace centre point with median value of centre and
                  immediate neighbours    }

MEDIAN: [[ FOR I FROM 0 TO 8 DO ARRAY[I]:=P0[I] OD;
           FOR I FROM 7 BY -1 TO 3 DO
           FOR J FROM 0 TO I DO
            IF ARRAY[J] > ARRAY[J+1] THEN A         :=ARRAY[J];
                                          ARRAY[J]  :=ARRAY[J+1];
                                          ARRAY[J+1]:=A                FI
            OD OD;
           Q0:=ARRAY[4]     ]] RETURN;



{ ROBERTS CROSS OPERATOR.
  From: Digital Image Processing by Pratt: P487  Eqn. 17.4-11       }

ROBERT: [[ Q0:= ?¦( ?+(P0-P2) + ?+(P1-P3) ) ]] RETURN;


{ SOBEL EDGE ENHANCEMENT.
  From: Digital Image Processing by Pratt: P489  Eqn. 17.4-13a,b,c
        Replacing squared + square roots by absolute values

     N.B. notation difference

         A0    A1    A2                P4    P3    P2

         A7  F(j,k)  A3    compare to  P5    P0    P1

         A6    A5    A4                P6    P7    P8             }

SOBEL: [[ EXE:= (P2+2*P1+P8) - (P4+2*P5+P6);
          WHY:= (P4+2*P3+P2) - (P6+2*P7+P8);
          Q0 := ?¦( ?+EXE + ?+WHY )   ]]  RETURN;



{ Generate a picture of random values.
  Using a 15 bit pseudo-random number generator   }

{ The generator          }

RAND: R:=R/2 ! (R&1 $ (R&2)/2 ) * ^40000 RETURN;

{ Fill picture space P with random values  }

RP: IF R=0 THEN R:=1 FI;      { Generator will not start if R=0 }
    [[ P0:=@RAND ]] RETURN;
```

C.8 SPECIAL HARDWARE

Any memory location in the PDP11 may be accessed from a PPL2 program enabling special hardware to be used. The commonly used special hardware has been given special names in PPL2 to facilitate access.

The general way to access a particular memory location uses the address calculation of an index to achieve the desired result. The following statement is used:

<constant> [<expression>]

<constant> can only be that, not a variable; it is the base
address in the machine in bytes.
Since word addressing is used this value must be even.

<expression> can be any clause returning a value, it is the
offset from the base in words (=2 bytes).

So the address referred to above is:

<constant> + 2*<expression> bytes.

C.8.1 JOYSTK

The variable JOYSTK returns a value which indicates the position of the joystick which is situated on the rack to the left of the terminal.

In the central position the value is 48 to which is added:

1 if the stick is pushed UP
2 " " " " " DOWN
4 " " " " " RIGHT
8 " " " " " LEFT
-16 " " " " " hard over UP or DOWN
-32 " " " " " " " LEFT or RIGHT
64 if the switch to the left of the stick is raised

C.8.2 RGB256

RGB256 is the variable name used to access the registers of the URGB-256 image board. This board provides a display of 256 x 256 pixels at 8-bit resolution in either grey scale or colour.

RGB256 [0] is the data/control register

RGB256 [1] is the Y and X register

RGB256 [2] is the scroll register

For further details see the MATROX description of this board.

Bit	Octal	Decimal	Colour controlled
7	200	128	Most significant GREEN
6	100	64	Most significant RED
5	40	32	Most significant BLUE
4	20	16	Next significant GREEN
3	10	8	Next significant RED
2	4	4	Next significant BLUE
1	2	2	Least significant GREEN
0	1	1	Least significant RED
	Uncontrolled		Least significant BLUE

C.8.3 MDC512

MDC512 is the variable name used to access the registers of the MDC-512 image board. This board provides a display of 512 x 512 pixels at single bit resolution, it is used for graphs.

MDC512 [0] is the data/control register

MDC512 [1] is the X register (contains X*2)

MDC512 [2] is the Y register

MDC512 [3] is the scroll/flag register

For further details see the MATROX description of this board.

C.8.4 Example programs

```
{ Transfer the four picture spaces P, Q, R and S from the
  video interface to the 256 x 256 display to see them
  all at the same time.                                    }

D: [[ RGB256[1]:= Y       *256 !  X;       RGB256:=P0;
      RGB256[1]:= Y       *256 ! (X+128); RGB256:=Q0 ]];
   [[ RGB256[1]:=(Y+128)*256 !  X;       RGB256:=R0;
      RGB256[1]:=(Y+128)*256 ! (X+128); RGB256:=S0 ]] RETURN;



{ 256 x 256 board.
  Draw a circle Centre (128,128), Radius R, Colour C ;
  F does not ask for R,C                                }

{ Ask for R and C     }

E: WRITE "(′ RADIUS ?′,$)"; READ R;
   WRITE "(′ COLOUR ?′,$)"; READ C;

{ Entry if R,C not asked for, e.g. call from program    }

F: [[ FOR YY FROM 127-R TO 129+R DO
      FOR XX FROM 127-R TO 129+R DO
      RGB256[1]:=YY*256!XX;
    { is the point in the circle ? If so set its colour  }
      IF ((XX-128)*(XX-128)+
          (YY-128)*(YY-128))  <=  R*R THEN RGB256:=C FI
      OD
      OD;  EXIT ]]  RETURN;
```

```
{ Transfer the picture space P as a perspective view in three
  dimensions to the 512 x 512 board, X and Y axes as the
  original and Z axis as the brightness of the original.
  Hidden surfaces are not seen.                            }

{ Clear the screen }

CLR: MDC512:=^100000;
     WHILE (MDC512[3]&^200)=0 DO SKIP OD
      RETURN;

{ Draw the graph }

ZZ: MDC512[3]:=0;
    @CLR;
    [[ FOR I FROM 0 TO 1023 DO ARRAY[I]:=2000 OD; EXIT ]];
    [[- MDC512[1]:=XX:=(X+Y)*4;
        YY:=2*Y+64-P0/4;
        IF YY<ARRAY[XX] THEN ARRAY[XX]:=YY;
                             MDC512[2]:=YY;
                             MDC512:=1;
                             MDC512[1]:=XX-2;
                             MDC512:=1
        FI
                                                -]] RETURN;
```