AN INTERACTIVE PICTURE MANIPULATION SYSTEM

by

Daniel Lewis Franklin

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREES OF

BACHELOR OF SCIENCE

and

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September, 1978

Signature of Author ....
Department of Electrical Engineering and
Computer Science, August 11, 1978

Certified by ....
Thesis Supervisor (MIT)

Certified by ....
Company Supervisor (Bell Labs)

Accepted by ....
Chairman, Departmental Committee on Graduate Students

An Interactive Picture Manipulation System

by

Daniel Lewis Franklin

Submitted to the Department of

Electrical Engineering and Computer Science

on August 11, 1978 in partial fulfillment of the requirements

for the Degrees of Master of Science and Bachelor of Science

ABSTRACT

A system for manipulating scanned picture files is described. The system features an interactive language, PIPO, for quickly and efficiently performing local, position-invariant operations. There are also commands for (inter alia) filling in an arbitrary closed curve, translating a picture file into a file of phototypesetter commands, and compressing a picture file into a compact format suitable for long-term storage. The system employs an extensible command language which permits the user to easily define his own commands in terms of those already existing. Its modular design also aids in the writing of entirely new commands.

Donald E. Troxel

Associate Professor, Dept. of Electrical Engineering

ACKNOWLEDGEMENT

I wish to thank my supervisor at Bell Labs, Dr. Peter Denes, for providing helpful criticism and a guiding hand; Prof. Troxel, my thesis advisor, for his advice; Susan Patterson of Bell Labs, for all her help both during my stay at Bell and afterwards; and Gary Madison and Mark Levine, for providing moral and financial support. I also wish to thank Jerry Roberts of ECD Corp, for allowing me to use ECD's SMART-ASCII terminal, without which this thesis would probably still have gotten done, but not without a great deal more pain. Finally, I thank the Student Information Processing Board of M.I.T. for funding computer time for the preparation of this thesis.

This thesis was prepared using the Honeywell Multics Word-Processing System.

CONTENTS

Chapter I

Introduction

Computers have been used to generate and display pictures ever since the first line printer was used to make pinups. It is only within the past few years, however, that they have begun to be employed widely in commercial picture production. Similarly, typesetters have traditionally been used only to produce text: pictures were handled by a combination of photography and pasteups, and only recently has there been a move towards processing pictures by computer control of the typesetters themselves.

An example of this changeover is Yellow Pages directory production. Its long-term goal is a system in which the entire directory, pictures as well as text, is on line and capable of being printed on a CRT phototypesetter in a single pass and shipped to the printer.

In Yellow Pages production, there are many examples of picture manipulation carried out by time-consuming photographic techniques. The directory customer has a wide variety of picture formats at his disposal, which are prepared to his specifications by the telephone company's commercial artists. These pictures can contain gray areas of several shades. The process of adding these areas is referred to as "screening", since it involves the use of a screen of dots to simulate gray scale. Unusual typography is also available. A commercial artist can be requested to do custom lettering, or a conventional font can be used with one or more modifications, such as outlining, "drop-shadowing" (adding shadows to the letters to make them look like three-dimensional block letters), and "haloing" (adding haloes around outlines of the letters).

Although advertisements containing all these features can theoretically be produced by CRT typesetters, the required software does not exist. Hence these operations are currently performed using photography, manual processing, and pasteups: a time-consuming process.

To automate this work requires a system capable of the following:

(1) Scanning and digitizing pictures drawn by a commercial artist to make the pictures available for computer processing, and scaling them to fit. (The pictures will be drawn off-line for convenience, and later digitized using the scanner.)

(2) Performing modifications and additions on those pictures, including combining them and screening portions of them.

(3) Generating special lettering.

(4) Producing a file of phototypesetter commands which reproduces the completed ads.

A system meeting these requirements is now under development. The system will run on a NOVA 830 16-bit minicomputer with 48K words running mapped RDOS. The hardware includes three Diablo moving head disks, a 9-track tape drive, a graphics tablet, a Dest Data digital scanner, a 512 x 512 color TV display, a Versatek printer/plotter, and a Tektronix 4010 console. To aid in its development, another system was needed which would provide an environment suitable for command development. The minicomputer itself was not particularly hospitable, as it was slow and lacked memory. Once a command for, e.g., drop-shadowing, was developed, it could be recoded in NOVA assembler for efficiency; but for testing purposes, an environment in which one was not penalized very much for inefficiency was highly desirable.

The problem, then, was to develop picture-processing operations for a commercial page-layout system, within the framework of a general interactive picture manipulation system. The operations to be developed included screening, drop-shadowing, haloing, and outlining. The Interactive Picture Manipulation System (IPMS) was the result.

IPMS is implemented using the following equipment:

1) An SEL 86 computer (600 nsec memory cycle time) with 40K 32-bit words of core, hardware floating point, and two fixed head disks;

2) A raster-scanned color television display with 512 X 508 picture elements, each up to 4 bits long, and stored in a digital refresh memory; the red, blue, and green video signals are generated automatically by accessing the refresh memory;

3) A Dest Data scanner-digitizer, capable of scanning an 8 1/2" x 11" black and white picture (no gray scale), digitizing at a density of 240 lines/inch, in about 5 seconds;

4) An 11" x 11" tablet and stylus (Computek GT50/10) with 1K x 1K resolution;

5) An alphanumeric keyboard, a Tektronix 611 storage scope, and a Tektronix 4601 Hard Copy Unit, used as the system console;

6) A PDP-11/45 running MERT/UNIX and attached through high-speed channels to the SEL 86. It provides a file system for the SEL (using several 65 megabyte moving head disks) and timesharing services for editing programs to be run on the SEL. Only the SEL, however, operates the interactive displays.

IPMS includes a language (PIPO) for describing a wide class of local, position-invariant picture operations. As defined by Rosenfeld,(1) a "position-invariant" picture operation is one in which the effect of the operation on a point does not depend on its position within the picture; i.e., it is a function only of the values of selected pixels, not their coordinates. Position-invariant picture functions are thus analogous to time-invariant functions in conventional signal processing. A "local" picture function is one which, to the extent that it is a function of input pixel values, only depends on those within a finite area.

The class of local, position-invariant picture operations is a large one, and it includes most useful picture-processing operations. For example, averaging, quantizing, and edge detection are all local, position invariant operations. Thus, a system which provides the ability to specify almost any such operation should be quite useful.

The usefulness of PIPO would be limited, however, were it not for its high efficiency. (A 512 by 512 TV picture contains approximately a quarter of a million points. An operation on all of them will take 0.15 seconds longer for each additional SEL machine cycle. They add up rather quickly.) Equivalent efficiency could be obtained by programming the function in

---

(1) Rosenfeld, A. Picture Processing by Computer, University of Maryland Computer Science Center Report TR-71, June 1968, Contract Nonr-5144(00), p. 1-11.

assembler, but such programming would be tedious and anything but interactive, requiring a complete assemble-link-load cycle for each change in the program. A complete cycle would also be required in order to adapt the code for a different number of pixels per line (if the code were to be as efficient as PIPO). The programming could be made less tedious by using FORTRAN, but at a great cost in execution time. Furthermore, a complete compile-link-load cycle would still be required for changes in the algorithm. PIPO avoids these problems by providing a compile-and-go translator. The language is simple, and heavily influenced by the instruction set of the machine, but is still a great improvement over the alternatives.

Other Picture Manipulation Systems

Graphical systems can be divided into two general types: those which work in terms of shapes (points, lines, curves, etc.) and those which work with scanned pictures (pictures stored as a two-dimensional collection of points). Historically, systems which work with scanned pictures have not offered nearly as many operations as the shape-drawing systems, nor have they been easy to augment with new operations.

An early example of a picture-generation system is SKETCHPAD,(1) which permitted the definition of complex line

(1) Sutherland, I. E.
SKETCHPAD, A Man - Machine Graphical Communication System.
Ph. D. Diss., Dept. of Electrical Engineering, MIT, Cambridge, Mass. (1963)

drawings using a light pen and vector CRT display. Objects, once drawn, could be readily replicated. Their shapes could be made dependent on "constraints," permitting, for example, demonstrations of lever mechanisms. However, SKETCHPAD was limited to line drawings entered at the CRT.

More recently, there have been animation systems such as that of Catmull,(1) which will generate halftone cartoons of arbitrary complexity from an easily understood set of commands. Designed for cartoonists with little or no computer experience, it only manipulates picture features; picture processing is not implemented.

There are also a wide range of graphics packages, of which the most popular is probably GRAFPAC. These provide a library of subroutines which draw points, lines, circles, etc. on graphics displays. Programs using such packages, by their very nature, are generally not interactive, at least regarding the specification of their behavior. Changing their operation requires programming the appropriate subroutine calls, recompiling, and reloading. Furthermore, the host language (usually FORTRAN) is generally not very well suited for picture manipulation.

There are also interactive graphics systems designed for special purposes, such as curve-fitting. While these perform

(1) Catmull, Edwin. "A System for Computer Generated Movies," Proceedings of the Association for Computing Machinery, August 1972 Annual Conference, p. 422.

well in their area of intended use, they are not suited for development work.

One example of a picture-processing system is BUGSYS,(1) which permits fairly extensive examination and change of previously scanned pictures on a pixel-by-pixel basis. It is, however, difficult to make significant modifications to selected areas, especially since BUGSYS is not interactive. Moreover, there are no facilities for combining two or more pictures.

The XAP system,(2) developed at the University of Maryland, allows logical and arithmetic operations to be performed using several previously digitized pictures, but provides no mechanism for introducing or deleting features of a picture. It also is not interactive. Users must write FORTRAN programs which call the appropriate XAP routines. It is thus similar to a subroutine package.

The CIPG system is an interactive facility capable of operating on several scanned pictures simultaneously, combining them, scaling them, and performing many other common operations on them. New operations, however, must be defined by writing new commands; there is no facility for specifying a

---

(1) Ledley, R. S., et al. "BUGSYS: A Programming System for Picture Processing - Not for Debugging," Comm. ACM Vol. 9, p. 79 (1966)

(2) Hayes, Kenneth C., Jr. "XAP User's Manual," Technical Report #348, Computer Science Center, University of Maryland. (1975)

picture-processing operation and seeing its behavior immediately.

Commercial artists can manually erase offending portions of a picture, crop the picture, superimpose parts of other pictures on it, draw on it, and generally make drastic changes to it. The IPMS is intended to possess the same capabilities and to provide them in a more convenient form than the photographer's equipment.

Chapter II

IPMS Structure

Introduction

As discussed in the previous chapter, the picture processing routines of main interest at the time the system was begun included the following:

(1) Screening specified areas of a picture. This is actually a two-step operation: specifying the area to be screened, and converting the specification into an appropriate file of phototypesetter commands.

(2) Drop-Shadowing -- adding simulated shadows to objects. In particular, it was desired to generate drop-shadowed characters.

(3) Outlining -- essentially, detecting the edges in two-level black and white pictures. This included both generating tracings around the edges of the objects, and hollowing out the insides.

In order to develop the routines required, a framework was designed in which to carry on the necessary development quickly and easily. The framework had to provide the ability to run individual routines whose precise nature could not be foreseen when the framework was designed -- routines whose arguments might differ considerably, both from routine to routine and from call to call of the same routine. A "command processor" was called for; that is, a routine whose sole job was to accept user instructions describing routines to be executed, and execute them in the appropriate environment.

Another requirement of the system was that it facilitate the "orthogonal command set" philosophy. This is the philosophy of providing a small set of general primitives with as little overlapping functionality as possible, so that while a single command might not do very much, most tasks could be undertaken by combining them in various ways.

Furthermore, if the primitives are not sufficient to perform a given task, only those additional capabilities needed to complete the task must be programmed; there is no need to "re-invent the wheel" for each new application. This permits faster development of new processes by encouraging a "building block" approach to command development, an approach which has been very successful on UNIX.(1) In order to encourage it, however, the command processor should provide ways of "coupling" individual commands. It must be possible to execute a series of commands as though they were one command.

Each command may need to communicate information, such as picture coordinates, to other commands. Obviously, the user should not have to memorize such information and type it back in. Ideally, the user should not need to be aware of the

(1) Ritchie, D. M. and Thompson, K. "The UNIX Time-Sharing System". Comm. of the ACM Vol. 17, No. 7 (July 1974), pp. 365-375.

communication process at all, except when he needs to influence it.

Yet another requirement for the system was that it handle interfacing with the peculiarities of the "outside world" -- differing file formats, etc. Providing clean interfaces to "dirty" environments permits even the most casually written command to properly handle them. This requirement was especially important in view of the many different file formats that IPMS was expected to handle, and the peculiarities of its TV display.

Overview

IPMS consists of the components diagrammed in figure 1. Each of its parts will be discussed in more detail.

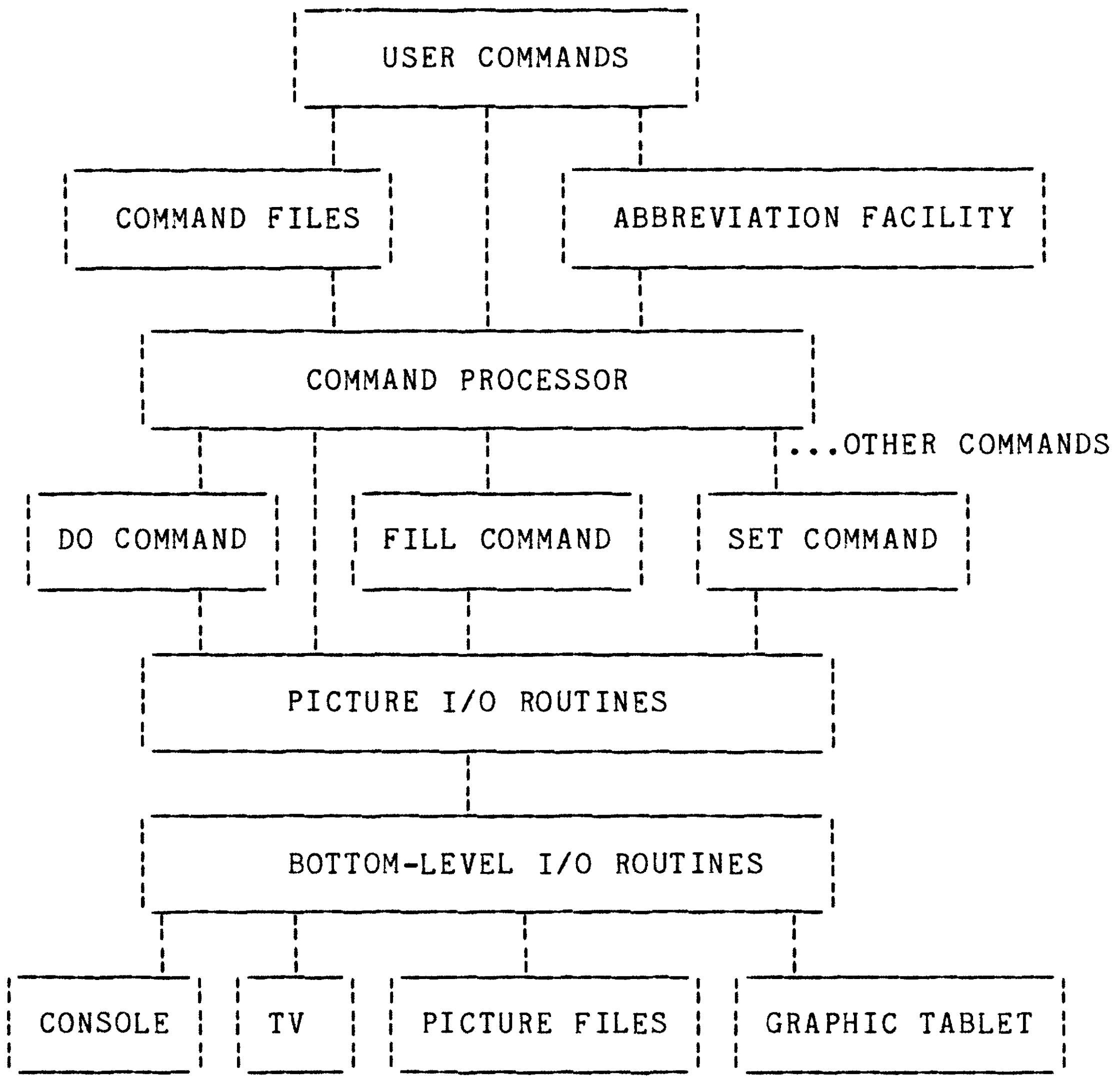

Figure 1. IPMS Block Diagram

The command processor implements a common, straightforward command syntax. Each command is specified by giving the name of the command, followed by its arguments. The name and arguments are separated by spaces.

Two facilities are provided by the commmand processor to combine elementary commands: command files and an abbreviation facility. Command files are just what the name implies: a file of commands to be executed. If IPMS does not recognize the name of a command, it checks to see if the user has a file by that name. If so, that file is used for further input, just as though the user had typed its contents. When the file is finished, IPMS returns to the console for further input.

The abbreviation facility is somewhat more complex. It permits the user to abbreviate any command, or part of a command, or series of commands, by specifying the desired abbreviation and the command sequence to the SET command. The abbreviation can then be used whenever desired by typing a period at the beginning of an argument. The rest of the argument will be taken as an abbreviation, and its value will be substituted in place of the argument. Because so many abbreviations are just commands, if the abbreviation is the name of the command to be executed, no period is necessary.

The abbreviation facility is also accessible to commands, which use it to save values of coordinates and other information intended to be invisible to most users. It is also used to keep certain information about pictures used during the session.

Much of this information -- e.g., abbreviations set up during a session -- should be saved for the next session with IPMS. This ability is provided by the SAVE command. It writes out all the information in the form of a series of SET commands which, when executed, set up the abbreviations, variables, etc.

to have the same value they did when the command was executed. Since it is just a file of commands, it is read in by the command-file facility just like any other command file. Additionally, it may be examined by the user and edited at will. Thus, as much or as little of the IPMS environment as desired may be restored. Note that by using the same facility -- which is nothing more than the ability to store strings by name and recall them at will -- for many similar purposes, IPMS is kept simple. This simplicity of design provides a high degree of functionality without requiring excessive memory.

The commands which manipulate picture files are also kept simple by simplifying the environment in which they do I/O. IPMS can deal with picture files having from 1-8 bits per pixel, up to 2048 pixels per line, and any number of lines. IPMS commands, however, deal only with pictures having 8 bits per pixel -- that is, one pixel per addressable unit of storage. They obtain their data by calling the picture I/O routines, which keep track of the actual size of pixels and pack or unpack them as required. The size of a pixel, the width of a line, and the number of lines must be fed to the picture I/O open routine. However, even this job is centralized; the command processor interprets a special argument syntax to indicate the name of a picture file to be opened, and looks up the format of the picture file in the "abbreviation" facility. If the format has not been specified, the command processor will interrogate the user for it. Once it has been obtained, however, it is retained until the end of the session, or longer if the user saves it and restores it later.

One other important job of the picture I/O routines is handling the TV. Adjacent scan lines in the TV are located half a frame apart. The amount of TV buffer memory needed to hold one frame varies depending on the size of a TV pixel, which can range from 1 to 4 bits. (Smaller pixel sizes permit more than one frame of data to be held in the TV; separate commands select the frame displayed.) Furthermore, if the TV is set to a pixel size of 3 bits, there aren't even an integral number of words per scan line! All these complications are handled by the picture I/O routines; a simple flag indicates whether the routines should address the raw file like a TV. Since the picture I/O routines do not actually "know" whether they are driving the TV or not, the user can prepare TV format files ahead of time and display them quickly by doing a direct copy.

The DO and FILL Commands

IPMS presently implements 12 commands "directly" -- that is, as FORTRAN subroutines. (Several more commands are actually abbreviations.) Two of these commands have already been discussed: SET and SAVE. Some other commands perform minor functions such as setting the pixel size or colors of the TV. These are not important to understanding IPMS. However, the DO and FILL commands are important enough to be described in some detail.

The first command is the "DO" command. It executes a specification of a local, position-invariant, picture operation over a specified portion of a picture. The result of the command is a new picture; the input picture is unmodified (under normal circumstances). The specification is in a language called PIPO (for Position Invariant Picture Operations). PIPO is described in detail in the next chapter; the following description is meant only as an overview.

PIPO provides a convenient, interactive way of specifying a large class of local, position-invariant operations on a picture, including arithmetic and logical operations among pixels, conditional operations, and table lookup.

The PIPO language was designed to meet the following objectives:

(1) It should permit easy, natural expression of picture operations;

(2) It should be compact, so that it will be easy for users to specify an operation at the keyboard;

(3) It should be simple to understand;

(4) It should be straightforward to translate a PIPO command into very efficient code.

The last objective was particularly important, in view of the tight timing requirements imposed by an interactive system. Although IPMS does not claim to be a real-time picture-processing system, typical PIPO commands should not take more than about 15 seconds to execute or the system is no longer truly interactive. But an operation on all the elements of a 512x508 TV picture must be very fast in order not to take a long time. Hence this requirement had a strong influence on the language.

A PIPO command is a specification of a series of operations to be performed in order to produce an output pixel. The command is iterated over a specified portion of the picture, producing one pixel of the new picture with each iteration.

For operands, a command may use local pixels, constants, or values in a separate array. Use of local pixels is done through reference to a matrix whose elements are numbered as follows:

```
0,0  0,1  0,2  ...
1,0  1,1  1,2  ...
2,0  2,1  2,2  ...
 .    .    .
 .    .    .
 .    .    .
```

This matrix is overlaid on the upper left hand corner of the picture, and shifted over between each iteration until it has covered the entire picture. The pixel at position (0,0) is the "current pixel." Its absolute coordinates in the input picture are equal to the coordinates of the output pixel that will result from the iteration. In other words, if the operation copies pixel (0,0) to the output on each iteration, the new picture will be identical to the old. The IPMS command to specify this operation is just

```
DO 0,0
```

The "DO" command translates its arguments into a PIPO program and executes it over the picture. The pictures to be used for input and output, and the portion of each picture (the subpicture) actually used, are specified separately.

If (1,1) had been specified instead, the output picture would have been shifted diagonally upward and to the left. Note that this notation does not permit specification of translation down or to the right. Note also that an operation which must refer to pixels to its left or above itself has to consider a pixel other than (0,0) to be the "current pixel." The operation will thus end up translating the picture to the left and/or up. These annoyances are not serious, because IPMS provides other commands for translating pictures in any direction.

PIPO commands can also combine several pixels. To produce an output picture summing four adjacent input pixels, one simply says

```
DO 0,0 + 0,1 + 1,0 + 1,1
```

(Spaces within a PIPO specification are ignored.) Each output pixel will be the sum of four input pixels. To divide the result by 4 for scaling purposes, another operator is added on the end:

```
DO 0,0 + 0,1 + 1,0 + 1,1 / #4
```

Note that constants are represented by preceeding them with an octothorpe ("#"). Note also that PIPO commands are performed from left to right, rather than according to algebraic precedence. All of the standard arithmetic and logical operations are implemented. As another example,

```
DO 1,0 - 0,0
```

implements a simple vertical edge detector. A more complicated command,

```
DO 0,0+0,1+0,2+1,0+1,2+2,0+2,1+2,2 / #8 - 1,1 * #-2 + 1,1
```

performs contrast enhancement, by first calculating the average value of the neighbors of (1,1), then subtracting (1,1) from it and doubling and negating this difference, and adding it to the original value. This is less than a completely natural expression of the algorithm because this version of PIPO lacks parentheses. They were not provided because the additional complexity would not have been sufficiently useful for the development of the artistic operations for which IPMS was intended.

The last kind of value a PIPO operation can refer to is an externally supplied array value. The value calculated so far is used as an index into the array, with a value of zero selecting the first element of the array. For example, to map each pixel of a picture into a new value, one simply says

```
DO 0,0 T
```

This simple operation causes the value of pixel (0,0) to be used as an index into the array. The value found in the array becomes the new current value.

An extension of this technique permits the taking of histograms by using the increment operator, I:

```
DO 0,0 I T
```

The current value is used as an index into the table, and the value found there is incremented.

To set and examine the table, the user employs the abbreviation TABLE. The value of this abbreviation is a string of numbers representing the values of successive elements of the array. For example, the commmand

```
SET TABLE 0 2 4 6 8 10
```

sets the first six elements of the table to the specified numbers. The DO command reads the value of TABLE when it begins, and updates it when it is done.

In order to implement some commercial artists' operations, another PIPO capability was required: the ability to execute sequences of operations conditionally, based on comparison of the last calculated value with other values.

Using the conditional feature of PIPO, one can compare the value generated so far with a constant or element of the matrix, performing some other operation conditionally as a result. The C operator is used to perform the comparison, and one of the IF operators is used to act on it. For example:

```
DO 0,0 C 0,1 IF< #0 ELSE #1
```

This command compares each pixel with its right-hand neighbor; for each pixel that was less than its neighbor, the output picture will contain a 0, otherwise it will contain a 1. Any number of commands may follow an IF or ELSE, and they may refer to the value last generated. All the commands following an IF are conditional upon the IF, until a succeeding ELSE or FI. FI, rarely needed, "closes" the IF statement so that later commands are performed regardless of the result of the IF. All the normal comparisons are provided; IF<, IF>, IF=, IF<=, IF>=, and IF!= (not equal).

With the operators discussed so far, it is possible to implement the following operations:

(1) Outlining. This operation consists of drawing lines around the edges of the picture. For a black-and-white picture, this is simply

```
DO 0,0¦0,1¦0,2¦1,0¦1,2¦2,0¦2,1¦2,2
                         C#0 IF> 1,1 C#0 IF= #1 ELSE #0 ELSE #0
```

This operation ORs together all the neighbors of (1,1). (The vertical bar, "¦", is used to indicate logical ORing.) If any of them were nonzero, the center pixel may be by an edge. If the center pixel is 0, then it is indeed next to an edge, so put out a white point (value = 1). Otherwise the center pixel must be in the middle of white space, so put out a black point (value = 0). Figure 1b contains an example of this operation.

(2) Edge generation. This operation produces a copy of the input picture with the insides of white areas "hollowed out." The technique is to simply check whether the current pixel is completely surrounded by white; if so, it is turned black.

```
DO 0,0&0,1&0,2&1,0&1,2&2,0&2,1&2,2 IF> #0 ELSE 1,1
```

The ampersand, "&", is used to indicate logical AND. The first part of the specification will have a value of 1 if and only if all the neighbors of (1,1) are 1 (white). The IF operator has no preceeding C operator, so the IF will behave as though a comparison had been made with zero.

(3) Shadowing. This consists of generating a shadow of the picture, omitting the picture itself. The picture and the shadow will later be combined, using two separate colors. The algorithm simply produces a white pixel whenever (1) the current pixel is black, and (2) a white pixel is within a fixed distance in a diagonal direction. The shadow lies along the axis of the diagonal.

```
DO 0,0 IF> #0 ELSE 1,1¦2,2¦3,3¦4,4¦5,5
```

This command puts out a black pixel whenever pixel (0,0) is white. Otherwise, it puts out a white pixel if any of the pixels on the diagonal within five pixels are white. As simple as this algorithm may appear, the shadows it produces are quite acceptable. See figures 1c and 1d.

All the DO commands given so far have assumed that one iteration is done for each and every pixel in the subpicture being used as input. Many picture operations require otherwise; sampling algorithms, for example, or ordered dither. To permit skipping pixels, the DO command looks up the values of the "abbreviations" DX and DY. Their values must be positive integers. DX is the horizontal stepsize; DY is the vertical stepsize. They are both normally one. The SET command may be used to change their values.

The usual use for DX and DY is in displaying digitized pictures on the TV. These pictures usually have a much higher resolution than the TV. Hence, setting DX and DY to a suitable value and executing the command

```
DO 0,0
```

will display a sampled picture on the TV.

Some operations required more flexibility than that provided by the DO command. In particular, in order to implement the screening operation, two new commands were needed:

(1) A command to interactively specify the area to be screened.

(2) A routine to convert the picture of the screened area into a series of phototypesetter commands.

The easiest way to specify an area to be screened was through a closed curve to be filled with screening of a specified shade. This was because the picture generally already contained curves around areas to be screened, and where it did not, the artist could draw curves to indicate the areas explicitly. At the console, specifying any point within the curve would indicate that the entire area should be filled. To specify the point, it was decided that the TV and graphic tablet would be the most convenient input device. Hence a command was written to display a cursor on the TV and read from the tablet. The coordinates obtained when the user depressed the stylus were stored as the values of abbreviations, where any other command could reference them.

A second command was written to fill an arbitrary closed curve with points of a given color. Given the user's specification of any point within the area to be filled, the color of the curve to be followed, and the color to be filled in, the command generated an output file containing the filler.

There were several requirements for the fill routine. It had to be reasonably fast, despite the fact that individual scan lines of the picture containing the curve would be on a slow random-access device. The routine also had to avoid requiring main memory proportional to the area to be filled, as it might be quite large, and the routine would eventually be implemented on a minicomputer.

Achieving these requirements took some work. The command had to keep track of the portion of the area already filled in. The simplest way to keep this information was to use the filler picture being built up. Refinements were necessary in order to be able to process this information at a reasonable rate. The complete algorithm is described in chapter 4.

Unfortunately, the fill command was not always successful when applied to digitized line drawings. The curve to be filled in often had small gaps. The fill command could "escape" through these gaps if they were not eliminated. To solve this problem, a command was added to draw lines at specified places on the picture. As with the fill command, the line command accepted coordinates which were obtained with the tablet-reading command. The user was asked to specify a pair of points, and a line was drawn between them. For convenience, the command combination accepted any number of pairs of points, until the user indicated that he was done by pressing a switch on the tablet.

A third command wrote out the picture in a form suitable for conversion to phototypesetter code, and another (non-interactive) command did the actual conversion. The conversion was not done immediately because it took much time and resulted in a larger set of data than the picture itself. These commands are also discussed in chapter 4.

IPMS also provided a command-level syntax for combining pictures. In its simplest form, it permitted a user to replace any rectangular section of a picture with a rectangular portion of another (or the same) picture, simply by setting the subpicture windows appropriately. In addition to replacement of the subpicture, the user could also OR and XOR the two subpictures together. This ability turned out to be only marginally useful in the development of the picture-processing operations described.

Chapter III

The DO Command

Introduction

The SEL-86 is a 32-bit single-address machine, with 8 32-bit general registers. Its instructions generally come in two forms: immediate, in which the operand is a constant contained in the instruction, and memory reference. A memory reference instruction can specify an index register and/or an indirect address (pointer) chain to be used in address calculation. It also specifies the size of the operand: byte, halfword, fullword, or doubleword. Thus, there is add immediate, add byte, add halfword, etc.

A memory cycle on SEL-86 is 600 nsec. Each simple immediate instruction takes one cycle; each memory reference instruction takes one cycle longer than its immediate counterpart, or longer still if indirect addressing is specified.

As discussed earlier on page 9, interactive picture operations of the sort envisioned for IPMS set severe constraints on execution efficiency and user convenience, rendering both assembly language and FORTRAN unacceptable. What was needed was a compile-and-go translator providing the efficiency of assembly language and the convenience of FORTRAN (or better). Such a translator is provided by the IPMS DO command.

Implementation of the DO Command

The DO command translates its arguments into SEL machine code and executes the result on the picture. Both the translation and the resulting code are quite fast; indeed, the code produced is nearly as efficient as hand-coded assembler would be.

The translator achieves this efficiency through a combination of careful language design and clever implementation. The PIPO language is actually close to the machine language of the SEL in many respects; in fact, there is very nearly a one-to-one correspondence between PIPO operators and the resulting machine instructions. The translation process consists of matching the first four characters of the string (from which all blanks have been removed) against the list of operators. Some operators are shorter than others, in which case less than four characters of the input string are actually compared against. In any event, further action is dependent on the table entries associated with that operator. These entries are (1) the machine instruction to be used, (2) a value indicating the type of operand, and (3) an optional set of flags for the operator.

For example, for the "+" operator, the instruction obtained would be "add_byte"(1) and its type would be "memref",

(1) This is not the actual name of the instruction in SEL assembler; throughout this discussion, long mnemonic names will be used rather than the SEL names.

meaning that a matrix reference was expected to follow in the input string. The translator would expect a coordinate specification of the form "m" or "m,n" which it would translate into an appropriate offset, indexed by a register which will later hold the address of the "current pixel."

If a constant were specified as the operand to the "+" operator, the table matcher would find a match for "+#", which would have the instruction "add_immediate" and type "immediate." The translator would expect a number, which it would insert into the instruction. Note that the "#" is effectively part of the opcode rather than the operand, a difference which is invisible to the user.

One operator which requires special assistance is the "/" operator, which divides the current value by its operand. The SEL requires that the dividend be 64 bits long, spread over two registers. Hence this operator is flagged so that an "extend_sign" instruction is put out first, extending the current value into the adjacent register.

Another group of operators requiring special handling are the ones that implement conditionals -- IF, ELSE, and FI. The IF operator actually consists of the "IF" and its one or two character condition name. The operator is translated into the appropriate conditional branch. However, since this is a branch forward, the translator does not yet know where it is branching to. So it stacks the location of the IF. When it finds an ELSE or FI, it unstacks the location and fills it in with the address of the next instruction. ELSE is, of course, an unconditional branch; its (earlier) matching IF is made to point to the location just after the unconditional branch, and the location of the unconditional branch is stacked in place of that of the IF.

When the end of the specification is reached, the stack is checked. If it is not empty, the specified locations are filled in with the address of the instruction after the last operator. This is the behavior that would have occurred anyway if the specification had ended with the appropriate number of FIs. This default action was chosen to minimize the number of FIs needed in most operations. In practice, it is rare to need any FIs at all.

The "T" operand also requires special treatment. This operand refers, not to a constant or a pixel value, but to the value of an element in a separately supplied array. The element is chosen on the basis of the current value. This operand is used for table lookup. It can be used with any "memref" operator. It assembles into the same general form as the matrix specification -- that is, an address and an index. But the address is the address of the base of the array -- that is, element 0. The index register is the register holding the current value. Thus, the operator uses the appropriate value from the array.

If the translator does not recognize the first few characters, it prefixes an "L" to them and tries again. If the first few characters specify a coordinate, such as "0,0", this action will generate a match to the "L" operator in the table, which generates a load instruction. This is how the first value specified is loaded for use. If the first few characters were a constant, "#34", for example, the "L#" operator would be found. It generates a "load_immediate" instruction, just like the other immediate instructions.

Before beginning translation, and after it is over, fixed sequences are put out. The prefix sequence starts loop execution; the suffix sequence stores the value left in the register all the operators use, and branches back to the beginning if the loop has not yet been executed the specified number of times.

Here is an example. The IPMS command

```
DO 0,0+#1
```

is translated into the loop

```
          copy_reg  r0,r5
          load_im   r2,-<number_of_iterations>

loop      load      r1,<addr1>,r3
          add_im    r1,1
          store     r1,<addr2>,r2
          add_im    r3,<stepsize>
          inc_br    r2,loop
          return
```

The two instructions preceeding the label "loop" set up the loop for execution. Strictly speaking, the copy is unnecessary; it is provided as a matter of convenience for the calling routine. The load immediate initializes register 2 with the negative (two's complement) of the number of iterations, which is inserted in the instruction before execution. The loop is then executed. Register 1 holds the current value. The store instruction outputs the current value into the appropriate output pixel. Then the index register for the input array, register 3, is bumped by the "stepsize" of the loop. This is the size of the "step" the virtual matrix takes between iterations -- that is, the value of DX (see chapter 2). Finally, register 2 is incremented, and if it is not yet zero, control is returned to the location labeled "loop."

This sequence takes 7 cycles per iteration, for a total time of 1.1 seconds when run on a 512x508 picture. The entire command actually takes over twice as long, because of time spent reading and writing the TV buffer. If the command were instead run on a disk file, it would take even longer.

Table 1 summarizes the operators in the PIPO language and their possible operands.

Reducing I/O Overhead

The description above has oversimplified the translation task somewhat. There is no limit to the size of the matrix coordinates. A DO command like "DO 0,0+1,0" requires only one input scan line, while "DO 0,0+1,1+2,2+3,3+4,4" requires five. All those scan lines are needed in order to produce one line of output. Hence the PIPO translator determines how many input lines are needed so that the DO command can allocate an appropriately-sized buffer and read them in before executing the PIPO loop.

Note that each use of the loop only produces one output line. It would appear that if a PIPO program requires five scan lines for each iteration, a fresh set of five lines must be read in each time the loop is executed.

This is inefficient, particularly since I/O overhead is such a large portion of total execution time. Note that successive loop executions generally require overlapping lines to be read in. In fact, once an initial set of lines has been read in, each further use of the loop generally requires one new scan line (if DY is one), and discards one old line. Therefore, it ought to be possible to simply read in the new line over the old one, and inform the PIPO program of the new mapping between y coordinates and memory addresses.

To provide this ability, the PIPO translator outputs an "unfinished" program -- one in which the mapping between y coordinates of matrix elements and memory addresses is left unspecified. A separate routine then fills in these addresses.

It is possible to relink the program with new addresses after each execution of the loop, but it is normally well worth the additional space to simply link N copies of the program, one for each cyclic permutation of the N scan lines in memory. The appropriate copy is then selected each time a new line is read in. The result is an N-fold decrease in execution time.

Extensions to the Language

Although PIPO as it is now implemented is easily capable of handling problems such as the operations desired by commercial artists, many picture-processing algorithms require considerable distortion before they can be implemented using PIPO. For example, consider a smoothing algorithm which scales the values of a group of pixels. It would be convenient if one could say

```
DO Ø,Ø + (1,Ø * #2) + (2,Ø * #4) + (3,Ø * #2) + 4,Ø
```

Since PIPO does not handle parentheses, it is necessary to say instead

```
DO 1,Ø * #2 S 1,Ø  2,Ø * #4 S 2,Ø  3,Ø * #2 + 1,Ø + 2,Ø + 4,Ø
```

Here, the store operator ("S") is used to store the products back into the input array elements. They are then added back in. This operation is slow (because of its use of memory to hold partial results) and clumsy. Furthermore, it does not generalize to operations requiring more than one input scan line; the optimization described above leaves around the old, scaled values.

Adding parentheses to the language would not be hard, and is clearly desirable. For efficiency, it ought to use whatever registers are available to hold partial results, storing into temporary memory locations only when absolutely necessary.

Another problem with PIPO as implemented is that it can generate only one output pixel per iteration. Some operations could be expressed in a more efficient format if they could write out many pixels in one iteration. For example, the shadow generating algorithm could work by scanning for a diagonal edge; that is, a configuration in which a black point is diagonally adjacent to a white one. At such an edge, the algorithm could write out a diagonal line of shadow pixels.(1)

Another extension which would be useful for picture processing would be the ability to calculate with floating-point pixels. Although this extension would not be difficult, it would not be useful for the artistic operations for which IPMS was intended, and so was not done at this time.

Additional flexibility could be provided if the user were given the ability to "return" out of an iteration. This would enable one to do pattern searching reasonably fast. Of course, there would need to be some way of communicating the coordinates of the pattern.

It might also seem useful to allow the user to call his own special-purpose routines. This extension, however, removes one of PIPO's chief benefits: its interactiveness. The user would have to go through the same tedious process of assembling, linking and loading that the DO command was intended to avoid. It would seem more worthwhile, therefore,

(1) This shadowing algorithm is due to Don Davis of PRIME Incorporated.

to increase the power of the language itself so that such a mechanism is unnecessary.

THE PIPO OPERATORS

NAME	OPERAND TYPES[1]	FUNCTION
!	memory	logical OR
&	memory	logical AND
^	memory	exclusive OR
+	memory, constant	add
-	memory, constant	subtract
*	memory, constant	multiply
/	memory, constant	divide
C	memory, constant	compare for IF [2]
IF	none	test on condition: =, <, >, <=, >=, !=, O, NO[3]
ELSE	none	alternate branch for IF
FI	none	terminates IF-ELSE clause
I	memory	increments the location
S	memory	stores into named location
RL, RR	# bits	rotates left or right
SL, SR	# bits	shifts left or right
SLA, SRA	# bits	like SL and SR, but signed
L	memory, constant	loads the value
LN	memory	loads negated value

Notes:

[1] "memory" is either a matrix point specified as X or X,Y for X,Y nonnegative integers (if Y not given, 0 assumed); or the T operand. "constant" is of the form #<integer>, where -32768 <= integer <= 32767. "# bits" is a nonnegative integer ranging up to 31 inclusive.

[2] If there is no C operator immediately before an IF, comparison with 0 is assumed.

[3] Calculations are carried out in 32 bits. Overflow can be tested for with the IFO (if overflow) and IFNO (if no overflow) operators.

Chapter IV

Screening and the FILL Command

Introduction

Screening is a process for simulating gray scale using reproduction methods normally incapable of it. Gray areas are represented by arrays of small dots. The size of the dots determines the shade of gray. This technique can be used to represent halftone photographs, which contain continuously changing gray levels, or just to add patches of gray of various shades to black and white pictures.

Presently, these patches of gray are added by hand; a craftsman carefully cuts transparent windows in overlays to match the areas to be screened. These overlays are then combined with sheets of screening and photographed. The result is combined with the original picture.

Sophisticated typesetters are readily capable of producing both the black-and-white and screened portions of the ad; all that is needed is the necessary software. Hence one of the tasks of the system under development was to automate as much of the procedure as possible.

The procedure can be divided into two parts: specification of the area to be screened, and generation of a file of phototypesetter commands for producing the complete ad.

As discussed in chapter 2, the artist indicates the area to be screened by drawing curves on the picture surrounding the areas to be filled. Then, at the console, he indicates one point anywhere in each area to be filled. The FILL command then fills in the area contained in the curve. The filled-in area is displayed on the TV. Filled-in areas can be changed in shade or erased entirely. Once the artist is satisfied, he writes out the picture. A non-interactive task will later convert the picture into phototypesetter commands.

IPMS was used to develop and check out the filling algorithm. Its FILL command was then rewritten for the NOVA target machine.

The FILL command regards all closed curves as convex curves with inner projections -- "stalactites" and "stalagmites". To see how it works, consider the following algorithm:

```
GIVEN:
   a white picture containing a closed curve in black
   a point (xØ,yØ) within the area enclosed by the curve

DO:
   xl := xØ
   yl := yØ
   repeat
       scan left along the line y=yl (current line)
                   until left edge (black point)
       scan right along the line
                   until right edge (black point), doing:
           find the leftmost white point on the line above
                push its coordinates
           find the leftmost white point on the line below
                push its coordinates
           fill in the current line
       pop the last point pushed; assign to xl,yl
   until there are no more points to pop
```

Here is an example of the operation of this algorithm. Consider the following picture, in which an at-sign ("@") is used to represent a black point and O, P, Q, and S represent white points:

```
O O O O @ @ @ O
O @ @ @ Q O @ O
@ O O S O O @ O
@ P O O O O @ O
O @ @ @ @ @ @ O
O O O O O O O O
```

The algorithm is started at point "S". It scans to the left on the current line until it reaches the edge. Then it scans right. It immediately finds point P, and pushes its coordinates. Then it finds point Q, at which time it pushes _its_ coordinates. Finally, it reaches the end of the current line. It fills it in. The picture now looks like:

```
O O O O @ @ @ O
O @ @ @ Q O @ O
@ @ @ @ @ @ @ O
@ P O O O O @ O
O @ @ @ @ @ @ O
O O O O O O O O
```

The coordinates of Q are popped off the stack and the process is repeated. This time there are no points pushed, as neither the line above nor the line below is white for any part of its length. Finally, the coordinates of P are popped off, and its line is filled in.

This algorithm will work on any "horizontally-convex" curve -- that is, any curve satisfying the property that for any two points on the same horizontal line, a line drawn between them is contained entirely within the curve. This can be seen from the fact that in order for all the points within the curve at a given y coordinate to be filled, it is only necessary to push one of them on the stack at some time. That every horizontal line will have at least one of its points pushed follows from the fact that two adjacent lines each have at least one point with the same x coordinate. Every horizontal line can eventually be reached through adjacency.

The algorithm fails, however, on closed curves not satisfying the above property. For such curves, it is not true that all the points on a horizontal line within the curve are "reachable" from one pushed point. Consider the curve

```
@ @ @ @ @ @ @
@ P O @ Q O @
@ O O S O O @
@ @ @ @ @ @ @
```

As the line containing point S (the starting point) is scanned, point P will be pushed; but point Q will not.

Clearly what is needed is to identify the beginning of these additional horizontal lines and push the coordinates of them as well. The beginning of such a line is indicated by the presence of a black point followed by a white one. The coordinates of the white point should be pushed. The modified algorithm is as follows:

```
GIVEN:
   a white picture containing a closed curve in black
   a point (xØ,yØ) within the area enclosed by the curve

DO:
   xl := xØ
   yl := yØ
   repeat
       scan left along the line y=yl (current line)
                    until left edge (black point)
       scan right along the line
                    until right edge (black point), doing:
           find the leftmost white point on the line above
                push its coordinates
           find the leftmost white point on the line below
                push its coordinates
           if the point above is white
                and if the point to its left is black
                push its coordinates
           if the point below is white
                and if the point to its left is black
                push its coordinates
           fill in the current line
       pop the last point pushed; assign to xl,yl
   until there are no more points to pop
```

The inner loop can be streamlined somewhat by combining the search for the first white point on the line above or below with the search for points after stalactites or stalagmites. Demonstrating this optimization requires somewhat more detail:

```
xl := xØ
yl := yØ
repeat
   loop for x := xl step -1 until point(x,yl) = black
   x_left_edge := x
   prev_point_above := black
   prev_point_below := black
   loop for x := x_left_edge+1 step 1 until point(x,yl)=black
      if prev_point_above = black and point(x, yl-1) = white
         push(x, yl - 1)
      if prev_point_below = black and point(x, yl+1) = white
         push(x, yl + 1)
      point(x, yl) := black
      prev_point_above := point(x, yl - 1)
      prev_point_below := point(x, yl + 1)
until pop(xl, yl) returns error
```

One further complication ensued. It was required that the command take as input the picture containing the curve, and produce as output a separate picture containing the filled areas. These areas would later be translated into screened regions, while the original would be translated into black and white.

The algorithm above modifies the picture as it goes. In fixing this problem, it must be remembered that the filled in areas generated in the course of the algorithm are needed to guide it, so that it does not attempt to fill in the same area more than once. (This is not merely an efficiency consideration; without the filled-in areas, the algorithm will, for all but the simplest cases, loop forever.)

Hence the fill routine was modified slightly: it produced an output picture separate from the given input, and read from both the input and output pictures.

Producing Screened Pictures on the Phototypesetter

Once the artist is satisfied with a screened ad, he writes it out. The picture is then converted to phototypesetter commands.

The actual commands to perform these operations are different on the IPMS and the target machine. Hence the following description applied only on IPMS.

After screening, the complete picture consists of several picture files. This includes the original picture and one picture for each shade of gray desired. Another command writes out these pictures in compressed format.

Given the compressed files, a non-interactive task converts them into a series of phototypesetter commands that generate the picture.

Producing screened pictures requires merely producing a series of dots across the appropriate regions. To make the lines less apparent, these points are arrayed along diagonal lines at angles of 45 and 135 degrees to the side of the page. The following formula yields the horizontal center-to-center distance between rules:

$$S = L / (D * \mathrm{sqrt}(2))$$

where D is the screening density in (diagonal) lines per inch, L is the resolution of the typesetter (basic units per inch), and S is the center-to-center spacing in the typesetter's basic units. For clarity, the rest of this discussion will assume a (not atypical) resolution of 720 units per inch: that is, a basic unit of one tenth of a point.

For the Yellow Pages, a screening density of about 90 lines to the inch was chosen as the nearest convenient density to the standard 85 lines/inch being employed. This density yields a spacing of 12 tenth points.

Given a center-to-center spacing of 12 tenth points, it can be seen that a rule 12 wide and 6 high is the largest that can be accomodated without overlapping. For non-overlapping rules, the gray level can be calculated from:

$$L = W * H / (S * S / 2)$$

where W and H are the width and height of the rule, S is the spacing found above, and L is the gray level (0 = white, 1 = black). Note that this formula assumes that rules are perfectly black. Although this is not strictly true, the discrepancy can be ignored for now.

Adjacent rules (that is, rules on adjacent lines) will overlap if both the width and height of the rule are more than 6 tenth points. In this case, the gray level is

$$L = (W * H - (W - 6) * (H - 6)) / (S * S / 2)$$

The second term is the amount of overlapping area.

How many distinct gray levels can be obtained? Assume that the phototypesetter is capable of putting out rules as small as 1 by 1. However, the Yellow Pages can reliably reproduce dots no smaller than .5 mm in diameter, or approximately 3 by 3. A selection of rule sizes, and the gray levels they produce, is given in table 2. For Yellow Pages purposes, only three distinct gray levels are actually required, so more than enough are available to choose from.

There is one minor obstacle to this process. Producing one dot requires two phototypesetter commands; one to move the beam, and one to draw the rule. For typical screened areas, the number of dots drawn can easily reach twenty thousand or more. Such a large number of commands would cause screened pictures to require an unacceptable amount of room -- both on the tape mounted on the phototypesetter, and stored on disk in the process of production. Hence the sophisticated instruction set of the particular phototypesetter employed -- an APS-4 -- was used to set up a loop in one of its buffers which could draw one line of dots. Then only three commands per line were required: one to step vertically, one to set the length of the rule, and one to execute the buffer. This procedure reduced the number of commands needed to an acceptable level.

Recall that a run-length encoded picture file is a sequence of value-length pairs, with each pair representing "length" consecutive pixels of value "value". A small amount of protection is provided against error by encoding each scan line individually, ending the sequence of pairs for each scan line with a special sequence. For the phototypesetter translator program, the pixel values will all be either 0 or 1. 1 can represent either a black pixel or a gray pixel.(1) An area of gray pixels is translated into a screened area.

The algorithm used to generate APS command files attempts to put out as few commands as possible. It uses a virtual cursor directed by the run-length encoded picture file, and a

(1) This awkward format, which requires a separate picture file for each gray level of the picture, was necessitated by the constraints on the target system.

"true" cursor which simulates the beam on the phototypesetter. For a value of zero (white), the virtual cursor is moved horizontally by the length specified. When the end-of-line character sequence is detected, the virtual cursor is moved down to the beginning of the line. The true cursor is not moved. If the value is one (black), then it is necessary to generate rules. First, the commands needed to move the phototypesetter beam, represented by the true cursor, are produced, and the true cursor is superimposed on the virtual cursor. Then the commands to produce a rule (or sequence of rules, if this is a screening file) are generated, and the true cursor is moved to the position it will occupy after those commands are executed. Since space can be saved if the rule being produced is the same size as the last rule, the program also keeps track of rule sizes, reusing them when possible. In this way the size of the resulting data is kept down.

The program attempts to produce phototypesetter pictures on a scale of one pixel per tenth point. For ordinary (black and white) picture files, this presents no problems. For screening files, since dots are considerably larger than one tenth point, some finessing is required. Horizontally, partial rules are put out at each end of a line when needed. Vertically, the program simply samples every sixth line. These compromises produce acceptable pictures. For testing purposes, however, it is convenient to be able to convert a picture file to a screened picture such that each scan line produces one line of dots. (Otherwise, it would have been necessary to pad each line with five lines of zeroes.) For this reason, scaling facilities were provided in the conversion program. A scale of six to one causes a picture file to be represented precisely.

SCREENING DENSITY AS A FUNCTION OF RULE DIMENSIONS
(9Ø LINES/INCH)

WIDTH	HEIGHT	GRAY LEVEL
3	3	Ø.125
3	4	Ø.167
3	5	Ø.2Ø8
3	6	Ø.25Ø
3	7	Ø.292
4	4	Ø.222
4	5	Ø.278
4	6	Ø.333
4	7	Ø.389
5	5	Ø.347
5	6	Ø.417
5	7	Ø.486
6	6	Ø.5ØØ
6	7	Ø.583

Table 2

Chapter V

Conclusions

The IPMS provided a convenient framework on which to design and test picture processing algorithms for the Yellow Pages project. The language PIPO provided a very convenient, interactive way of specifying local, position-invariant picture operations. Although simple in both design and implementation, PIPO proved to be both powerful and efficient. Using it, drop-shadowing, outlining, and haloing were easily designed and tested. It is expected that future operations of the same general type will be just as straightforward to accomplish.

For serious picture processing, however, PIPO is not nearly powerful enough. Some extensions to the language which would make it more expressive for operations such as contrast enhancement and averaging were discussed in chapter 3.

When it was necessary to design an algorithm for an operation -- filling -- which was not position-invariant and hence fell outside the range of PIPO's capabilities, the modular design of IPMS enabled concentrating on the algorithm while spending very little time worrying about interfacing with the system or the user.

By splitting a given operation into its component parts -- for example, separating the filling operation into (1) obtaining user input, (2) filling in the area, and (3) producing phototypesetter output, it was possible to use the same individual routines for other purposes. For example, the routine to obtain user input from the tablet was used by the line-drawing routine. Also, the fill command could be used in the composition of full-color pictures on the TV, even though it was actually used only for black-and-white phototypesetter output.

The general-purpose abbreviation facility also proved extremely useful. It enabled single commands, which were abbreviations, to be at once powerful, easy to use, and easy to change. Other users, and other tasks for IPMS, might require totally different sets of abbreviations; the system permits any degree of abbreviating without penalizing the user in any way for abbreviations he does not need.

The abbreviation facility was especially helpful with the DO command. Once a PIPO specification was worked out, a user would have no desire to retype it every time that operation was desired. Abbreviating the DO command with that specification permitted building up a library of picture operations.

As described in chapters two through four, the abbreviation facility also provided a convenient way for commands to communicate with other commands, as well as system routines. It also provided a place for storage of data needed by a single command from one of its uses to the next, minimizing the need for retyping. All in all, it proved to be a very clean, accessible facility.

Providing permanent storage of abbreviations in textual form, rather than some internal binary format, also had many advantages. The user could, without being at the SEL-86 console, examine the abbreviations developed during the last session, generate new abbreviations, or delete obsolete ones, all with the system text editor. Also, because the form in which the abbreviations were stored was a file of IPMS commands, other parameters which were not abbreviation values, such as TV characteristics, could be set. As a bonus, a complete demonstration sequence could be prepared in advance.

The system was not without its problems. The elementary picture I/O routines lacked any notion of "subpictures". These were implemented at a much higher level (see the description of the DO command). This higher level provided two windows, one applied on input and one on output. As long as the input picture and the output picture did not change, the effect was identical to that which would have been obtained by associating the subpicture specification with the individual pictures. However, any time several different pictures were operated upon, or an output picture fed back as input, the windows proved remarkably clumsy.

Although the original specification of the I/O routines included subpicture operations, for the sake of simplicity they were not implemented. It was felt that providing them at the lowest picture-manipulation level made operations not requiring them needlessly complicated and inefficient.

This was a mistake. First, nearly all practical picture operations require subpictures; even though this was only a development system, such operations were annoyingly frequent. Second, the simplicity obtained by leaving out the capability was an illusion; the needed complexity was merely pushed up to a higher, far more visible level. Third, the loss of efficiency would have been insignificant compared to the losses incurred in doing the actual bottom-level I/O. Therefore, one enhancement that ought to be made to the system is a subpicture capability at the lowest level (that is, in the picture I/O routines.)

Another decision made early in the development cycle was to avoid a special picture format so as to be able to work with almost any scanned picture that might be obtained. The system considered a picture file as merely a stream of pixels, with no formatting information whatsoever. This view also enabled convenient examination of pictures of unknown format; changing the system's notion of the pixel length, picture width, or picture height only required the use of one or two commands.

Decoupling the picture formatting information from the picture itself was probably also a mistake. Again, by keeping the picture file format simple, genuinely necessary complexity was pushed up to a far more visible level. While the user only had to specify the picture format once, he did need to be aware of the fact that the format had to be saved from session to session. Furthermore, it would be difficult to extend the format description to include other parameters. It would have been better to convert the picture into a special internal format when the user first specified its properties. This would have simplified the storing of a great deal of per-picture information, including the dimensions of a subpicture and the colors which should be used to display each pixel value on the TV. Furthermore, picture files would not have to be in the scanned, bit-map picture format which IPMS commands expected; they could, for example, be in some compressed format which the bottom-level routines could automatically convert.

Even without these extensions, IPMS proved to be a convenient system on which to develop algorithms for operations on scanned pictures.

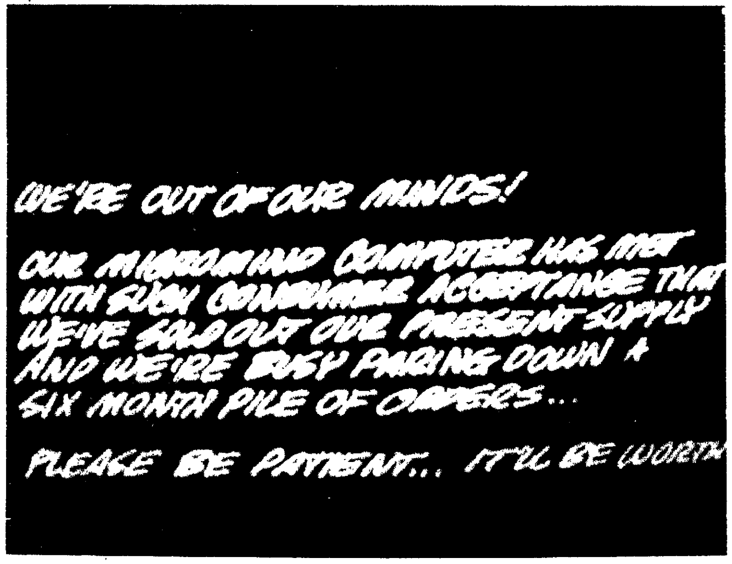
WE'RE OUT OF OUR MINDS!
OUR MICROMIND COMPUTER HAS MET
WITH SUCH CONSUMER ACCEPTANCE THAT
WE'VE SOLD OUT OUR PRESENT SUPPLY
AND WE'RE BUSY PARING DOWN A
SIX MONTH PILE OF ORDERS...
PLEASE BE PATIENT... IT'LL BE WORTH

Fig. 1A -- Original

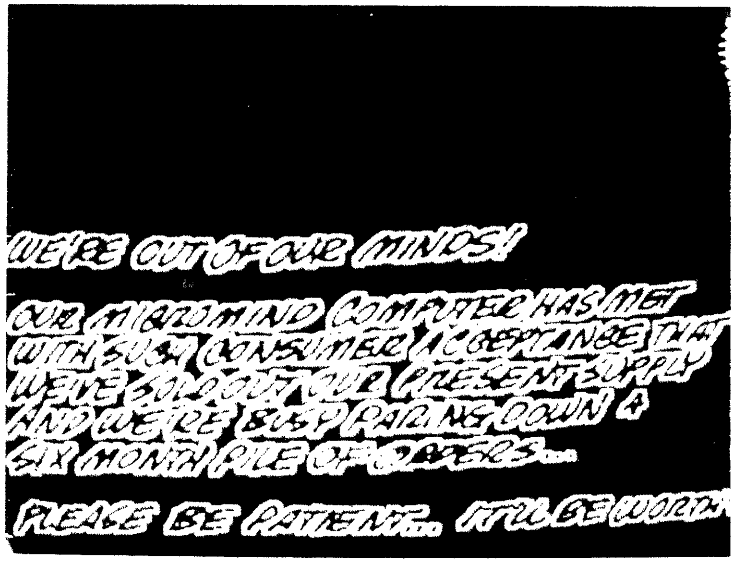
WE'RE OUT OF OUR MINDS!
OUR MICROMIND COMPUTER HAS MET
WITH SUCH CONSUMER ACCEPTANCE THAT
WE'VE SOLD OUT OUR PRESENT SUPPLY
AND WE'RE BUSY PARING DOWN A
SIX MONTH PILE OF ORDERS...
PLEASE BE PATIENT... IT'LL BE WORTH

Fig. 1B -- Outline of 1A

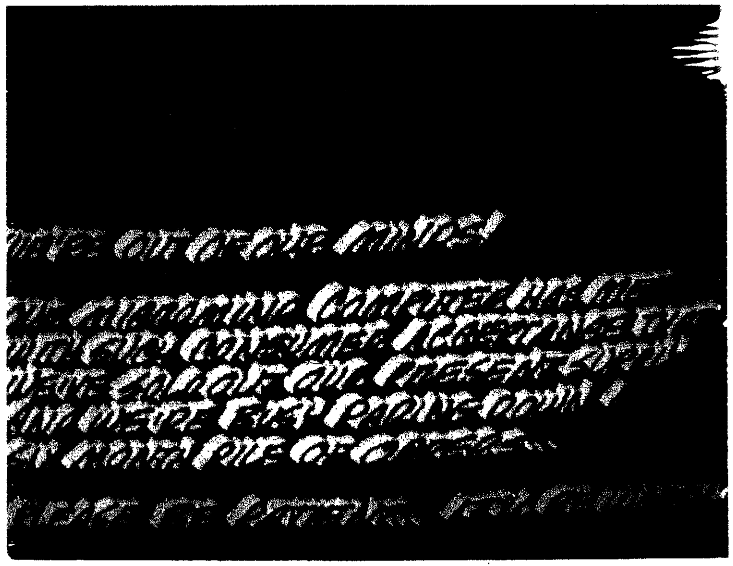

Fig. 1C -- Shadow of 1A

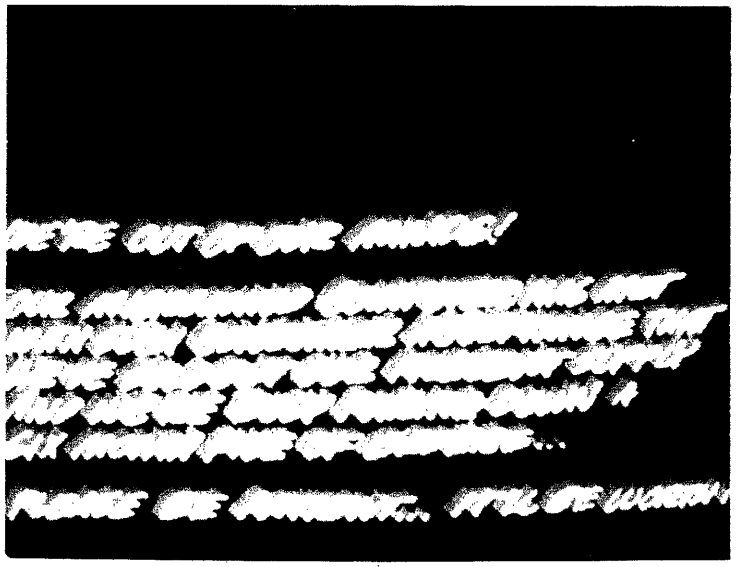

Fig. 1D -- 1A Overlaid with 1C