POPEYE: A Gray-Level Vision System for Robotics Applications

Rafael Bracho
John F. Schlag
Arthur C. Sanderson

CMU-RI-TR-83-6

Department of Electrical Engineering and
The Robotics Institute
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

4 May 1983

This research was partially supported by the National Science Foundation under Research Grant ECS-7923893 and the Robotics Institute, Carnegie-Mellon University.

Table of Contents

List of Figures

Abstract

A gray-level image processing system has been constructed to provide capability for inspection, object orientation, object classification, and interactive control tasks in an inexpensive, stand-alone system with moderate processing speed. The POPEYE system offers a range of functions including algorithms for preprocessing, feature extraction, image modeling, focusing, automatic pan, tilt, and zoom, interactive communication with other devices, and convenient user interaction. The host processor is a Motorola 68000 processor with Multibus communication between principal modules, an image data bus for acquisition and storage and a pipeline bus for image preprocessing and programmable transform operations. The software structure provides hierarchical control over multiple i/o devices, file management of system storage, an image management package and a vector package. Performance of the system is evaluated using convolution filters, adaptive modeling, histogram modification, and connectivity analysis. Cellular logic operations, piecewise gradient segmentation, automatic focusing, and adaptive spatial filtering examples are described in detail. The system is being applied to a number of practical industrial applications.

1. Introduction

Image processing and computer vision systems offer tremendous potential in the development of integrated systems which sense and adapt to external events. Visual feedback permits such robotic systems to evaluate, plan and execute courses of action based on sensory perceptions. In practice, such capabilities allow a robotic system to inspect and evaluate work in progress, to acquire and orient objects under visual control, and to plan manipulation or navigation in complex environments.[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11] and [12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]

The application of computer-based vision systems and their integration into complex systems has been limited by a number of factors inherent in current systems:

- SPEED. Most implementations require inspection speeds of about 1-10 seconds for manufacturing tasks and less than 1 second for robot control tasks.

- FUNCTION. While existing systems do recognition of gross silhouette shape in binary systems or image transformation and preprocessing in gray-level systems, no commercial systems do general forms of gray-level object recognition or inspection.

- FLEXIBILITY. The nature of industrial inspection tasks varies widely and systems must be inherently adaptable to many different tasks in order to be cost-effective.

- ROBUSTNESS. The system should offer robust performance under changing lighting or other environmental conditions. Binary vision systems are particularly sensitive to such factors.

- USER INTERACTION. The system should provide user interactive modes of operation to be useful as both an experimental tool for the development of applications as well as an on-line monitor of inspection results.

- SYSTEM INTERACTION. Integration of a vision system into a more complex environment depends strongly on the ability to interface and communicate. The lack of effective communications links in many current systems impairs the speed and flexibility of resulting integrated systems.

- COST. The cost of both development and production-line systems affects the feasibility of adoption. Current vision systems are major investments as components in a robotic system and have discouraged many prototype industrial applications.

The development of gray-level vision system algorithms, hardware, and software is still a difficult research task.[24, 25, 26, 27, 28, 29] Algorithms for such scene interpretation and object identification exist only for highly structured environments and have most often been developed on large, general-purpose computing machines. Imaging data is inherently complex due to the ambiguity which occurs between an observed

two-dimensional image and a given three-dimensional scene[30, 31, 32, 33]. The observed image depends not only on the geometry of the scene but also on light source geometry, surface orientation, surface reflectivity, and spectral distribution. Practical experiments on object description from imaging data require two or three cameras and significant assumptions about the scene characteristics.[30, 33, 10, 11, 12, 14] At CMU we have designed and constructed a gray-level processing system which will serve as an experimental tool in the development of algorithms, modular hardware elements, and interactive software. The principal goals of the system are to provide inexpensive gray-level capability for inspection, object orientation, object classification, and interactive control tasks in a stand-alone system with moderate processing speed. Inherent in these goals were decisions not to build special purpose hardware for the basic system structure, but to build functional hardware units utilizing commercially available components wherever possible. The software structure should provide for a complete range of system functions including digitization, frame storage, preprocessing, feature extraction, segmentation, image modeling, classification, automatic focus, pan, tilt and zoom, display, storage, communications with other automated devices and convenient user interaction. In addition, the software structure should be largely independent of particular modular hardware components so that hardware enhancements may be added without major restructuring of the software.

The general characteristics of the resulting system are described in this paper. The system currently is in routine use for algorithm development with particular attention to model-based approaches to object orientation and classification. The system communicates with the Flexible Assembly Station[34], an experimental system for investigating research issues in sensor-based assembly, and is used for interactive control of robots as well as on-line inspection of assembly components. The gray-level vision system has been applied to a number of specific industrial problems under funding from industrial sponsors and affiliates of The Robotics Institute.

This paper provides an overview of the hardware and software organization of POPEYE, the CMU gray-level vision system. It includes a quantitative evaluation of the basic system with some discussion of projected enhancements by new board designs. Applications of the system in the performance of cellular logic operations[35, 36], piecewise gradient segmentation[19], automatic focusing and adaptive spatial filtering are also presented.

2. Hardware

There are a number of alternatives to consider in the design of a vision system. Some of the early work was geared to the use of general purpose computers coupled to a frame-buffer display system. Although this type of system offers advantages such as mass storage capabilities, extensive software libraries and good operating systems that hide the hardware from the user, it tends to be too slow for on-line applications such as

industrial inspection tasks. The main characteristic of such systems is that the operations must be performed serially in a single processing unit.

The extreme alternative is to dedicate a processing unit for each picture element[26]. Designs of this type have proven to be extremely fast but difficult to program, so they have found places only in laboratories or special applications. Other alternatives include pipelined and parallel multiprocessor architectures such as crossbar switches and time-shared busses.[37] The POPEYE vision system is a loosely coupled multiprocessor system under the MULTIBUS convention. Figure 2-1 shows the block diagram of the main subsystems and figure 2-2 shows a photograph of the current POPEYE vision system.

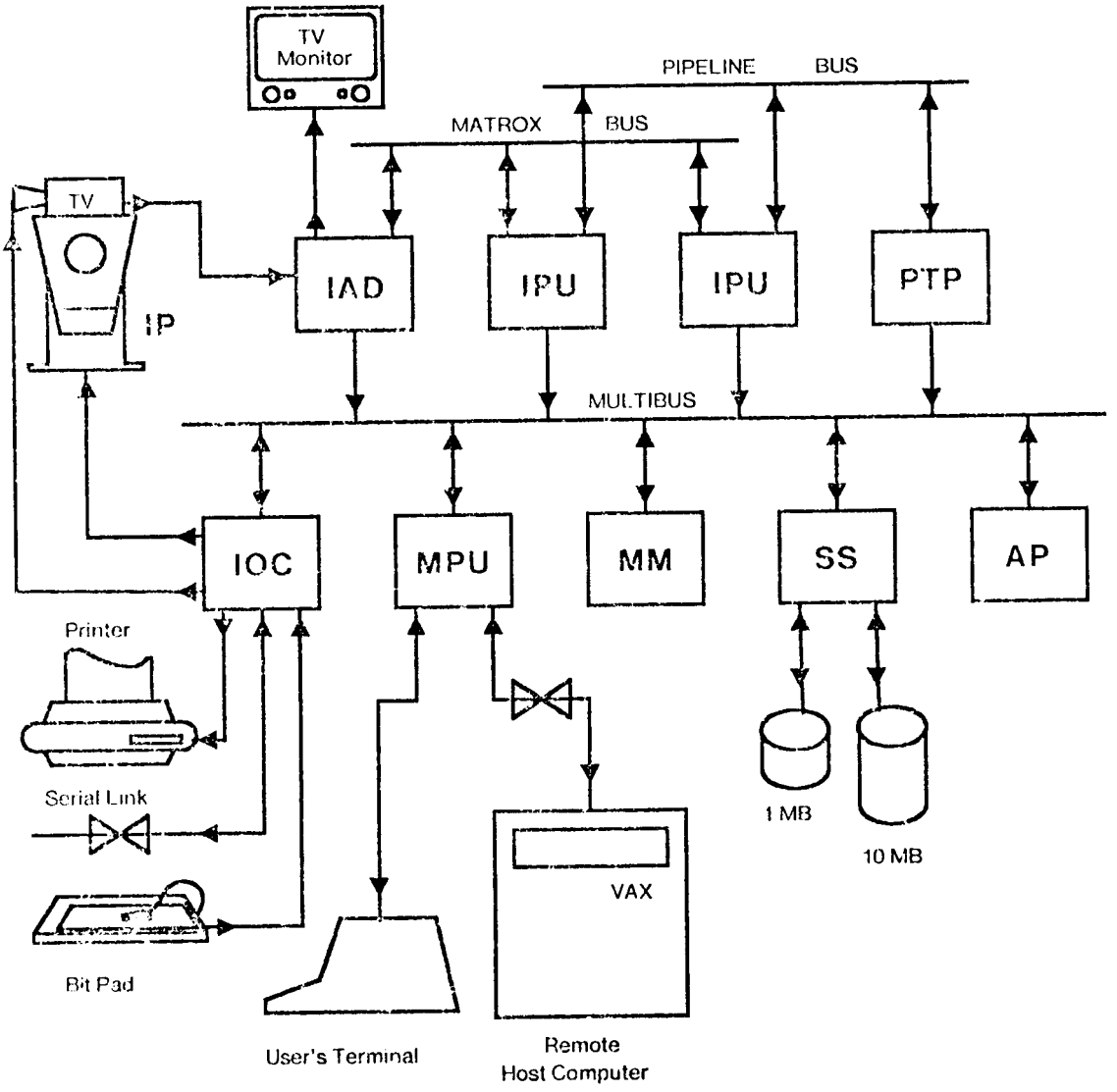

Figure 2-1: Hardware Configuration of CMU's POPEYE vision system

Figure 2-2: Photograph of the POPEYE vision system

2.1. Main Processing Unit (MPU)

An in-house design based on Motorola's MC68000 16/32 bit microprocessor, the MPU functions as the flow controller of the entire system. It features a 10MHz CPU, 8 KB of EPROM for the monitor (see the Software section) and 4 KB of RAM for the stack. It also features two serial lines and five counter/timers. The serial lines are typically used to communicate with the user's terminal and the host computer, a DEC VAX 11/750 running UNIX. Three of the timers are used by the system as a real time clock and the other two are available to the user.

The MPU's functions include downloading code from the host computer to the other processors, interaction with the user, real-time events and orchestrating the flow of information within the system.

2.2. Main Memory (MM)

Two memory boards, providing a total of 640 KB, comprise the system's main memory. The memory is divided into 128 KB (Central Data Corporation's *CDC-128K*) used for programs and system utilities, and 0.5 MB (Chrislin Industries' *CI-512*) used for data. Space on the latter board is obtained from system calls to a dynamic allocation package.

2.3. Secondary Storage (SS)

A 10 MB Winchester drive (Shugart Associates' *SA-1004*) and a 1 MB floppy-disk drive (*SA-800*) give the system 11 MB of on-line secondary storage. The disk controller, manufactured by Data Technology Corp. (type *DTC-1403D*,) may be connected to up to four drives. The data transfer is done via direct memory access (DMA) between the MM and the disk controller's MULTIBUS adapter (*DTC-86*). The adapter controls the transfer. Other features include copying data between the drives without going through main memory.

2.4. Input/Output Control (IOC)

The rest of the Input/Output (other than communicating with the user's terminal or the host computer) is handled by a board made by Monolithic Systems Corp. This z80-based I/O controller (*MSC-8007*) has 32 KB of dual-ported RAM which it uses to communicate with the MPU. The board's collection of I/O devices includes three serial lines (normally connected to a printer, a bit-pad and a general purpose serial link) and two parallel ports which are typically used to communicate with the Image Positioning subsystem described below.

The IOC has a floating-point processor, capable of 10000-40000 flops, which is used mainly by the on-board z80. 32 KB of EPROM will contain the I/O drivers and some low-level algorithms for the Image Positioning subsystem.

2.5. Image Acquisition and Display (IAD)

Four boards, all manufactured by Matrox Electronic Systems, Ltd., provide the capability of digitizing and displaying images in real time (60 fields per second). The frame grabber (an *FG-01*) digitizes a 256 x 256 pixel image directly from the TV camera with up to 256 levels of gray (8-bit quantization) in 1/60 of a second. It accepts its input from one of four cameras under software selection.

The 8-bit picture elements (pixels) are transferred via a fast bus, hereafter called the **Matrox Bus**, to the frame buffer (two *RGB-256* boards) which continuously displays its contents on a TV monitor. Each board holds four bits of the eight bit resolution. The frame buffer has both composite video and RGB outputs and thus it may be used to display color or black-and-white images. The color map is fixed by the hardware, which provides three bits for red, three for green and two for blue.

The last board of the IAD is a one-bit overlay plane (*MSBC-512*) used to nondestructively display cursors, viewport boundaries and other temporary objects. When an overlay pixel is set to 1 the corresponding area of the screen is at full brightness, regardless of the pixel's frame-buffer value.

2.6. Image Positioning (IP)

In order to add flexibility to the IAD subsystem, the TV camera was mounted on a pan/tilt head (*Vicon V300PT*) and fitted with a remote zoom/focus lens (*Vicon V12.5-75*). These two elements constitute the image positioning subsystem used in object tracking and automatic focusing algorithms. A small hardware interface connects the parallel port of the IOC to the standard controller (*V129-8PP*) provided by the manufacturers of the head and lens. This provides the user with control over the pan and tilt parameters of the head and the zoom and focus parameters of the lens. The pan/tilt head is large enough to hold two cameras for stereo vision applications.

2.7. Array Processor (AP)

An array processor was added to POPEYE for number crunching applications. The two-board set manufactured by Sky Computers, Inc (*SKYMNK-M*) is capable of up to 1 Mflops and it is utilized by the system to perform vector calculations and Fourier analysis on raw data.

The AP has a rather sophisticated DMA controller to retrieve the data and store the results in main memory. It is possible to specify not only the number of consecutive words (**n**) but a number of words (m) to be skipped before retrieving the next n words. The user may also specify the number of **(n + m)** combinations to be used in a single command. This complex addressing scheme is especially useful for image processing tasks.

2.8. Image Pre-processing Units (IPUs)

When implementing image pre-processing algorithms, one often has to deal with very large amounts of data and, while the operations tend to be simple and repetitive, it is necessary to perform them very quickly to achieve the required overall performance.

We are constructing two Image Pre-processing Units (IPUs), consisting of an MC68000 processor, an image page and a pipeline page (see Figure 2-3). Each of the two pages is 64 KB long so they can accommodate a 256 x 256 x 8 bit image. The image page may be loaded from, or dumped to, the Matrox Bus in 1/60th of a second. It is normally used to hold the input data to be processed by the MC68000 processor or the results of the pre-processing algorithm.

The on-board 12 MHz MC68000 has 32 KB of RAM from which it executes instructions. This memory and the image page are mapped into the MULTIBUS memory space so they may be loaded or read by the MPU. In a normal application, the MPU first downloads code from the host computer into this program memory so the IPU can execute it.

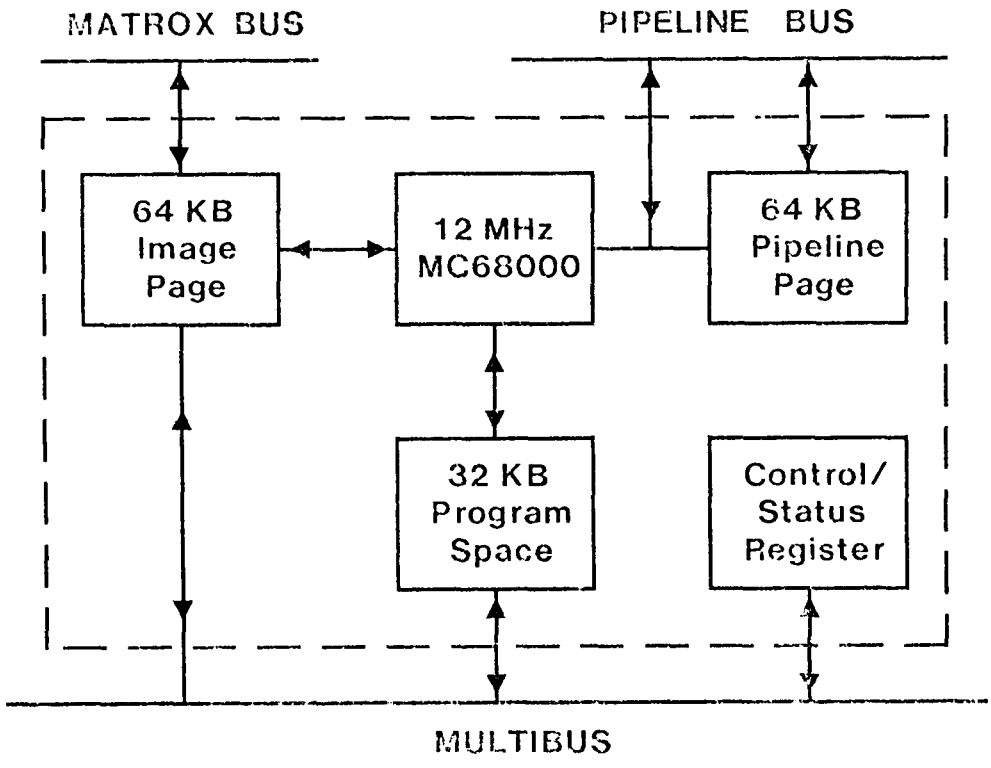

Figure 2-3: Block Diagram of the Image Pre-processing Units

The other 64 KB of memory, the pipeline page, is only accessible to the on-board MC68000 and a fast bus called the *Pipeline Bus*. It is thus possible to connect the two IPUs back to back by means of the pipeline bus. In such a configuration, one IPU would receive the raw image in its image page and perform a pre-processing algorithm storing the results in its pipeline page. The other IPU would take the data from that pipeline page and perform a second pre-processing algorithm putting the results in its image page from which they may be displayed in 1/60 of a second. This is possible because each MC68000 has access to the pipeline bus, and thus to the other IPU's pipeline page.

2.9. Programmable Transform Processor (PTP)

A number of vision algorithms require that an image be transformed either logically or mathematically. Most of these transforms are relatively straightforward, applying a number of simple operations to a neighborhood around the pixel being analyzed.

The PTP is a microprogrammable processor specifically designed to implement either logical or mathematical transforms over a programmable neighborhood. A block diagram of the PTP is shown in Figure 2-4. It is capable of convolving a 3 x 3 mask with the full image in less than 240 msecs or running a cellular-logic

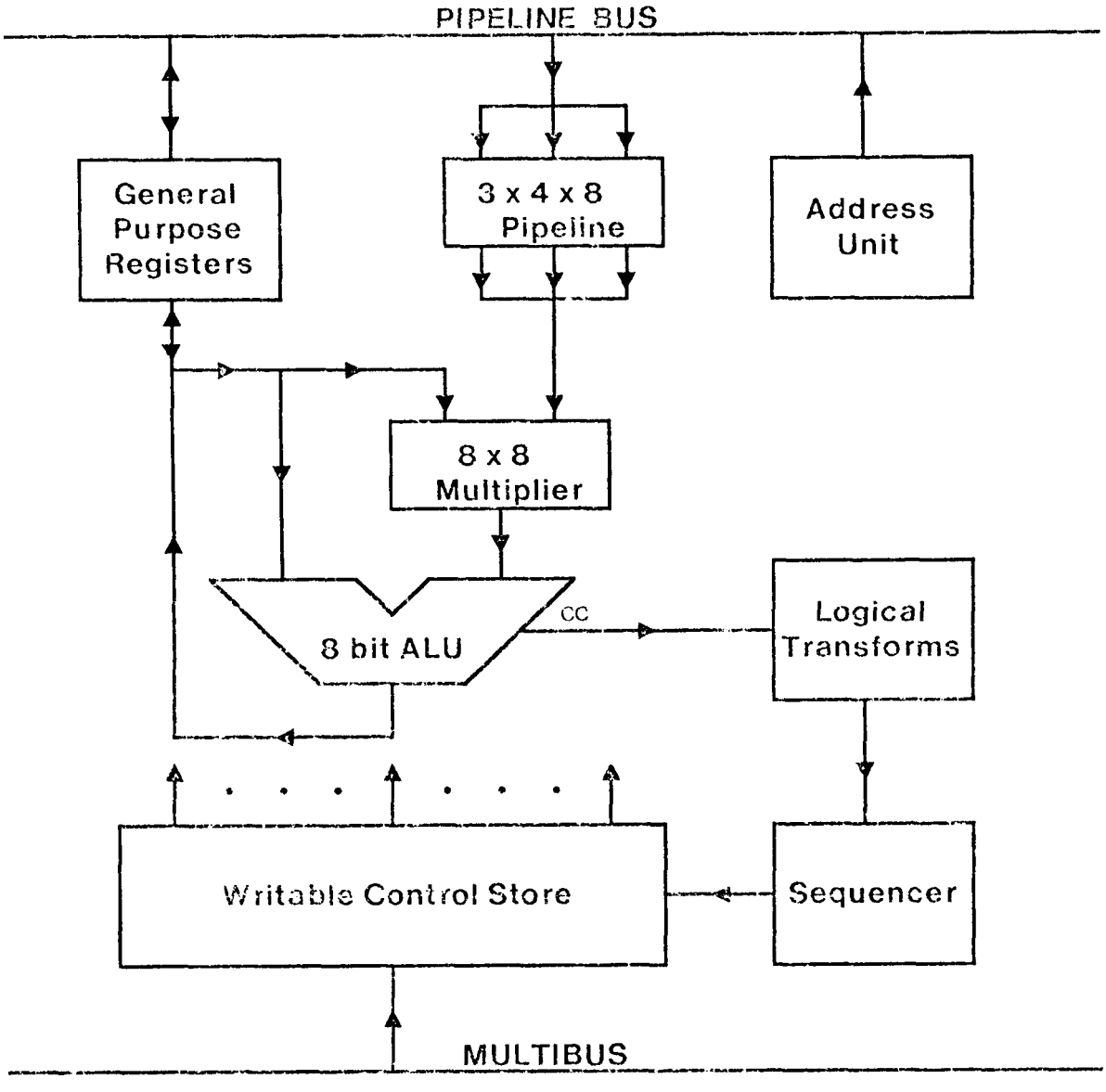

Figure 2-4: Block Diagram of the Programmable Transform Processor

cycle in little over 100 msecs. The design includes a 3 x 4 pixel pipeline, an 8 x 8 flash multiplier, an 8-bit ALU and a powerful neighbor-address generator which may calculate up to 16 neighbor-pixels' addresses in parallel to the main computations. The control store holds 1K 64 bit μwords and is mapped onto the MULTIBUS and it is loaded by the MPU during an initialization phase. It is implemented with very high speed RAM permitting typical microcycle times of less than 200 nsecs.

2.10. Future enhancements

In the future we will add a 10 Mb Ethernet controller to speed up the communication link to the host as well as to give the system access to a number of resources available at Carnegie-Mellon University. Within the Robotics Institute we will have a Three Rivers Computers Corp. *PERQ* and several special processors linked via the 10 Mb Ethernet. Also, a gateway to the 3 Mb Ethernet is planned which would link us to more than a dozen VAXen and other resources, including a 60 page per minute laser printer.

For color vision, we have acquired a filter wheel which will enable us to obtain three component color images corresponding to the three primary hues. A controlled-lighting environment is planned to perform critical experiments.

3. Software

The software for the POPEYE vision system can be divided into four levels: host level support, device level support, object level support and applications programming. (Refer to Figure 3-1.) Each level consists of several programs and subroutine libraries. The total software effort has grown to approximately 400 pages of code, written mostly in C, all of which was written, edited and compiled on the host computer. This machine serves as a support facility for several projects of this type, running C cross-compilers for four different machines. In addition, it is linked to CMU's Ethernet, allowing it to keep abreast of system software updates, bulletin board information and electronic mail traffic.

Much of the software for POPEYE was consciously patterned after similar components in UNIX. In several cases, we were able (or forced) to port source code from the VAX to the POPEYE system.

Software engineering practices are strongly adhered to throughout the vision system software, including manual entries for each program and subroutine, a header page for each module of source code and verbose and plentiful comments.

3.1. Host Level Support

3.1.1. Editing and Compiling

All the programs that run on POPEYE's main processing unit are written, edited and compiled on the host machine. Almost all the code is written in C, with only small utilities where efficiency is a major consideration being written in M68000 assembly language.

The C cross compiler package for the 68000 is very similar to the native C compiler for the VAX in that it consists of a translator, post optimizer, assembler, linking loader, and symbol table maintainer. The loader

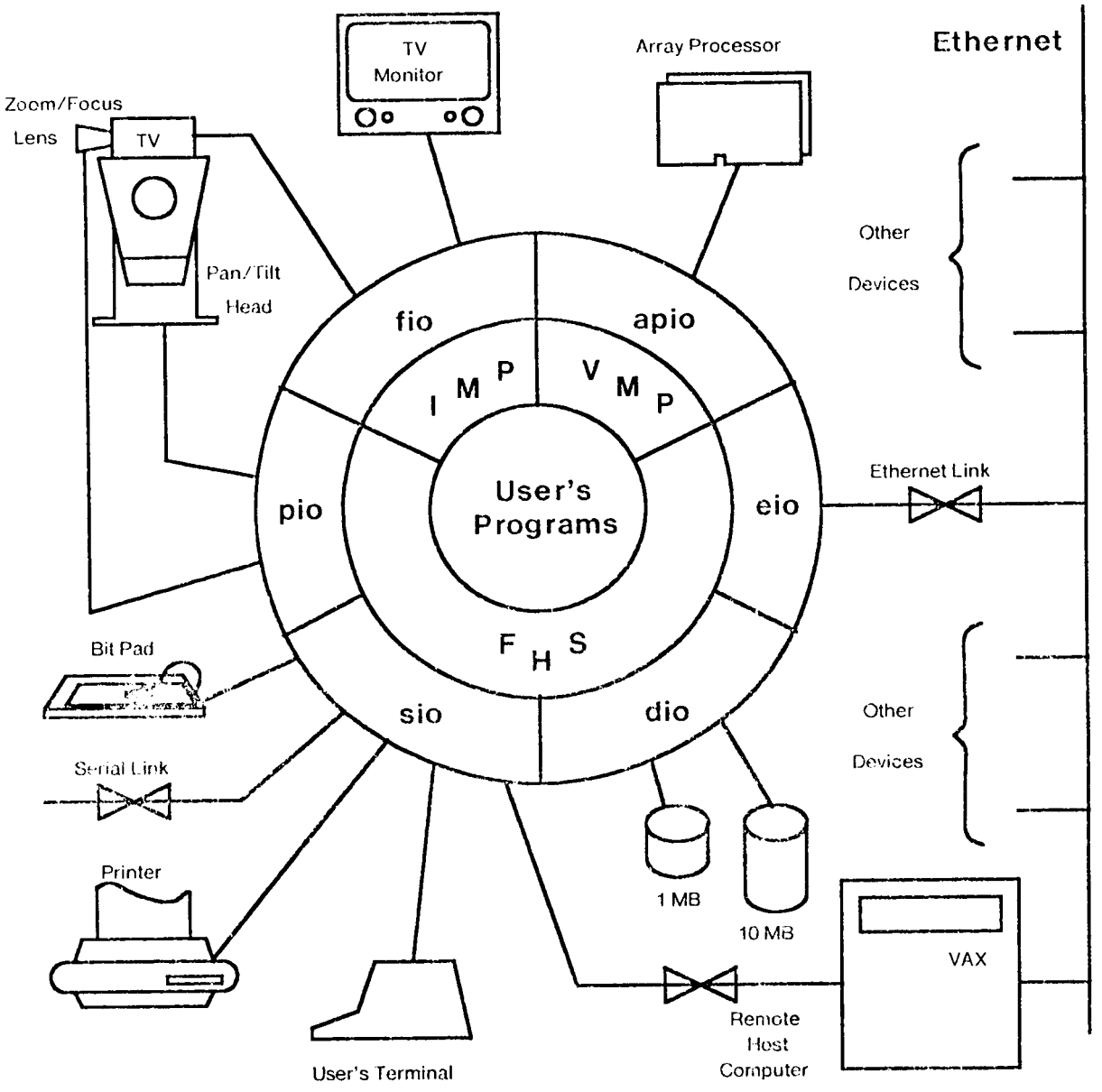

Figure 3-1: Software Configuration of CMU's POPEYE vision system

uses the same subroutine library format as the UNIX loader, which allows us to use the same archiver. The cross compiler loader also allows external symbol references to be resolved by searching the symbol table files from other programs, something which is very useful in generating programs for a single process, single user environment. Often, a program which tests the algorithm of the day may be changed, recompiled, downloaded and executed every few minutes, so it helps to divide the program into two segments. A small piece which contains only the algorithm implementation can be quickly recompiled and downloaded, while a second, larger piece containing support utilities such as image display subroutines can sit in main memory

unchanged. This is a great boon, as downloading code even at 9600 baud is painfully slow.

3.1.2. Debugging

Another important piece of host level support is the symbolic debugger. Building a debugger for our environment proved to be a much more complicated task than building a standard debugger, since the host machine must communicate with the MPU in the vision system, polling memory locations, stopping and restarting execution, single stepping either through assembly language instructions or through lines of source code and setting and deleting break points. Thus, the debugger is actually a distributed software system, or a "cross-debugger" (see Figure 3-2).

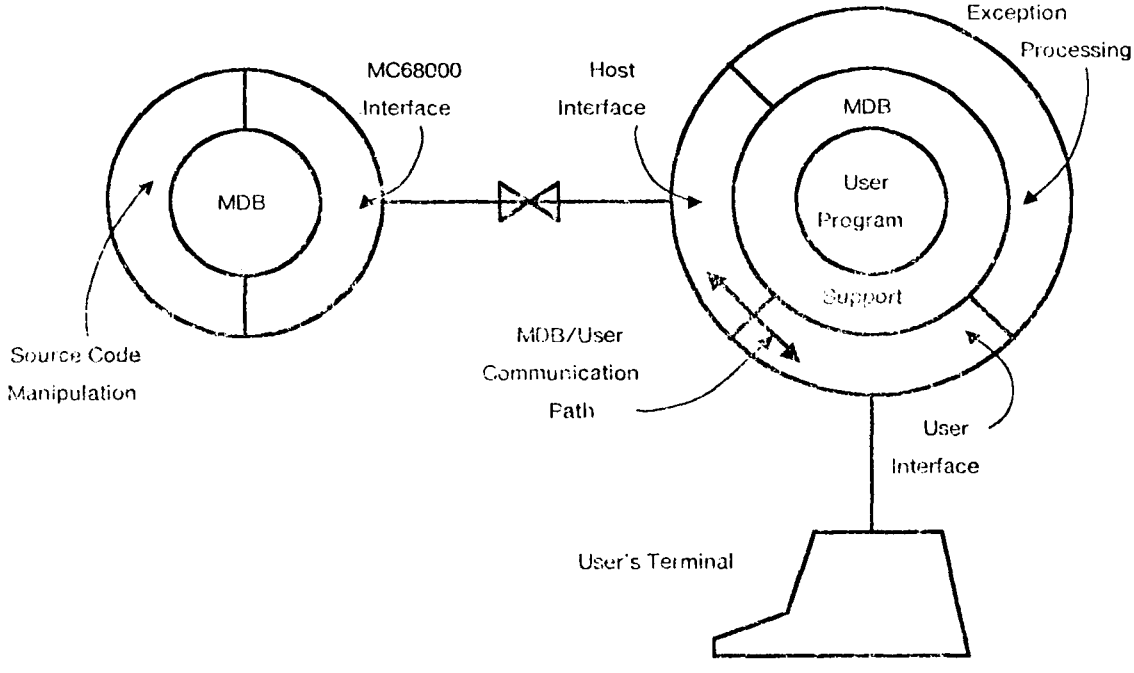

Figure 3-2: Representation of the Cross-Debugger System

At present, when a program dies unexpectedly, the monitor prints a cryptic diagnostic on the user terminal which shows the contents of the program counter, status register and possibly some other information. Given the address where the program died, the debugger will search the symbol table file for that program, figure out which subroutine contains the address and disassemble the subroutine. Like its UNIX counterpart, the debugger can manipulate several programs with their associated symbol tables and executable segments.

The compiler also supports the debugging effort by placing labels in the assembly language output that correspond to the beginning of each line of source code. This allows the debugger to execute the program on

a line by line basis.

Although incompletely implemented at present, future plans include extension of the debugger to its full interactive capability.

3.1.3. Downloading and Uploading

At the end of the compilation process, an extra phase of the C cross-compiler produces an ASCII version of the executable program in Motorola VERSABUG format. At the request of the MPU, the host machine dumps this file over the serial line connecting the two processors. The MPU executes a subroutine which reads the file, decodes the VERSABUG records and loads the executable code into main memory. This again is a distributed software system, though not nearly as complicated as the debugger.

In addition to trading in VERSABUG format, the host machine also implements a generalized upload/download protocol designed to support the debugger communications and the transfer of image data. The black and white camera attached to POPEYE can be used with color filters to obtain component color images, which can then be uploaded to the host machine, recombined and displayed on the Grinnell color frame buffer system.

3.1.4. Language Development

Many of the applications programs for POPEYE are simple enough to need only a single character menu driven input paradigm. In certain cases, however, the input is structured enough to warrant a parser and/or a lexical analyzer. The host UNIX system has tools for building just such items, and which output code in C. With only minor modifications relating to i/o, this code can be cross-compiled and executed on POPEYE's MPU. An example of a program which uses both the parser generator and lexical analyzer generator will be described later.

3.1.5. Hardcopy

Often, hardcopy of some entity such as an image, a line scan or a histogram plot is desired. The high resolution laser printer connected to CMU's Ethernet is used for this purpose. The information is uploaded to the host in one of several data formats, converted by some program or sequence of programs into a printable file and finally shipped over the Ethernet to the printer. The printable files can also be included as illustrations in documents. Because of printer limitations, images must be binarized before being printed.

3.2. Device Level Support

3.2.1. The Monitor

At the heart of the device level software lies the monitor. This program is stored in EPROM in the MPU and is executed on power-up and on receipt of fatal exceptions such as bus errors. The monitor provides enough capability to download and execute programs through the implementation of the following features.

- TALK-THRU MODE. The monitor can make a software connection between the two serial lines on the MPU board to allow the user access to the host as if there were no vision system between the two. This is the mode of operation during logins, editing and compiling. After editing and recompiling a program, the user can exit talk-thru mode and return to POPEYE.

- DOWNLOADING. When the user wishes to download and execute a program, he gives the name of the program to the monitor. The monitor requests the program from the host and enters download mode. During the downloading process, the monitor takes apart the VERSABUG format file produced by the cross-compiler and sets the executable code into main memory. If desired, the monitor will automatically execute the program at the end of the file transfer. If the execution of the program is successful, the monitor regains control in normal mode after termination. If not, the monitor regains control through an exception handler, urps a message to the user terminal and again returns to normal mode.

- DEBUGGING. For simple hand debugging jobs, the monitor allows the user to examine and change the contents of memory on an 8, 16 or 32 bit word basis. In the future, the monitor will also support the lowest level of the cross-debugger communications protocol. This is a particularly difficult problem since communications between the user terminal and the host must be maintained while silently allowing the debugging program on the host to access the contents of main memory. (Refer back to Figure 3-2.)

- DYNAMIC MEMORY ALLOCATION. To make the applications programs smaller, cleaner and easier to write, a dynamic memory allocation package was installed in the monitor. The package is initialized before the execution of each program and provides whatever space the program may request for temporary storage. For example, the image manipulation package, to be described shortly, uses the allocator to obtain space for storing image data in main memory.

To maintain independence of hardware configuration, the monitor knows nothing about any hardware outside of the MPU.

3.2.2. Device Drivers

The remainder of the device level support layer is a collection of device drivers for the various hardware subsystems described in section 2. The drivers are stored on the disk as files and read into main memory when a particular device is opened.

- The serial i/o package communicates with the terminal, host, printer, bitpad and general purpose serial line. Serial i/o is interrupt driven.

- The parallel i/o package communicates with a special purpose hardware interface to provide the MPU with control over the pan/tilt head and the motorized zoom lens. Thus, a user program can independently control the pan and tilt angles and the zoom and focus parameters of the lens. A tracking program which exercises this control will be described insection 5. Parallel i/o may be interrupt driven or polled.

- The disk i/o package handles the lowest level of data transfers to and from the disks and consists of a primitive space manager and the interface to the DMA controller. A copy command is available to make disk backups simple.

- The frame i/o package talks to the image acquisition and display subsystem, controlling the transfer of data to and from the frame buffer and main memory and the grabbing of frames from up to four television cameras.

- The array processor i/o package merely sets up DMA commands for the hardware.

3.3. Object Level Support

The object level support layer consists of the vector manipulation package, the file handling system and the image manipulation package. Together, these three pieces provide user programs with an elegant interface to the hardware capabilities of CMU's POPEYE vision system.

3.3.1. The Vector Manipulation Package

The vector manipulation package is the simplest of the three pieces and provides access to the capabilities of the array processor subsystem without the headaches of talking directly to the hardware. The hardware is manipulated at the lowest level by vendor supplied microcode which resides on the MULTIBUS boards. Above the microcode lies the device driver, and above the driver lies a layer of assembly language subroutines, supplied in part by the vendor as a library. These routines implement functions such as data format conversion, vector algebra routines and the FFT algorithm.

3.3.2. The File Handling System

POPEYE's file structure is one of the parts that was consciously modeled after UNIX. In the spirit of UNIX, it unifies the myriad of details relating to both disk storage and program i/o into a single framework. This allows the devices attached to the system to be regarded as files. Input to a running program (a process) always comes from a file, but often the "file" actually points through to the user terminal. Pulling the next character from the input causes the serial line device driver to get a character from the terminal. Likewise, the output from a process always goes to a file, but again, the file could actually be the terminal.

Our primary motivation for attaching a disk controller to the vision system was the need to store images. In addition, once the size of our applications programs grew to the point that downloading became uncomfortable, the natural thing to do was to store the programs on disk. Our first inclination was to buy a UNIX-like operating system for the 68000 and be done with worrying about files. Unfortunately, often an operating system slows down the raw speed of a computer system, thus diminishing its performance. It was decided that POPEYE would be a single user, single task, machine. In addition, after researching the details of file storage on UNIX, we decided that certain aspects of the file system were unattractive. We had grown accustomed to the fast image access that comes from contiguous file storage. In UNIX, files can be fragmented and strewn about all over the disk. In a multiprocess, multiuser environment where garbage compacting is impractical, this storage scheme makes sense. In our environment, however, speed of access is more highly valued. What we ended up with is a file system with the convenient tree structure of UNIX, along with the option of specifying files to be contiguous.

3.3.3. The Image Manipulation Package

The last and largest piece of the object level support layer is the image manipulation package, a subroutine library which provides primitives for the manipulation of images on disk, in main memory and on the screen. The following conventions have been established. (Refer to Figure 3-3.)

A collection of pixels on disk is called an **image**. To the file handling system, an image is just another file, save that it is stored contiguously. The contents of the file can be created by any means: grabbing frames from the camera, processing another image and random number generation are all valid means of image creation. Presently, images are constrained to be a multiple of 16k bytes in length. This means that a 256 x 256 pixel image typical of POPEYE is of length 4, while a 512 x 512 pixel image typical of the Grinnell system attached to the Vax is of length 16.

To process an image, the pixels must be moved from disk to main memory, where they reside in a **window**. Windows can be of arbitrary size and shape. The pixels are again stored contiguously. To aid in processing a window, there exists another object, a rectangular subset of the pixels in a window called a **pane**.

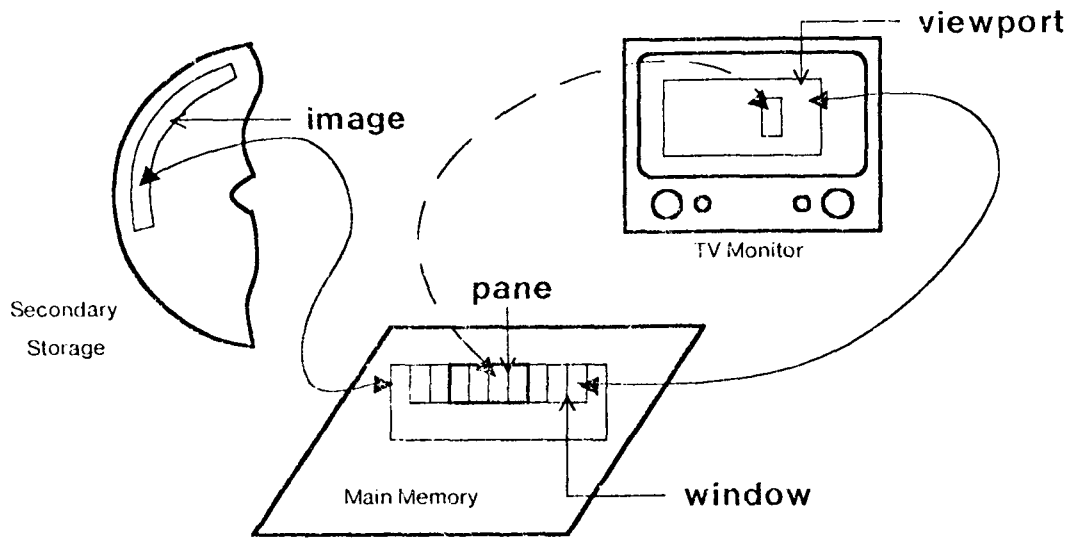

Figure 3-3: Representation of the Image Manipulation System

Once the pixels is a window have been moved to main memory, the Pane can be moved about within the window, thus eliminating the need to reread the pixels from the disk or from the frame buffer each time the area of interest changes.

To view the contents of a window on the monitor, a **viewport** is created and linked to the window. Viewports must have identical dimensions to the windows to which they are linked, but are free to occupy any position on the screen. The size and location of a viewport may be changed interactively by using the cursor movement commands of the terminal. Several viewports may be linked to a single window. Changes made to the contents of a window will be reflected in each viewport to which it is linked.

The last type of object, the Cursor, is used for pointing to specific locations on the screen.

3.4. Application Programs

The remainder of the vision system software is a collection of application programs and subroutines. A large piece in this category is a subroutine library full of garden-variety image processing algorithms such as high pass and low pass filter convolution kernels, the Sobel edge detector, a temporal averaging subroutine to reduce the effects of camera noise, histogram manipulation subroutines, a contrast enhancement package, binarization and cellular logic transform operators and a temporal differencing subroutine. All of these subroutines operate on one or more of the objects described previously.

Above this rather standard library is a collection of more advanced image processing algorithms which

we have written for our own purposes.

- The standard binary cellular logic idea has been extended to operate on grey scale images, resulting in Adaptive Cellular Logic, or ACL. This is useful for performing a more intelligent binarization than can be obtained by simple thresholding as well as for edge detection and blob smoothing in grey scale images.

- Several data compression schemes have been implemented for the purpose of reducing the amount of processing necessary to perform pattern recognition to a level compatible with real time control. This is the subject of section 5.2.

- A small interpretive language for multipass image filtering has been specified and implemented. This is described in section 5.4.

- A large support program of the type described earlier in conjunction with compiling and downloading has been provided as a base for algorithm development. This program contains most of the subroutines described above, including the software for controlling the pan/tilt head and zoom lens, so that test programs may remain small. The support program is capable of downloading and executing test programs without returning to the monitor, and so does not have to be reinitialized after each program call.

- A general purpose command interpreter package has been written to make the construction of menu driven programs as painless as possible. The package includes facilities for recognizing and executing commands, changing variables during execution and on-line help information. As mentioned earlier, users intending to build programs for general use -- especially demonstration programs -- are encouraged to use this package. Thus, some uniformity between pieces of application software is achieved. First-time users have little or no trouble running demonstration programs on POPEYE.

- A simple tracking algorithm utilizing the image positioning system was implemented to see how close the processing power of the vision system could pull toward real time. The program grabs a frame from the camera and simultaneously binarizes and computes the area and center of energy while reading the pixels from the frame buffer. The area and center of energy are compared to their previous values and the differences used to deliver control signals to the image positioning system. Movement in the x direction generates pan signals, movement in the y direction generates tilt signals and movement in the z direction (change in area) generates zoom signals. While processing the full 256 x 256 pixel frame size, the sampling period is just under one second and all processing is done in the MPU. To achieve faster rates, some of the computations should be transferred to the IPUs and the PTP.

- Two automatic focusing algorithms have been implemented. These will be described in section 5.3.

In many industrial production environments it is desirable to use automated vision systems for inspection jobs which are considered dangerous, boring or unreliable when carried out by humans. A number of the application programs have come from the implementation of industrial inspection algorithms for these tasks. Typically, a concept demonstration is carried out that evaluates speed of performance, computational complexity and cost of implementation. The application packages written for this system have served not only to demonstrate the feasibility of specific inspection algorithms, but have also driven the software development of the system to a significant extent. Many of the amenities now present on the system were originally developed for specific demonstrations. Conversely, several of the image processing algorithms developed and implemented for research purposes have found their way into industrial inspection packages.

3.5. Future Plans

The following pieces of software are expected to be integrated into CMU's POPEYE vision system environment in the near future.

- The Ethernet i/o package will provide a device level interface to the 10 megabyte Ethernet when the capability becomes necessary. The Ethernet will be needed for high speed data transfers between POPEYE and the Perq. The Perq has a high resolution bit mapped screen and a microprogrammable processor, making it a desirable complement for the vision system.

- The IPUs installed in the system require simple device drivers. The existing software for downloading code will be used to load the 32 kB program space. (Refer back to section 2.8).

- Since the PTP is a microcodable machine, it requires a microassembler. This is a medium sized development project. The microassembler should provide for the symbolic manipulation of microinstructions and perform rudimentary error checking to prevent the programmer from damaging the hardware. In addition, we plan to define a microsubroutine format for use with an archiver and linking loader so programmers may build libraries of useful transform subroutines.

- After all the hardware, device drivers and support software becomes operational, we will be faced with a familiar but difficult problem: programming a multiprocessor system. This is a major research problem we do not expect to solve the first time around. We would like to see support for multiprocessing in the form of an editor, a compiler and a debugger. Ada is being considered as a language for multiprogramming, although a custom extension to C may be in order. Our first approach, however, will be to write some applications software and use it to evaluate the extent to which a mutiprocessor support is needed.

4. Performance

It is always difficult to evaluate a computer system since every architecture has its strong and weak points. The problem is more complex if the system to be evaluated is a multiprocessor, as in our case. In our discussion about the system's performance, we chose to evaluate the system in the context of its applications.

The vision system was specifically designed to be used in image processing tasks so it seems useful to compare it with other systems used in those tasks. When appropriate, we will perform comparisons with a display-type system consisting of a frame buffer (like those manufactured by Grinnell or De Anza) and a general purpose computer (typically a single-user PDP-11 or a multi-user VAX 11). We will also try to compare POPEYE with an analysis-type system such as those manufactured by Vicom or Quantex that execute a number of pre-defined algorithms very quickly.

It is important at this point to note that since the POPEYE vision system was designed to be a tool in the development and testing of vision algorithms, it was essential that it be programmable. The system was not intended to be used for any other purpose, unlike the Vax host of the display-type system. With this in mind, we'll look at four image processing tasks: convolution filters, adaptive modeling, histogram modification and connectivity.

4.1. Convolution Filter

In this type of problem, a 3 x 3 mask is convolved with a 256 x 256 pixel image. This is a repetitive operation that may be implemented in hardware. Since it is commonly used, most analysis-type machines have such a hardware device. Therefore they are able to perform the convolution in real time (30 msecs).

Assuming that the image has been acquired already, the vision system is able to do the convolution and display the result in 300 - 350 msecs which compares favorably with a display-type machine. Our Grinnell-VAX 11/780 combination takes anywhere from 2 to 5 secs of CPU time, depending on the system load.

4.2. Adaptive Modeling

In this task, we would like to model the image using some data dependent model. An example would be a 2-D auto-regressive (AR) model. The data dependency of the algorithm does not allow an efficient hardware implementation, so the analysis-type machines do not perform well. It may be necessary to piece together the algorithm from lower level routines but this assembly seldom allows the user to efficiently utilize the pipelined architecture of the system. The display-type machine does not perform any worse than in the convolution problem since both tasks must be programmed in software. Again the system's load will determine its performance.

The POPEYE vision system offers a few advantages over the other systems: First, due to its large main memory space, it can keep the entire image in RAM, allowing a floating-point number per pixel if necessary; second, the PTP may perform the raw computations on the image while an IPU determines the model parameters; third, the user still controls the data flow through the MPU so intermediate results may be made available to him.

4.3. Histogram Modification

In this task a pixel by pixel (or *point*) transformation is done on the image. Unless the transformation is fixed and doesn't depend on the raw data, two phases are necessary: calculation of the histogram and pixel modification.

An analysis-type machine could implement the two phases in a pipeline of processes, making it possible to achieve real-time rates (30 msecs) unless the modification function is complex and data dependent. In that case there is an intermediate step of calculating the function which would be handled by a programmable processor. For the display-type system, the user must program both phases separately and probably write temporary files between them; although easy to do, this approach is time consuming.

CMU's POPEYE vision system would use one IPU to calculate the histogram and the modifying function while another uses the results to perform the pixel modification. The two IPUs would then operate as a pipeline. If the modification to be performed is simple equalization, processing times as low as 50 msecs per image may be obtained.

4.4. Connectivity

In this problem we try to decide whether a pixel belongs to a cluster of pixels or not. A criterion, typically similarity in intensity value, is used to determine if a pixel is part of any of the known clusters. This is a data dependent operation and is therefore difficult to implement in hardware unless the image is binary. A display-type system is programmed to perform the algorithm and its execution speed depends only on the raw speed and load of the host computer.

The implementation in the POPEYE vision system is straightforward due to the logical transform operations available in the PTP subsystem. The PTP will execute an optimized connectivity algorithm in less than 100 msecs.

4.5. Conclusions

It was shown that POPEYE compares very well with other architectures when dealing with image processing tasks. Even though it is in general slower than the analysis-type systems, its programmability makes it an ideal candidate to evaluate different vision algorithms. A few examples will be presented in section 5.

It should be mentioned that even though the display-type system exhibited lower performance than the other two systems, it is often supplied with a library of functions directly callable from an application program. This type of system is also not limited by memory which makes it very well suited to off-line image processing like satellite cartography or multi-color imaging such as that used in medical applications.

On a system like POPEYE, the user must develop all the software (at least once) which often takes a considerable amount of time and effort. One of the advantages is that it is possible to clone similar systems — possibly scaled down versions — to be used in the field.

5. Examples

5.1. Cellular Logic Operations

A large number of image processing problems may be solved with simple binary images. The main problem with binary vision systems is that light variations affect the choice of threshold. The vision system, being a gray level system, deals with these problems in a very simple way: it presents the gray level image to the user, allowing him to choose the threshold based on any criterion he wants. Furthermore, the image is typically kept with the full 8 bits of resolution so another threshold may be chosen at a later time.

One of the reasons why someone may want to solve a problem via binary vision is that all the posible operations with binary pixels are boolean in nature and thus capable of being performed in hardware. Preston et al.[38, 36] have defined a number of elementary neighborhood operations for binary images. They are based on two local measures on a neighborhood: the **factor number** (*f-num*) and the **crossing number** (*c-num*).

In the local neighborhood of a pixel, the f-num will be the number of 1s found while the c-num will be the number of 1-0 or 0-1 transitions found while traversing the neighborhood in the clockwise direction. Based on the f-num and the c-num of a pixel (say u_{ij}), two boolean variables f_{ij} and c_{ij} are defined as

$$f_{ij} = \begin{cases} 1 \text{ iff } (f\text{-}num \text{ of } u_{ij}) > \varphi \\ 0 \text{ otherwise} \end{cases}$$

$$c_{ij} = \begin{cases} 1 \text{ iff } (c\text{-}num \text{ of } u_{ij}) \geq \psi \\ 0 \text{ otherwise} \end{cases}$$

where $0 \leq \varphi \leq 8$ and $0 \leq \psi \leq 9$ are the two thresholds that determine the properties of the particular cellular

logic operator (CLO). The two most common CLOs are the reduce (RED) operator and the augment (AUG) operator.

The RED operator is defined by the boolean equation

$$v_{i,j} = u_{i,j} \wedge (f_{i,j} \vee c_{i,j}).$$

Note that due to the AND operator, only pixels which were originally 1 may change (to 0). If we use the convention that a region consists of 1s embedded in a background of 0s, the number of pixels in a region may only be reduced (hence the name of the operator). The inverse operator (AUG) may only change pixels that were 0 (to 1) and is defined as

$$v_{i,j} = u_{i,j} \vee \neg (f'_{i,j} \vee c_{i,j})$$

where $f'_{i,j}$ is f-num redefined so it counts the number of 0s in the neighborhood. That is,

$$f'_{i,j} = \begin{cases} 1 \text{ iff } (MAX - (f\text{-}num \text{ of } u_{i,j})) > \varphi \\ 0 \text{ otherwise.} \end{cases}$$

Here MAX is the number of pixels in the neighborhood. Preston[36] has shown the behaviour of the RED CLO with different threshold combinations.

The PTP has been designed to execute both CLOs very rapidly (around 100 msecs. per CLO over a 256 x 256 pixel image). Furthermore, we are currently studying the extension of the cellular logic ideas to gray level images and the PTP will be just as fast with gray level data. In the next section we'll present an example that utilizes the cellular logic operations.

5.2. Gradient Segmentation

POPEYE has been used to implement the Piecewise Gradient Segmentation Algorithm[19] illustrated in figure 5-1. The algorithm consists of six major steps.

1. ONE DIMENSIONAL FEATURE EXTRACTION. The algorithm starts by extracting one dimensional features from the original image. On each of the two major directions, along rows and columns, the image is analyzed. The image is modeled using fixed-length *blocks* of pixels to make the procedure less computationally expensive. For each block we calculate the mean intensity, the standard deviation from the mean and the slope of the best linear regression fit to the pixels of the block. This slope is related to the intensity gradient component in the modeled direction. This first step is implemented on the IPUs with each one modelling in one of the two major directions.

2. GRADIENT TO INTENSITY MAPPING. The output of the previous step is an array of *models* for each of the two analyzed directions: horizontal and vertical. From the slopes of the linear regression fit we generate a **slope map**, an intensity display of the model slopes where the largest positive slopes

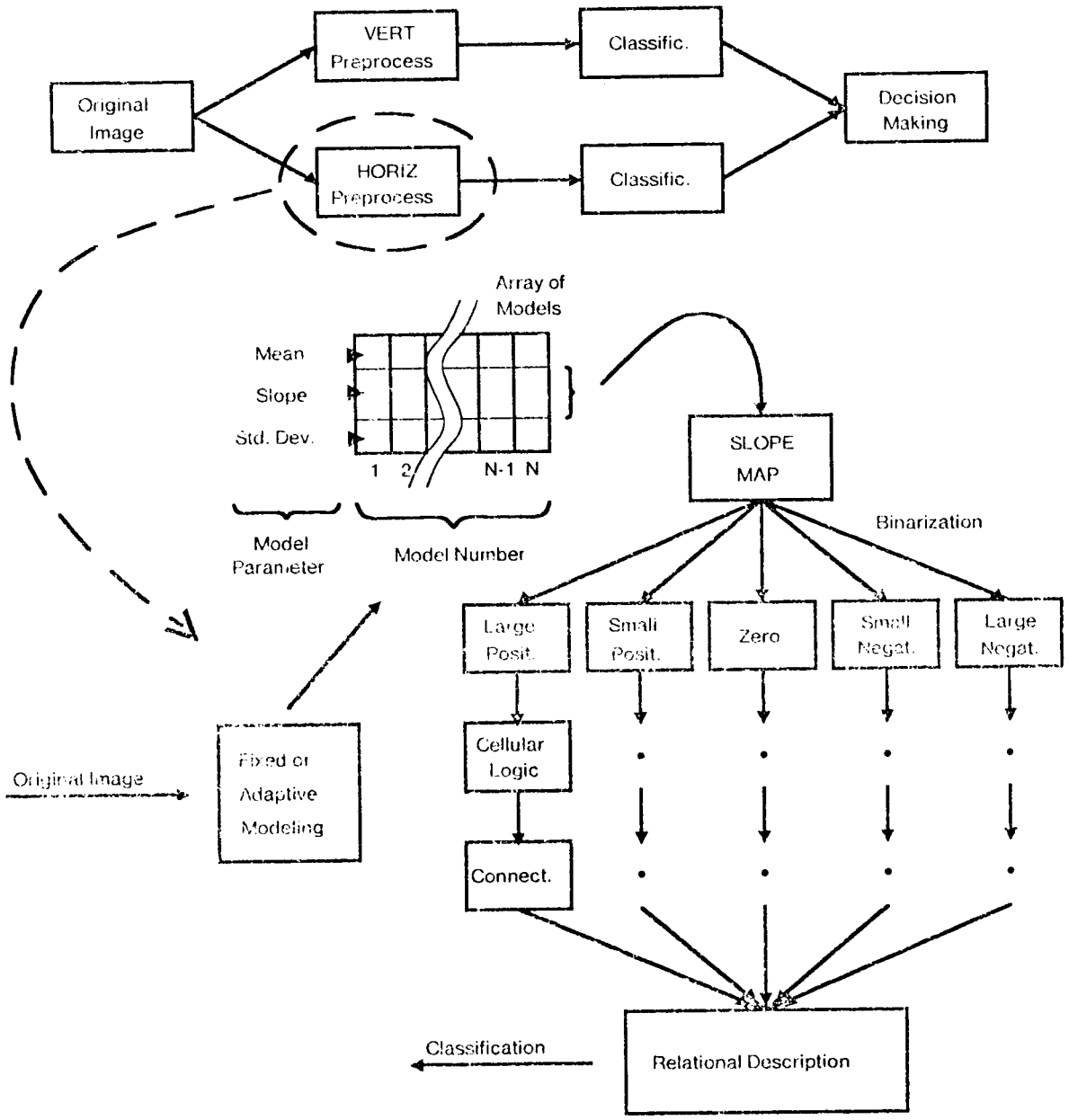

Figure 5-1: Block Diagram of the Piece-wise Gradient Segmentation Algorithm

are assigned the maximum brightness value of 255 and the largest negative slopes are assigned the minimum value of 0. Thus the pixels with a value of 128 belong to regions of constant intensity (no intensity slope). This step and the previous step are implemented simultaneosly on the IPUs.

3. THRESHOLDING OF THE SLOPE MAP. Each slope map is next thresholded to obtain up to five binary images corresponding to regions of zero slope, small positive and negative slopes and large positive and negative slopes. Although preliminary results have shown that the threshold is not

strongly dependent on the lighting conditions, it is nevertheless a data-dependent operation. The thresholding is done by the PTP at the same time it performs the first cycle of the next step: cellular logic operations.

4. CELLULAR LOGIC OPERATIONS ON THE BINARY IMAGES. This step uses the cellular logic operations described in the previous example. An AUG cycle with factor number of two followed by a RED cycle with the same parameters are done first to filter spurious blocks set to 1 by noise or inaccuracies in the modeling. Then eight AUG cycles with factor number of four followed by eight RED cycles are used to smooth the ragged regions obtained from the simple thresholding. This stage in the algorithm is performed in the PTP as discussed previously.

5. CONNECTIVITY ANALYSIS. Once the regions have been cleaned up, we proceed to extract their two dimensional geometrical features (area, perimeter, center of gravity, first- and second-moment invariants and first cross moment) along with a description of their spatial relations with one another. A fast one-pass algorithm has been designed to be used in the PTP as discussed in the performance section. The IPUs retrieve the results from the PTP and add to them the typical model parameters (mean intensity and standard deviation) so the MPU can retrieve all the information from the IPUs' image page.

6. GENERATION OF A RELATIONAL DESCRIPTION. Finally, a structural description of all the slope regions (up to five in each direction) is formed in memory by the MPU. This representation may be used to classify an object, determine its orientation or even perform scene interpretation as explained in reference[19].

Figure 5-2 shows the photograph of a paper cup lighted from one side. It is easy to see how the shading makes it impossible for simple thresholding to provide an adequate representation of the object. The figure also shows two of the five possible regions obtained from the horizontal models, they correspond to the small positive and negative slopes.

5.3. Automatic Focusing

Several automatic focusing algorithms have been used by various researchers in the past, all of which depend on a quality of focus criterion whose value is monotonically related to the high frequency content of the image. It is usually assumed that the point of best focus lies at the point of largest high frequency content. Horn[39] at MIT used a one dimensional FFT whose input points were circularly arranged in the image. Tenenbaum[40] at Stanford used a thresholded version of the Sobel gradient operator. Both were successful.

Several focusing methods are described below.

- HISTOGRAM ENTROPY MINIMIZATION. The histogram is tallied over a window of the image and its entropy computed. The sharper the focus of the image, the more definite the peaks in the

26

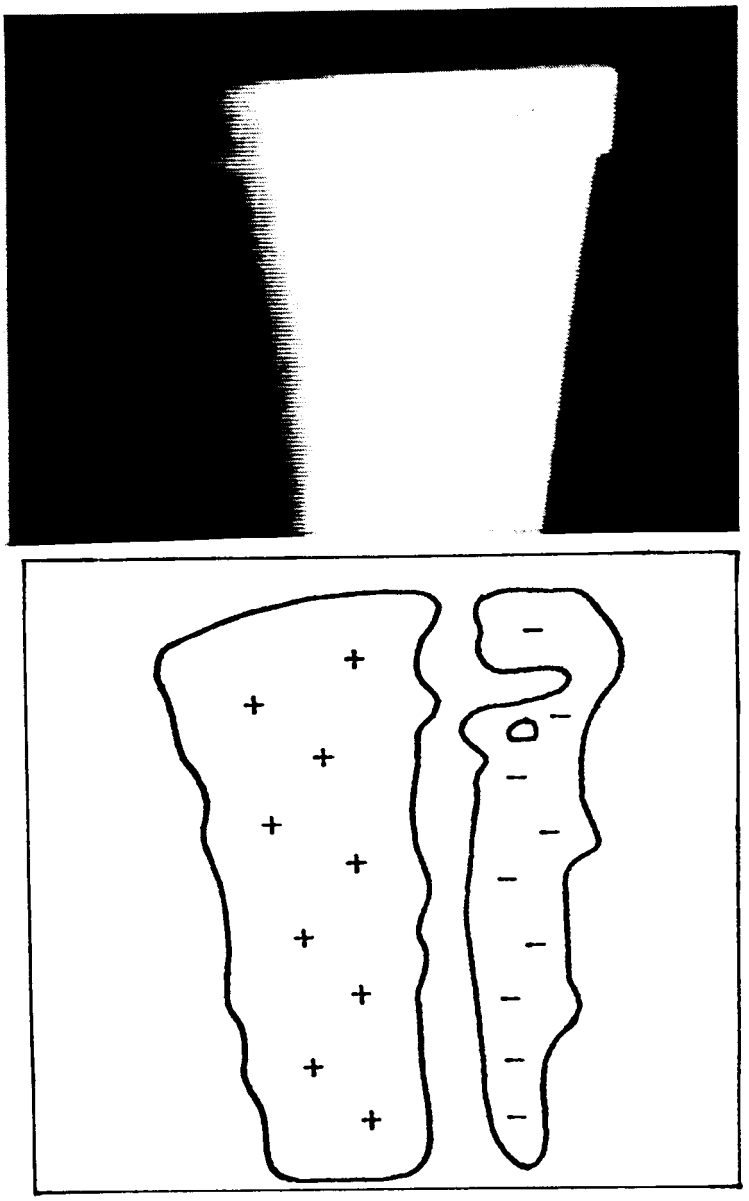

Figure 5-2: Small positive and negative slope regions of a paper cup (photo).

histogram become. The entropy, a measure of the "randomness" of a probability density function, is related to the shape of the peaks. In image processing, we use the histogram as an estimate of the probability density.

- HIGH FREQUENCY CONTENT MAXIMIZATION. All the focusing algorithms described here somehow depend on high frequency content, but none so obviously as the Fourier Transform. The usual scheme is to compute a one or two dimensional FFT, estimate the power spectrum density

from the squared magnitude of the FFT, sum the high frequency terms, and then maximize the sum by refocusing.

- THRESHOLDED GRADIENT MAGNITUDE. The steepness of dark to light and light to dark transitions in an image is dependent on the quality of focus. In two dimensions, the steepness is found by computing the gradient. By summing the gradient estimates over a window of the image, another estimate of the quality of focus is obtained. Unfortunately, since the gradient sum is constant by definition, the gradient estimates obtained at each point must be thresholded, thereby making the operation nonlinear. The nonlinearity makes the algorithm difficult to analyze.

- ADAPTIVE SEGMENTATION. One of the newer schemes for describing an image has been developed recently here at CMU, and is referred to as adaptive segmentation. This is a generalization of the gradient segmentation algorithm described previously.

  Typically, an image will contain large homogeneous sections. The general idea of segmentation is to cluster all the pixels in these sections into one bin, thereby reducing the amount of data which needs processing. The hard part is defining what we mean by homogeneous. Several successful ideas have been tried so far, and some seem to be applicable to focusing. In particular, descriptions that yield information concerning the variance of the pixel values in certain areas can be used to *extremize* the variance, thereby focusing the input image.

- CELLULAR LOGIC. One of the most attractive features of cellular logic is its deftness at edge detection. Edges are the single most important features of images which strive to be in focus, and successful attempts at automatic focusing using cellular logic have already been made in the image processing laboratory of a nearby hospital. The insights gained from study there are being applied to the focusing problem at CMU.

The histogram entropy and thresholded gradient magnitude algorithms have been implemented. Due to aliasing on the spatial frequency domain, the histogram entropy algorithm is useful only in the region near the point of best focus, but runs very quickly. The gradient algorithm is slower by a factor of approximately 5, but focuses as well as humans can.

5.4. Adaptive Spatial Filtering

Often an image has enough noise in it to foil whatever algorithm is attempting to make sense of it. The natural thing to try is removing the noise. By far the most common technique used by image processing wizards to reduce the amount of noise present in an image is spatial averaging. The two algorithms most often used are the simple four and eight pixel replacements of Equations 1 and 2, where the pixels are labelled as in Figure 5-3.

NW	N	NE
W	X	E
SW	S	SE

Figure 5-3: Pixel Map for the Standard Spatial Averaging Algorithms

$$x = \frac{N + E + S + W}{4} \tag{1}$$

$$x = \frac{N + NE + E + SE + S + SW + W + NW}{8} \tag{2}$$

The action of the spatial filtering algorithms is easily interpreted in the context of Laplace's equation. Consider the intensity of an image as a function of the two spatial variables as a surface in three dimensional space. To reduce noise, what's needed is to minimize the curvature of the surface at every point. The best we can hope for is *zero* curvature, so we set some estimate of the curvature to zero. This is exactly what Laplace's equation does (Equation 3).

$$\frac{\partial^2 i}{\partial x^2} + \frac{\partial^2 i}{\partial y^2} = 0 \tag{3}$$

Equation 4 is one of the most grotesque yet still acceptable approximations to the second derivative available.

$$\frac{\partial^2 i}{\partial x^2}\Big|_{m,n} \approx \frac{i_{m+1,n} - 2i_{m,n} + i_{m-1,n}}{4} \tag{4}$$

Combination of Equations 3 and 4 yeilds Equation 1, the four pixel averaging scheme. The eight pixel scheme comes from taking into account the derivatives in the diagonal directions as well.

The principal drawback inherent in spatial averaging is the tendency to blur the image. Since the processed value of each pixel depends on the values of its neighbors as well as on its own, the energy in the image spreads out after each filtering pass. Both algorithms are actually low-pass filters, and may be analyzed

as such. In the four pixel case, the z-transform of Equation 1 yields Equation 5, where z_1 and z_2 are the z transform variables of m and n.

$$\begin{aligned} X &= \tfrac{1}{4}(z_1 + z_1^{-1} + z_2 + z_2^{-1})\, I(z_1,z_2) \\ &= \tfrac{1}{2}(\frac{z_1 + z_1^{-1}}{2} + \frac{z_2 + z_2^{-1}}{2})\, I(z_1,z_2) \end{aligned} \tag{5}$$

To find the frequency response, replace z by $e^{j\omega}$ to get Equation 6.

$$H(\omega_1,\omega_2) = \tfrac{1}{2}(\cos\omega_1 + \cos\omega_2) \tag{6}$$

By incorporating some "intelligence" into the filtering algorithm, it's possible to remove noise in certain areas of the image while leaving others untouched. For example, homogeneous areas of the image could be filtered without sacrificing edge character, an operation clearly needed when performing edge or line detection. This type of smart filter, called an adaptive spatial averaging, or ASA filter, is actually two filters: one which decides which areas of the image are to be filtered, and another which performs the filtering.

A small interpretive language to implement the idea of two pass filtering was written with the aid of the compiler writing tools on UNIX. Figure 5-4 gives a syntax summary of the language. A small set of utility commands is included to avoid returning to the support program every time the user wants to do something simple like clearing or updating the screen. A simple conditional statement and a library of filtering functions enable the processing engine to use one filter to select certain pixels for processing by a second filter, or to mark the selected pixels so the user can see what's going on. Currently implemented filters include the Sobel edge operator and several low and high pass convolution kernels. Lp8, for example, is an eight point neighborhood average.

```
command:        <simplecmd> or <filter> or <statement>
simplecmd:      read, show, clear, pause, sleep <n>, quit or ↑D
filter:         lp4, lp8, hp4, hp8, pixel or sobel
statement:      clip <op><n> or
                if <cond> then <action> or
                <variable> = <n>
cond:           <filter> <op> <n>  or  (cond)
op:             <, <=, >, >=, = or !=
action:         <filter> or mark <n>
n:              an integer
```

Figure 5-4: Syntax of the Adaptive Spatial Filtering Language

The usefulness of the language is certainly not limited to ASA operations, since the library of filters can

be easily expanded. It is our intention to extend the capabilities of the language in the near future. The following are two examples of input to the interpreter.

```
|
| Produce a binarized edge map of the image.
|
sobel                              | Run the Sobel edge operator.
read                               | Read the new image into memory.
if (pixel < 200) then mark 0       | Mark low edge-strength pixels black.
if (pixel >= 200) then mark 255    | Mark high edge-strength pixels white.
```

Alternatively, a program producing the same results with less computation since it only makes two passes over the window is given below.

```
|
| Produce a binarized edge map of the image (fast version).
|
if (sobel < 200) then mark 0       | Mark low edge-strength pixels black.
read                               | Read the new image into memory.
if (pixel > 0) then mark 255       | Mark high edge-strength pixels white.
```

The second example marks pixels with a high edge strength, pauses, updates the screen and then filters all the pixels with a low edge strength using a low pass filter. The result is that only the homogeneous or slightly shaded areas of the image undergo spatial averaging.

```
|
| Adaptive Spatial Averaging Example
|
if (sobel > 200) then mark 255     | Show which pixels will be filtered.
pause                              | Let the user look for a bit.
show                               | Put the old image back up.
if (sobel <= 200) then lp8         | Perform the ASA passes.
```

6. Conclusions

The POPEYE vision system described in this paper has been developed at CMU as an experimental tool for the study of visual inspection, object orientation, object classification, and interactive control tasks. The design goals of the system were to provide flexibility in the development of algorithms and systems concepts with reasonable speed of performance and moderate cost. The resulting hardware/software system now serves as a semi-portable stand-alone system which may conveniently be utilized in different laboratories for studies of specific applications. The POPEYE system provides an integrated gray-level vision system capability for the Flexible Assembly Laboratory and is used in conjunction with robotic manipulators, a binary vision system, tactile and force sensors for sensor-based control and assembly experiments.

The capabilities of the POPEYE system are evolving through the addition of custom boards. The multiple bus architecture offers useful alternatives for the design of boards with varying complexity and cost. As

specific strategies for recognition and interpretation of images for industrial applications evolve, we anticipate refined implementation of hardware and software mechanisms for these purposes. Recent applications of the system to industrial problems have included the characterization of a coating process using variance measures of local texture, inspection of glass integrity using edge-following techniques, the determination of object orientation for robot acquisition using piecewise gradient modeling and histogram modification methods, and the validation of assembly procedures using image subtraction to isolate component parts under manipulator control.

References

1. Agin, G. J., "Real time control of a robot with a mobile camera," *Proc. 9th Int'l. Symp. Industrial Robots*, S.M.E. and R.I.A., Washington, D.C., March 1979, pp. 233-246.

2. Binford, et al., " Exploratory Study of Computer Integrated Assembly Systems," Memorandum AIM-285.4, Artificial Intelligence Laboratory, 1977.

3. Birk, J., Kelley, R., Brownell, T., Chen, N., et al., "General methods to enable robots with vision to acquire, orient, and transport workpieces," Fifth Report, Department of Electrical Engineering, University of Rhode Island, August 1979.

4. Birk, J., Dessimoz, J., Kelley, R., Ray, R., et al., " General methods to enable robots with vision to acquire, orient, and transport workpieces," Seventh Report Grant DAR 78-27337, University of Rhode Island, December 1981.

5. Bolles, R., et al., " The Use of Sensory Feedback in a Programmable Assembly System," Tech. report 220, Stanford Artificial Intelligence Project Memo, Stanford university, October 1973.

6. Bolles, R., " Verification vision for programmable assembly," *Proc. 5th Int. Joint Conf. AI*, 1977, .

7. Chen, N-Y., Birk, J. R. and Kelley, R. B., " Estimating workpiece pose using the feature points method," *IEEE Trans. Aut. Control*, Vol. MC-25, December 1980, pp. 1027-1041.

8. Coleman, E.N. Jr., and Jain, R., " Shape from shading for surfaces with texture and specularity," *Proc. 7th International Joint Conference on Artificial Intelligence*, Vol. II, August 1982, pp. 652-657.

9. Ejiri, M., et al., " A Prototype Intelligent Robot That Assembles Objects from Plane Drawings," *IEEE Trans. Comp.*, Vol. C-21, 1972, pp. 161-170.

10. Horn, B., *Shape from shading: a method for obtaining the shape of a smooth opaque object from one view*, PhD dissertation, Department of Electrical Engineering, M.I.T., 1970.

11. Horn, B.Artificial Intelligence Lab., M.I.T., *Image intensity understanding*, 1975.

12. Horn, B., "Hill-shading and the reflectance map," *Proc. Image Understanding Workshop*, Palo Alto, California, April 1979, pp. 79-100.

13. Hunt, A. E., and Sanderson, A. C., "Vision-Based Predictive Robotic Tracking of a Moving Target," Tech. report, CMU Robotics Institute, 1982.

14. Ikeuchi, K., " Determining surface orientation of specular surfaces by using the photometric stereo method," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, Vol. 3, No. 6, November 1981, pp. 661-669.

15. Nitzan, D., et al., "Machine Intelligence Research Applied to Industrial Automation," Tech. report, SRI International, August 1979.

16. Perkins, W., " A model-based vision system for industrial parts," *IEEE Trans. Comput.*, Vol. C-27, Feb. 1978, pp. 126-143.

17. Rosen, C. A., "Machine vision and robotics: Industrial requirements," in *Computer Vision and Sensor-based Robots*, G. C. Dodd and L. Rossol, ed., Plenum, New York, 1979, pp. 3-22.

18. Sanderson, A. C., and Weiss, L. E., "Image-based Visual Servo Control of Robots," *Proceedings 26th SPIE International Symposium*, August 22-27 1982, .

19. Arthur C. Sanderson and Rafael Bracho, "Segmentation and Compression of Gray Level Images Using Piecewise Gradient Models," *Proceedings of 26th. SPIE Technical Symposium*, August 1982, .

20. Sanderson, A. C. and Weiss, L. E., " Adaptive Visual Servo Control of Robots," in *Robot Vision*, Alan Pugh, ed., I.F.S. Publications, Ltd., Bedford, England, 1982.

21. Shirai, Y. et al., " Visual Feedback of a Robot Assembly," Group Memo, ETL A.I.R., 1972.

22. Warnecke, H. J., et al., " An adaptable programmable assembly system using compliance and visual feedback," *Proc. 10th Int. Symp. on Industrial Robots and 5th Int. Conf. on Industrial Robot Technology*, Milan, Italy, March 5-7 1980, pp. 481-490.

23. Yachida, M. and Tsuji, S., " A versatile machine vision system for complex industrial parts," *IEEE Trans. Comput.*, Vol. C-26, Sept. 1977, pp. 882-894.

24. Duff, M.J.B., and Levialdi, S., *Languages and Architectures for Image Processing*, Academic Press, New York, 1981.

25. Reddy, D. R., and Hon, R. W., " Computer Architecture for Vision," in *Computer Vision and Sensor-based Robots*, Plenum Press, New York, 1980.

26. Duff, M.J.B., " Parallel processors for digital image processing," in *Advances in Digital Image Processing*, P. Stucki, ed., Plenum Press, New York, 1979, pp. 265-276.

27. Fountain, T. J., and Goetcherian, V., " CLIP4 parallel processing system," *IEEE Proc.*, Vol. 27, No. Pt. E, 1980, pp. 219-224.

28. Batcher, K. E., "Design of a massively parallel processor," *Trans. IEEE on Computers*, Vol. C-29, 1980, pp. 836-840.

29. Siegel, H. J., " PASM: a reconfigurable multimicroprocessor system for image processing," in *Languages and Architectures for Image Processing*, M.J.B. Duff and S. Levialdi, ed., Academic Press, New York, 1981.

30. Ballard, D. H., and Brown, C M., *Computer Vision*, Prentice Hall, Inc., Englewood Cliffs, New Jersey 07632, 1982.

31. Pratt, W. K., *Digital Image Processing*, Wiley-Interscience, New York, 1978.

32. Rosefeld, A., ed, *Image Modeling*, Academic Press, New York, 1981.

33. Tanimoto, S., and Klinger, A., eds., *Structured Computer Vision*, Academic Press, New York, 1980.

34. Sanderson, A.C., and Perry, G. R., "Sensor-based robotic assembly systems: Research and applications in electronics manufacturing," *IEEE Special Issue on Robotics*, November 1982, , In Review

35. Preston, K., " Image manipulative languages: a preliminary survey:," in *Pattern Recognition in Practice*, Kanal, L. N., and Gelsema, E. S., ed., North-Holland, Amsterdam, 1980.

36. K. Preston Jr., "Some Notes on Cellular Logic Operators," *IEEE trans. on Patt. Anal. and Mach. Intellig.*, Vol. PAMI-3, No. 4, July 1981, pp. 476-482.

37. *Computer Architecture for Pattern Analysis and Image Database Management*, IEEE Computer Society, IEEE Computer Society Press. Hot Springs, Va., 1981.

38. K. Preston Jr., M. J. B. Duff, S. Levialdi, P. E. Norgren and J-i Toriwaki, "Basics of Cellular Logic with Some Applications in Medical Image Processing," *Proceedings of the IEEE*, Vol. 67, No. 5, May 1979, pp. 826-856.

39. Horn, Berthold, "Focusing," MIT Project MAC, A. I. Memo 160, Massachussets Institute of Technology, May 1968.

40. Tenenbaum, Jay M., *Accommodation in Computer Vision*, PhD dissertation, Stanford University, November 1970.