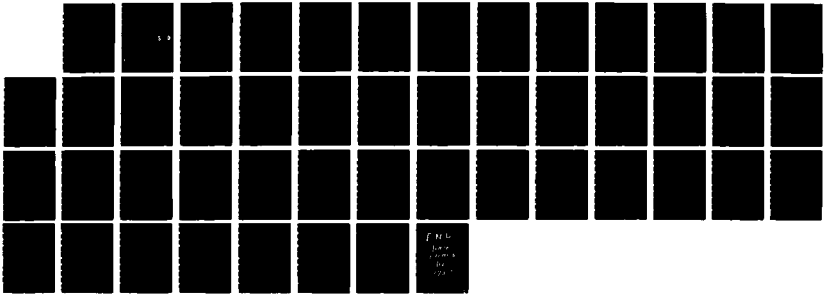
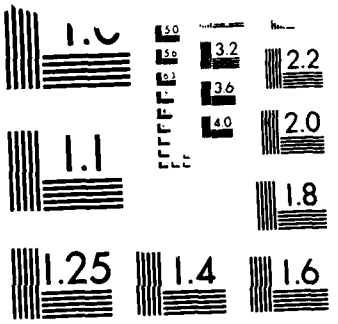
AD-A185 560 VIRTUAL IMAGE PROCESSOR: A PROTOTYPE IMPLEMENTATION(U) 1/1
AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH
R A KAUCIC 1987 AFIT/CI/NR-87-98T

UNCLASSIFIED F/G 12/6 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD-A185 560

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/CI/NR 87-98T	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Virtual Image Processor: A Prototype Implementation		5. TYPE OF REPORT & PERIOD COVERED THESIS/DISSERTATION 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Robert A. Kaucic, Jr.		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFIT STUDENT AT: University of Washington		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS AFIT/NR WPAFB OH 45433-6583		12. REPORT DATE 1987 13. NUMBER OF PAGES 39
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

DTIC ELECTE S NOV 0 4 1987 D

aD

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

APPROVED FOR PUBLIC RELEASE: IAW AFR 190-1

LYNN E. WOLAVER 23 Oct 87
Dean for Research and
Professional Development
AFIT/NR

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ATTACHED

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DTIC FILE COPY

University of Washington

Abstract

Virtual Image Processor: A Prototype Implementation

by Robert A. Kaucic, Jr.

Chairman of the Supervisory Committee: Dr. Yongmin Kim

Department of Electrical Engineering

A prototype virtual image processor has been developed for low-cost image processing applications, and has been implemented on three separate image processors using an IBM PC/AT as a host. Application programs make device-independent calls to the virtual image processor which is currently implemented as sets of library calls and linked into the application programs at compilation time, which facilitates the porting of application software among different image processors. The virtual image processor can be used as an aid in the development of portable image processing software or as an educational tool for students new to the field of image processing.

Virtual Image Processor: A Prototype Implementation

By

Robert August Kaucic, Jr.

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science in Electrical Engineering

University of Washington

1987

Approved by: _____
(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree: _____

Date: _____

In presenting this thesis in partial fulfillment of the requirements for a Master's Degree at the University of Washington, I agree that the library shall make copies freely available for inspection. I further agree that extensive copying of this thesis is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Any other reproduction for any purpose or any means shall not be allowed without my written permission.

Signature: _____

Date: _____

University of Washington

Abstract

Virtual Image Processor: A Prototype Implementation

by Robert A. Kaucic, Jr.

Chairman of the Supervisory Committee: Dr. Yongmin Kim

Department of Electrical Engineering

A prototype virtual image processor has been developed for low-cost image processing applications, and has been implemented on three separate image processors using an IBM PC/AT as a host. Application programs make device-independent calls to the virtual image processor which is currently implemented as sets of library calls and linked into the application programs at compilation time, which facilitates the porting of application software among different image processors. The virtual image processor can be used as an aid in the development of portable image processing software or as an educational tool for students new to the field of image processing.

TABLE OF CONTENTS

LIST OF FIGURES

Introduction

A significant problem in computer engineering today is the inability of application programmers to keep pace with the rapid technological advances being made in computer hardware. The field of image processing is an example of this phenomenon, in which new hardware developments are slow to be incorporated into existing software packages. A major contributing factor to this problem is the application software that has been written is poorly structured, heavily laden with device dependencies, and unportable to new hardware.

Historically, image processing software packages have been created in an ad hoc fashion, that is, as a need developed, an application would be written to meet that need. As the number of image processing programs grew, similar applications were grouped together and called a "system." While such practices are natural and commonplace, they suffer the deficiency of leading to poorly structured software packages that are difficult to modify and extremely device/system-dependent. Even though this ad hoc approach is very effective in terms of meeting deadlines and satisfying immediate research needs, the resultant software is often virtually unportable to new hardware. Upgrades to new hardware normally require a complete rewrite of the software, or at the very minimum, extensive, very costly and

time-consuming revisions. This implies that users remain bound to older outdated hardware for long periods of time, even though newer, more powerful hardware may have already been purchased and is operable in house.

A better alternative to this ad hoc approach to software development would be to use a consistent and well-structured, layered approach. In this method, the software would be partitioned into layers, as shown in Fig. 1, where each layer communicates to its nearest neighbors via a predefined protocol. Software written in this manner would remove many of the device dependencies found in current image processing software packages, as each layer has no knowledge of, nor has any need of knowing how things are done beneath it. This layered software approach permits programmers to continue to add applications to their system, as the need for them arises, as long as they adhere rigidly to the layered structure and the communications protocol between layers.

Unfortunately, such practices have been the exception rather than the rule. Furthermore, it would seem inevitable that the requirements for each layer and the communication protocol between layers would certainly differ from organization to organization, and even from system to system within the same organization. In addition, experience with other devices outside of image processing, strongly indicates that there should exist a set of primitive image

processing functions that all image processors should be able to perform, and that the definition and use of this standard significantly enhances software portability. It would thus be logical to group this set of functions into a "virtual image processor," where a virtual image processor is merely an abstract representation of a device along with the functions it performs and a language with which to communicate.

Assuming that a standard virtual image processor (VIP) could be implemented, application programs could then make virtual calls to this device using the predefined VIP language. A VIP would force application programmers to write device-independent code, and would thus greatly enhance the portability of image processing software packages. Since the actual implementation of the VIP would be completely transparent to the application programs, the same application program could run on several different image processors merely by switching the hardware on which the VIP is implemented. Powerful image processors would require very little software to implement the VIP, while less powerful image processors would have to emulate many of its functions, but a standard VIP interface would trivialize upgrades to more powerful hardware as the new hardware would replace much of the software emulation. Finally, a VIP would also facilitate the implementation of an image

processing network where several different image processors and workstations could be configured as a network and used in concert.

The need for virtual image processor development has been recognized in the image processing user community and by image processing equipment manufactures as well. The National Aviation and Space Administration (NASA), one of the pioneers in image processing software development and heavily dependent on such software, and the Gould Inc., a major distributor of image processing equipment, are two such organizations that recognize the problems of device and system-dependent software packages.

Gould has developed an Imaging Kernal System (IKS) which introduces a formalized model of a Virtual Image Processing System (VIPS) in addition to virtual image processing objects that include images, overlays, control transforms, windows, and programmable cursors [1]. Gould's primary concern in developing IKS was to maximize the portability of applications software (internally as well as externally) across the Gould image processing product line. IKS was not designed to be specific to the Gould product line, but it is currently proprietary software and has only been implemented on Gould machines. Furthermore, only a FORTRAN binding presently exists running under the VMS operating system.

NASA also has spent many man years in an attempt at

providing coherent, structured software to their community of users by developing their own version of a virtual image processor [2]. NASA's system called DMS (Display Management Subsystem) along with its virtual operating system TAE (Transportable Applications Executive) [3] address problems related to device-dependent code as well as operating system dependent code. TAE provides a system programming library that performs disk and file management and other related operating system dependent features, which allows programmers to write operating system independent software. Alternately, DMS provides a virtual image processor interface which enables application programmers to write device-independent code. TAE has been implemented on various systems, and DMS on several different image processors, but the problem with utilizing NASA's software is that they assume as a basis, a large, powerful system which is usually beyond the capabilities of typical low-cost systems. Thus, as an alternative (albeit, less powerful alternative), this paper will discuss a prototype virtual image processor for low-cost image processors.

Image Processors

A digital image processor can be defined as a machine that aids a computer in manipulating images. The word, image, can refer to many things, but for this paper, it will be restricted to a two-dimensional numerical representation of an object. A minimum image processor must therefore provide a means for moving images to and from a computer as well as a storage medium for the images once they arrive. A structure for a typical processor is shown in Fig. 2.

The host interface enables communication between the host computer and the image processor. The image memory, or memories, often called frame buffers, are used to store the images once they are received from the host computer or the

labeling the stomach in an image of the human digestive system without destroying the original image, or as a storage area for overflow conditions that arise when performing image arithmetic operations such as image averaging. Finally, most image processors have special purpose hardware that aids the host in manipulating the images. One such operation is the zooming of an image, where each element in a particular portion of the image is replicated several times to enable a more in-depth examination of the selected area. Other operations are panning and scrolling, which permit examination of the displayed image throughout its horizontal and vertical extent, particularly when the image has been zoomed. In addition, sometimes there is hardware cursor support, where a cursor is an illuminated pattern on the display monitor similar to a cursor on a typical computer terminal screen. Finally, more powerful image processors may provide hardware support for high-level image processing functions such as computing the histogram of an image, where a histogram is a one-dimensional table containing the number of pixels at each digital light intensity, or gray level.

Virtual Devices

Humans naturally view machines in an abstract manner defined only by the functions they perform and the language they require for communication, rather than in terms of voltage levels, devices being on or off, registers being clocked or cleared, and the like. It was necessary for the early programmers to communicate with computers in a very primitive "machine language" using jumpers, switches, and buttons. This of course was done out of necessity as they had no other means of communicating with the computers. After the early programmers became more familiar with computers, they were able to abstract the machine and communicate with it in a language known today as machine language, where instead of worrying about jumpers and switches, the provrammer concerned himself with setting "bits" in registers and moving "bytes/words" to and from memory locations. Viewing the interaction with the machine in this manner reflected much more closely what was actually happening at the macro level than stating what the voltage level at point A or point B should have been.

Machine languages were a step in the abstraction of the machine away from its physical construction, but the programmer was still forced to communicate in a cumbersome language of ones and zeros. Thus, the next step was the advent of assembly languages, which permitted the programmer

to communicate using mnemonic instructions that were much more natural and easier to understand. The programmer was still required to learn a different language for each computer, but his model of the machine had grown from one of a physical piece of hardware to one that could move "data" from one point to another and even output "information" via peripherals.

A major breakthrough came with the advent of higher-level languages. While early high-level languages such as FORTRAN and BASIC closely resembled assembly languages, they were nevertheless the first attempt at abstracting the basic capabilities of the machine. With high-level languages, the emphasis had changed from how the machine did things to what exactly the machine could do. As a result of this higher level of abstraction, the programmer no longer needed to worry about the physical hardware with which he was communicating; rather, he could replace his model of the machine with the capabilities afforded by the higher-level language he chose to use.

With this new virtual model of the machine, the programmer was able to write code for this "virtual machine," (which is defined by the language chosen to interact with it), which could be directly portable across "real machines" provided the implementation of the language was consistent across machines. Thus, each high-level language, in a sense, defined its own virtual machine.

Similiarly, other tools such as operating systems can also be viewed as virtual machines, as they too have been written to extract the basic capabilities of the machine and define an interface to it. Thus, we see, as humans, that we naturally abstract our view of physical devices to virtual machine models, and hence a virtual model for an image processor is merely an extension of this concept.

Virtual Image Processor Model

The logical first step in defining a virtual image processor is, of course, determining the functions that all image processors should be able to perform. At present, there does not exist a set of standard image processing functions that should be supported as is the case with graphics commands [4]. However, certain minimum capabilities can be specified, and as a starting point for the development of a prototype virtual image processor, we have attempted to determine these capabilities.

First and foremost, an image processor should support initialization which prepares the device for further interaction. This includes the initialization of device registers, lookup tables, refresh memories (frame buffers), overlay planes, and cursors. It should also enable the manipulation of images to include writing to and reading from a host computer as well as zooming, panning and scrolling. It should provide cursor support that allows the definition of different shapes and sizes, allows the cursor to be illuminated or removed from the display (often referred to as turning the cursor on and off), supports various cursor colors, and permits movement of the cursor. In addition, it should provide access to one or more graphics overlay buffers that permit the turning on and off of bit planes, the drawing of lines and circles, the zooming

Virtual Image Processor Model

The logical first step in defining a virtual image processor is, of course, determining the functions that all image processors should be able to perform. At present, there does not exist a set of standard image processing functions that should be supported as is the case with graphics commands [4]. However, certain minimum capabilities can be specified, and as a starting point for the development of a prototype virtual image processor, we have attempted to determine these capabilities.

First and foremost, an image processor should support initialization which prepares the device for further interaction. This includes the initialization of devhce registers, lookup tables, refresh memories (frame buffers), overlay planes, and cursors. It should also enable the manipulation of images to include writing to and reading from a host computer as well as zooming, panning and scrolling. It should provide cursor support that allows the definition of different shapes and sizes, allows the cursor to be illuminated or removed from the display (often referred to as turning the cursor on and off), supports various cursor colors, and permits movement of the cursor. In addition, it should provide access to one or more graphics overlay buffers that permit the turning on and off of bit planes, the drawing of lines and circles, the zooming

of the overlay buffer, and the selection of portions of the image for region of interest operations. Finally, it must permit writing to and reading from lookup tables to allow histogram equalization, point operations, and psuedo and real color image display.

In light of the functional requirements as stated above, the prototype virtual image processor has been divided into five sections. They are initialization, image manipulation, cursor support, graphics, and lookup table manipulation.

A. Initialization

This package contains those routines required to initialize the image processor. It consists primarily of three routines: ip_open(), ip_close(), and ip_init(). Ip_open(), which is similar to the standard C open() library call, merely establishes the connection between the physical device and its software representation. Ip_close() is similar to C's close() command as it severs this connection.

Ip_init() puts the image processor in a known state, initializes the lookup tables to a linear mapping, and clears the display and graphics overlay planes when specified.

B. Image Manipulation

Image manipulation includes routines that perform

operations on the displayed image as well as any additional frame buffers in the image processor. It consists of several routines: zoom(), clear(), get_pixel(), set_pixel(), vert_line(), horiz_line, get_block(), put_block(), add_image(), sub_image(), mult_image(), div_image(), and ave_image().

Zoom() allows vertical and horizontal zoom of the displayed image. Further, it pans and scrolls the displayed image when zoomed to utilize the entire display. Clear() initializes the selected frame buffer to all zeros, while get_pixel() and set_pixel() are simple routines that allow random access to any pixel in the display buffer. Vert_line() and horiz_line() are used to write a one-dimensional array of data into a frame buffer. They have been included to take advantage of image processors that are optimized for display in a particular direction.

Get_block() reads a rectangular window from the selected frame buffer and returns it to the host. Put_block() writes a rectangular window to the selected frame buffer. These routines are sufficiently powerful to take advantage of block move instructions that are available on some low-cost image processors, yet simple enough that emulation on less powerful image processors is rather straightforward.

Add_image() performs pixel by pixel additions of two

images. Sub_image(), mult_image(), and div_image() are similiar in that they also operate on a pixel by pixel basis. Ave_image(), which averages two or more images, is primarily used to improve the signal-to-noise ratio in images.

C. Cursor Support

This package contains routines that set up the cursor at a given location, move the cursor on the image processor, and change the cursor size, shape and color. These routines are init_cursor(), on_cursor(), off_cursor(), move_cursor(), where_cursor(), change_curs_color(), change_curs_size(), and change_curs_shape().

Init_cursor() initializes the cursor to the default location (the center of the display), and the default size, shape, and color, which are specific to each image processor. It then turns the cursor on. On_cursor() illuminates the cursor for display, while off_cursor() removes the cursor from the display. Move_cursor() moves the displayed cursor to the location specified, while where_cursor() returns the current location of the cursor. Change_curs_color() changes the current color of the displayed cursor. Change_curs_size() increments the size of the cursor, and change_curs_shape() steps through the various cursor shapes available on the image processor.

D. Graphics

This contains functions that allow for the annotation of images, to include text annotation, the drawing of lines and circles, the turning on and off of bit planes, and the selection of regions for ROI (region of interest) operations. It consists of several routines: write_text(), graph_vert_line(), graph_horiz_line(), graph_put_block(), graph_erase_block(), ip_vect(), ip_circle(), on_display_color(), off_display_color(), and erase_display_color().

Write_text() writes a scaled string of text in the specified color at the specified location. Graph_vert_line() and graph_horiz_line() are used to write a one-dimensional array of a specified color into the graphics overlay area.

Graph_put_block() writes a rectangular buffer of the specified color, while graph_erase_block() clears a rectangular region in the graphics overlay buffer to all zeros. They are useful for creating logos, updating graphical plots, and defining one-of-a-kind fonts. For example, one could specify his organization's logo as a rectangular grid containing the appropriate color at the appropriate location. Graph_put_block() could then be used to write the logo into the graphics overlay.

Ip_vect() and ip_circle() draw vector lines and circles respectively in the specified color. On_display_color()

turns on a given color for display, while off_display_color() turns off the display of the specified color. These functions are useful for distinguishing between different overlays, markers, regions of interest, and others on the display. Erase_display_color() is used to remove an entire color from the graphics overlay buffer. This is useful in that it allows users to initialize the graphics overlay buffers, correct mistakes, or even reuse the same color if so desired.

E. Lookup Table Manipulation

This allows the application programs to initialize the lookup tables or to modify them to reflect the effects of histogram equalization, point operations, or to display pseudo or full color images. There are only three routines required: init_LUT(), load_LUT(), and read_LUT().

Init_LUT() initializes the lookup tables (LUTs) to a linear mapping. Load_LUT() lets the host load a selected LUT with arbitrary values, while read_LUT() returns a buffer containing the desired LUT to the host. These functions permit psuedo and full color display, facilitate point operations, and enable lookup tables with certain desirable features to be saved for future use, as the application program need only read the desired lookup table and then write it to disk for permanent storage.

Physical Implementation

In order to test the generality and usefulness of the virtual image processor (VIP), it has been implemented on three separate hardware devices. The IBM PC/AT running the Santa Cruz Operation (SCO) Operating System V XENIX was chosen as the host computer for these three image processors. The PC/AT was chosen primarily because of its relatively low cost (as compared to more powerful minicomputers) and its wide acceptance as a de facto personal computer standard. Also, due to the AT's popularity, it was possible to find three separate image processors that could fit into its chassis, requiring no more than two expansion slots.

XENIX, a dialect of the UNIX operating system for microcomputers, was chosen because UNIX is the first operating system to be widely available across computer vendors from the microcomputer to the mainframe. The selection of a portable operating system like UNIX was a step in the creation of a truly virtual image processor, as operating system dependencies could limit the portability of the virtual image processor to the chosen operating system. Ideally one would also like to have a "virtual operating system" to free the VIP from any bias towards one particular operating system, similar to TAE developed by NASA [3], but for the initial prototype VIP, XENIX was certainly an

acceptable compromise.

A. Hardware Devices

The three image processors chosen were: an IBM Personal Computer Professional Graphics Controller Card (sometimes referred to as a PGC -- Professional Graphics Controller) [5]; an Imaging Technologies Incorporated FG-100-AT image processor [6]; and a custom two-board image processor built at the University of Washington based on Texas Instruments' 34010 Graphical Signal Processor (GSP) called the UWGSP [7].

1. Professional Graphics Controller (PGC)

The PGC is the least powerful of the three image processors. It contains a 640 x 480 x 8 bit frame buffer and three 256 x 4 bit lookup tables (LUTs). Besides the frame buffer and the LUTs, its remaining feature is an Intel 8088 microprocessor with 64 kbytes of ROM which controls the PGC and supports many high-level graphics commands including three-dimensional drawing, modeling transforms, pattern filling, and viewport rotation.

Given these limited capabilities, it is apparent that the PGC has many drawbacks. For starters, the frame buffer and the LUTs are internal to the PGC and cannot be mapped into the host's address space. Hence, all communication between the host and the PGC is accomplished through three 256 x 8 bit FIFO buffers, one for receiving image data from the PGC, one for writing image data to the PGC, and one for

receiving error and warning codes from the PGC. This FIFO communication protocol along with the small size of these FIFOs is a major bottleneck in the PGC as the transfer of a typical 512 x 480 x 8 bit image to and from the PGC takes on the order of 10 sec. Also, the crude resolution of the LUTs often introduces contouring into black and white images. In addition, since the PGC has only a single 8-bit frame buffer with no provision for graphics overlay planes, all annotation, including the cursor, must be emulated, which requires overwriting the image. Finally, because the PGC is a board developed primarily for computer graphics' applications it does not support in hardware such basic functions as zoom, pan, or scroll.

2. Imaging Technology Incorporated FG-100-AT (ITI)

The ITI provides four 512 x 512 x 12 bit frame buffers which give one the option of using all 12 bits for extended resolution, or alternately, 8 bits for resolution, and 4 bits for graphic overlay information. These extra frame buffers can also facilitate some image processing operations, e.g., convolution and FFT computation, as they can be used locally to compute intermediate results without affecting the original image. Three 4096 x 8 bit LUTs are available for psuedo color or even true color display. In addition, all four frame buffers and the three LUTs can be mapped into the host's memory address space enabling fast

transfer of data to and from the ITI.

The ITI also has 16 I/O channel-mapped registers that provide hardware support for zoom, pan, and scroll, and rapid clearing of frame buffers. The real-time digitization circuitry on the ITI provides a means for rapid acquisition of images. A feedback path from the frame buffers to a fourth lookup table, as well as a regular path from the digitizer, enables real-time addition, subtraction, multiplication, or division operations to be performed on 6-bit images as they are acquired using this 12-bit feedback lookup table.

Despite these features, the ITI lacks one major component, a processor, which could provide rapid text annotation and the drawing of lines, circles, and other geometric objects.

3. University of Washington Graphics System Processor

The UWGSP is the most powerful of the three image processors. Its most salient feature is that it utilizes an on-board TMS 34010 Graphics System Processor (GSP) and a TMS 32020 Digital Signal Processor (DSP) with 320 kbytes of program memory to support many image processing functions, including two-dimensional drawing functions, alphanumeric annotation in several character fonts, block read and writes, and high-speed cursor support. It also contains four 512 x 512 x 8 bit frame buffers, a 512 x 512 x 4 bit

graphics overlay buffer, and three 4096 x 8 bit lookup tables. Independent horizontal and vertical zoom is also supported in hardware.

B. Implementation

The actual implementation of the VIP, as shown in Fig. 1, requires device drivers to enable communication with the hardware, vendor software which builds on the device drivers providing basic low-level capabilities, and finally the VIP interface software. In general, vendors of image processing boards usually supply the device drivers and the vendor software with the image processing boards themselves, reducing the task of VIP implementation down to the writing of the VIP interface software. However, this was not the case for these three image processors as the vendor-supplied software was written to run under the MS-DOS operating system, which is of no help in a XENIX environment. What follows is a description of the software written to implement several functions of the VIP on each of the image processors demonstrating how the salient features of an image processor can be used to make the implementation of certain functions trivial, while lack of these features often requires extensive software emulation.

One such function is the zoom operation, which inherently requires the panning and scrolling of the image

to center the display on the point about which the zoom is being performed. The ITI board supports zooming, panning, and scrolling in hardware. Thus, zooming the displayed image is performed by writing the appropriate values to the ITI's zoom, pan, and scroll registers. Zoom, pan, and scroll are supported in firmware on the UWGSP, that is, the UWGSP provides hardware zoom, but only about the upper-left corner of the display. To permit near real-time zooming of images, there exists a subroutine in the UWGSP program memory that moves the region to be zoomed into the upper-left corner of the display buffer and then loads the x and y zoom registers with the appropriate zoom factors. Since this is done within the UWGSP, and hence transparent to the host, the host needs only to send a command to the UWGSP instructing it to zoom about a particular point. For the PGC, it is possible to zoom an image, but it is very slow and requires a large amount of memory. A copy of the image being displayed on the PGC is kept in a separate frame buffer on the host. This extra frame buffer is necessary, because the display buffer inside the PGC cannot be accessed directly by the host because of the FIFO communication protocol used by the PGC, and hence it is loaded whenever an image is sent to the PGC for display. If this were not done, the entire image would have to be read in from the PGC in 256 byte blocks every time a zoom was to be performed and would require around 20 sec to zoom an image. Currently, a

zoom is performed by copying the selected region from the extra frame buffer into a temporary frame buffer and replicating each pixel by the desired x and y zoom factors. This temporary frame buffer is then sent to the PGC which can only accept 256 bytes at a time.

The implementation of the cursor package also serves to highlight the differences among the image processors. The implementation on the UWGSP was very straightforward, as it provides firmware cursor support. To implement each of the functions in the cursor package, the host called the appropriate routine within the UWGSP. Implementation of the cursor package on the ITI board required a fair amount of software, but not nearly as much as was required by the PGC. For example, in the ITI, the most significant bit plane in its 4-bit graphics overlay buffer was used to indicate the presence or absence of the cursor. After the default pattern for the cursor was created, the cursor was illuminated for display (turned on) by lighting the appropriate locations in the cursor's bit plane corresponding to the cursor pattern. The cursor was turned off (removed from the display) by erasing these previously illuminated locations from the cursor's bit plane. The PGC, on the other hand, does not have a graphics overlay buffer, and hence the only way to provide cursor support was to overwrite the image with the cursor pattern at the desired

cursor location. To prevent permanent damage to the image, the overwritten area had to be temporarily saved and then written back into the image, when the cursor was turned off. To move the cursor, the overwritten area was written back into the image, a new region was read from the image corresponding to the new cursor position, and then the cursor pattern was written into the image. This is very slow as the PGC does not support block reads or writes, and transfer of a small number of pixels requires substantial overhead.

Changing the cursor color was also very straightforward with the ITI board, and awkward with the PGC. The 4096 x 8 bit lookup tables in the ITI were divided into sixteen 256 x 8 bit lookup tables. The 4-bit graphics overlay buffer was used to select among these sixteen lookup tables. Changing the cursor color was thus reduced to writing the new color into the LUTs corresponding to the cursor bit plane. A slightly different approach was used in the PGC. Since the implementation of the cursor on the PGC required that it be written into the image, changing the cursor color entailed writing the desired color into the lookup table entry corresponding to the pixel intensity assigned to the cursor. Here is where the problem arose, as each pixel in the image having its intensity equal to the value chosen for the cursor would change color when the cursor color changed. To guard against this possibility, a histogram of the displayed

image is computed as the image is being displayed, to ensure that the cursor is assigned a unique pixel intensity. For those rare cases where the image contains pixels at all possible gray levels, the PGC returns an error stating that it can not support color graphics, and hence the cursor color defaults to white.

These examples illustrate some of the wide range of capabilities among low-cost image processors and the vastly different methods to perform the same functions on different devices. This performance variation among different image processors is precisely what leads to poorly written, very device-dependent code, as the system programmers often develop a mind-set based on their experiences with a particular image processor. It is difficult for programmers to imagine how their code would run on different image processors with different capabilities and completely different communication protocols.

One way to combat the problem of device-dependent programming is to isolate the features that are specific to the given image processor into one particular area of the overall system, and then build device-independent software on top of it. A definition of a virtual image processor attempts to isolate these features, but it is still necessary to build on this VIP to demonstrate its utility. What follows is a description of several application

programs that have been built on top of this prototype virtual image processor.

Application Software

The application software chosen to build on the VIP was the Quick and Dirty Image Processing System (QDIPS) acquired from the Computer Systems Laboratory at the University of California at Santa Barbara. QDIPS was originally developed to aid researchers in the analysis of satellite images of mountainous regions, where their primary goal was to develop a means of accurately predicting run-off from snow melt. As the researchers' work broadened into other areas, QDIPS grew to encompass these applications as well. At present, the use of QDIPS has expanded to many areas of research, including satellite oceanography, digital terrain mapping, marine resource management, and data compression.

There are several features of QDIPS that are of particular interest to us. The most useful feature of QDIPS, in terms of familiarizing new users with the system, is its on-line help menu. With this on-line help menu, users can query the system for information on all of the commands, receive a list of commands based on a keyword, and receive information on many of the low-level commands available for use. Another desirable feature of QDIPS is that it was written in the C program language running under the Berkley UNIX 4.2 Operating System and an attempt was made to facilitate the porting of the software to other dialects of UNIX by isolating features unique to the

particular dialect. QDIPS also provides many useful utilities for use by application programmers. For example, QDIPS has incorporated within it a debugging facility that aids programmers in creating new applications or porting existing applications. These debugging routines have been implemented using the C macro preprocessor to enable their inclusion or exclusion based on a compilation flag. QDIPS also has extensive error handling capabilities, where encountered errors generate an error message suggesting the probable cause for the error along with the program name and line number within the program where the error occurred. The error handling capabities also provide warning messages when appropriate and a trace of the path leading to the error if requested. There are also functions that enable the rapid parsing of command line arguments. In addition, a crude form of protection has been incorporated into QDIPS that denies access to unauthorized users.

One additional feature of QDIPS is that it does not run under a burdensome ehecutive which requires the selection of menus with an interactive device such as a mouse or trackball, rather each function can be called directly by typing in the appropriate command from the keyboard. This also facilitates the creation of new application programs, as each application program can be a stand-alone process totally separate from the other applications. In addition, each function has its own help facility which gives a

detailed explanation of the command and the parameters it requires. This help facility is activated by typing the command name followed by the single argument, help.

Three software libraries were built using the virtual image processor as a foundation, in addition to the libraries that already existed, which provide useful utilities to the application programs. These three libraries are a keyboard package that allows one to move around the image and emulate buttons being pushed on interactive devices, a cursor and color graphics overlay package that provides a simple interface to application programs needing to access the cursor and graphics overlay capabilities of the virtual image processor, and an interactive plotting package which allows the image processor to appear as a plotting device to the outside world.

The keyboard package can work in conjunction with interactive devices to enable rapid cursor movement throughout the image with, for example, a mouse, while still permitting precise cursor movements using the keyboard. This could be very useful in selecting a region of interest on which an operation is to be performed, such as the selection and measurement of the densitometric and morphometric properties of cell nuclei. It also allows the execution of an image processing operation such as zooming

or annotating an image, changing the cursor color, or computing a histogram, with the single press of a button on an interactive device or a single key on the keyboard. Currently, only the Microsoft parallel mouse has been integrated into the keyboard package, but other interactive devices can be easily added.

The cursor and color graphics overlay package provides a high-level interface to routines needing to manipulate the cursor or examine the various graphics planes. To use the cursor and color graphics overlay package, the application program needs only to specify a sequence of colors that it wishes to display when changing the cursor color and/or viewing the graphics planes. For example, by using the cursor and color graphics overlay package in conjunction with the keyboard package, the user can define the cursor color sequence to be red, green, yellow, magenta, blue, cyan, and white. He can then trace a nucleus in an image in red, press a button, label the nucleus as such in green, press a button, trace the next necleus in yellow, press a button, label it in magenta, and so on. The user can then view all of this annotation at once, one color at a time, or select a set of colors to be displayed.

The interactive plotting package, when used in conjunction with the keyboard package, allows the user to rapidly annotate images with lines, circles, ellipses, etc. This could be used in determining the circularity of an

object by first encircling it and then performing circularity measurements on it.

Several functions have been built on top of these library packages to demonstrate some of the capabilities of the system as a whole, in addition to showing how easily application software can be ported from one image processor to another, when a model of a virtual image processor has been properly defined and utilized. These functions include initializing the "real" image processor, displaying images on the image processor, annotating images with user-defined fonts, computing histograms of images, performing linear contrast enhancement and histogram equalization on images, loading predefined lookup tables into the image processor to permit displays in both pseudo and full color, drawing gray scales, adding or subtracting two images, and averaging two or more images. In addition, the cursor and color graphics package, the keyboard package, and the interactive plotting package have been incorporated into each of these functions to enable color annotation and permit rapid cursor movement throughout the image.

Discussion

The system, at present, does not have sufficient capabilities to serve as a legitimate image processing workstation, primarily because the application software was implemented to help shape the model of the virtual image processor and to demonstrate the increase in software portability that naturally follows the definition and use of a virtual image processor, rather than to create a full-blown system. As is, the combination of the virtual image processor and the QDIPS application software can serve as a useful educational tool for newcomers to the field of image processing, as well as a test bed for the development of new application software or a full-scale virtual image processor.

Beginners in image processing should not have to concern themselves with the low-level details required to interface to different image processors. They should be given a set of functions to use, and instructed on how to interface to these routines. This enables them to focus on how these tools can be used to accomplish a given task rather than how these tools are actually implemented. A model of a virtual image processor is ideally suited for this purpose, as the student can make simple calls to general image processing functions as opposed to learning one particular system with its own set of peculiarities.

Furthermore, the utilities provided by QDIPS make it very easy for students to implement and debug new application software -- a must for students who are also most likely experimenting with the C programming language for the first time. In addition, since the virtual image processor prototype was written for low-cost systems, educational institutions do not need to spend an exorbitant sum of money, typical of many image processing systems, to acquire workstations for student use.

The system can also be used to develop new application software, because the very nature of the virtual image processor prevents application programmers from incorporating device-dependencies into their software. In addition, as new applications are added, it is inevitable that the programmers will see a need to expand the capabilities of the virtual image processor to meet their needs. Adding to the virtual machine will then serve to increase their knowledge of virtual software development, which will in turn facilitate the development of additional application programs.

There are many improvements that could be made to the virtual image processor model. The first task would be to remove the XENIX operating system dependencies. Although UNIX and its dialects are among the most portable operating systems, defining a virtual operating system and making calls to it would provide many of the advantages that were

gained by the creation and use of a virtual image processor. Operating systems dependencies are not as severe as device dependencies, but they are nevertheless a barrier to system portability. Another improvement would require the application software to query the device being used to determine its capabilities and make decisions based on the capabilities of the device.

The virtual image processor is currently implemented as a set of library calls. Thus the entire virtual image processor is linked into the application programs at compilation time. This often results in very large executable files. A better approach would be to have the virtual image processor run as a separate process and have the application software send metacode commands to the virtual image processor which would then interpret the commands and perform the desired operation. This could substantially reduce the amount of memory required by application programs, would not bind the application software to one particular device until run time, and would permit dynamic selection of devices as the application program would need only to break communication with the first device and start up communication with a second. Work in this area is currently under investigation in our laboratory [8].

Finally, the definition of a standard interface is

usually a very iterative process, as was the case with this prototype virtual image processor, and the best way to demonstrate its utility and generality is to continue to add capabilities and port additional application programs to it.

Conclusion

A prototype low-cost virtual image processor has been built and implemented on three separate image processors. Application software has been successfully ported to these image processors using the model of the virtual image processor. The application software has helped to shape the definition of the virtual image processor, and demonstrate the inherent increase in the portability of application software written to a virtual image processor, as opposed to software written for a particular device which is usually plagued with device dependencies. The model can also be used as an educational tool representing many features currently present in low-cost image processors.

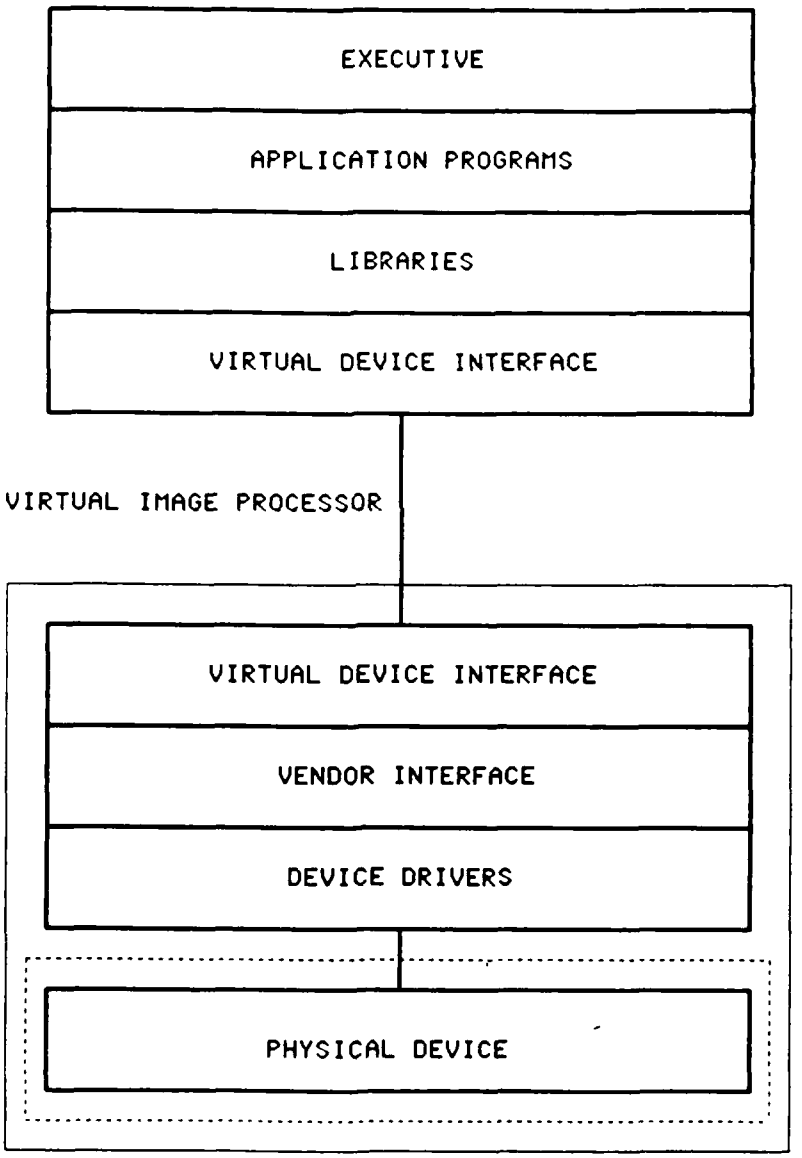


Figure 1. Image Processing Layered Software

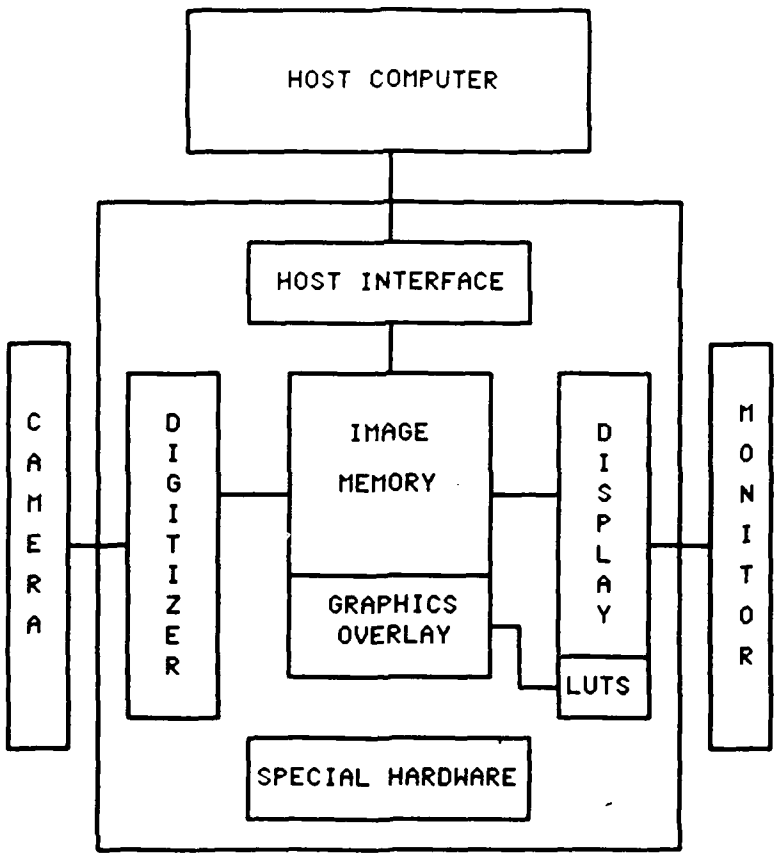


Figure 2. Image Processor

REFERENCES

[1] Dierker, M., Pilkenton, D., Levey, L., Reese, A. and Wilcox, J. A., "The Imaging Kernel System (IKS): A device independent software interface to image processing systems, Electronic Imaging '86, pp. 126-132, 1986.

[2] "Functional description for the Display Management Subsystem (DMS), version 1.0," NASA Internal Document Number: 86-DMS-DSCV1, TAE Support Office, Code 636, NASA/GSFC, Greenbelt MD, July 1986.

[3] "System Manager's Guide for the Transportable Application Executive (TAE), version 1.3" NASA Internal Document Number: 84-TAE-UNIX-SYSV1F, NASA/GSFC, Greenbelt MD, Sept 1985.

[4] Information Processing Systems - Graphical Kernel System (GKS) Functional Description, ISO 7942, 1985.

[5] IBM Personal Computer Professional Graphics Controller Technical Reference, IBM Personal Computer Hardware Reference Library, Aug 1984.

[6] FG-100-AT User's Manual, Imaging Technology Incorporated Department of Technical Publications, Aug 1986.

[7] Steiner, A. R., Chauvin, J. W., Blattenbauer, J. A., and Kim, Y., "A Versatile Biomedical Imaging System for Personal Computers," Electronic Imaging '87, in press, 1987.

[8] Fahy, J. B. and Kim, Y., "A UNIX-based prototype biomedical virtual image processor," SPIE Medical Imaging, Vol 767, in press, 1987.

END

DATE
FILMED

DEC.

1987