COMPUTER VISION, GRAPHICS, AND IMAGE PROCESSING 25, 331–347 (1984)

HIPS†: A Unix*-Based Image Processing System

MICHAEL S. LANDY, YOAV COHEN, AND GEORGE SPERLING

Human Information Processing Laboratory, Psychology Department, New York University

Received January 13, 1983; revised February 25, 1983

A software system for image processing, HIPS, was developed for use in a UNIX environment. It includes a small set of subroutines which primarily deals with a standardized descriptive image sequence header, and an ever-growing library of image transformation tools in the form of UNIX "filters." Programs have been developed for simple image transformations, filtering, convolution, Fourier and other transform processing, edge detection and line drawing manipulation, simulation of digital compression and transmission, noise generation, and image statistics computation. The system has the useful feature that images are self-documenting to the extent that each image as stored in the system includes a history of the transformations that have been applied to that image. Although it has been used primarily with a Grinnell image processor, the bulk of the system is machine-independent. The system has proven itself a highly flexible system, both as an interactive research tool, and for more production-oriented tasks. It is both easy to use, and quickly adapted and extended to new uses.

1. INTRODUCTION

There is currently rapid growth in the number of research labs involved in image processing. In setting up such a facility, there is a seemingly inevitable period of time during which hardware and software tools are developed in order to provide a convenient and flexible research environment. In setting up our lab, we decided to give some thought to the software environment before we embarked on our various research projects. Our lab includes a variety of image-related tasks at any given time, many of which are carried out by neophytes in computerized image processing. Our problem was thus to provide an environment in which later development of software image processing tools would be easy, and yet provide general flexible tools which were easy to use. A wide variety of image processing related projects are now ongoing, and using these tools we feel that the system has proven its flexibility.

The original project which prompted the construction of this image processing facility was research into the bandwidth requirements for comprehensible yet maximally compressed transmission of imagery of American Sign Language (ASL) [1–3]. (ASL is a gesture language used for communication with and among the deaf.) This research involves both an investigation of the relative importance of the various visual features of ASL imagery, and a measurement of the ability of speakers of ASL to comprehend imagery which has been modified along various parameters such as pixel density, gray scale, spatial frequency content, applied image transformations, and so on. Thus, a computing environment was needed in which sequences of images

*UNIX is a trademark of Bell Laboratories.

†HIPS stands for the *H*uman Information Processing Laboratory's *I*mage *P*rocessing *S*ystem.

0734-189X/84 $3.00

could be input into the computer, transformed in various ways in the spatial and frequency domains, and eventually be presented to experimental subjects in order to test for comprehensibility.

Once the development of this system was made known to other researchers and computer users, it became apparent that any software tools might also be used by several other local projects in the visual sciences. This made the need for a highly flexible and yet easy-to-use system all the more pressing. We therefore decided to expend some extra initial effort in order to provide such a working environment, and we feel that the effort has been repaid handsomely.

2. ENVIRONMENT

The computing environment consists of a VAX 11/750 computer running the Berkeley 4.1 version of the UNIX operating system. This is an operating system designed for generality and portability, with very little hardware-specific code, and thus provided impetus for the design of a software system which is equally portable and non-device specific. The main image processing device is a Grinnell GMR 27-30 image processor (connected to the VAX via a DR11-B DMA interface), which possesses the capability of storing and digitizing video frames into 256 gray levels with a resolution of 480 × 512. It can also convert digital frames back to video, and has a joystick, cursor, alphanumerics, etc. Other peripherals include dot matrix printers (used for half-tone representation of images), film and video cassette equipment, etc. Lastly, a slow-speed parallel interface (a DR11-C) is used to control film and video equipment, and to allow the system to synchronize its image output with the Grinnell's vertical retrace.

The system we have developed is closely tied with the UNIX operating environment [4], and so a brief introduction to UNIX concepts is in order. The two most important ideas here are those of a *filter* and a *pipe*. Standard utility programs in the UNIX system are generally written as fairly simple transformations. The command language is such that every program that is executed has an associated *standard input* and *standard output*. Filters are programs that provide an easily described transform of their standard input as their standard output. For example, a sorting routine might take a list of unsorted items as its input, and yield a sorted set as its output. The sorting parameters (such as type of ordering, keys, etc.) can be provided along with the routine name when the command is entered. Thus, like an ordinary electronic filter, a UNIX filter is merely a transformation of input to output. The UNIX conventions for typing a command with parameters are

$$\text{transform parameter}_1 \cdots \text{parameter}_n \quad < \text{input} > \text{output}.$$

Electronic filters can be cascaded in order to form more complex filtering operations from simple primitive filters. This is also the case with UNIX filters. The operation that provides this cascading is called the *pipe* facility, and in the command language is symbolized as |. Using a *pipe*-line allows one to automatically plug the standard output of one filter into the input of the next. For example, the line

$$\text{transform}_1 < \text{pipe-input} \mid \text{transform}_2 \mid \cdots \mid \text{transform}_n > \text{pipe-output}$$

applies *transform*$_1$ to *pipe-input*, sends its output to *transform*$_2$, and so on, until

finally *transform*$_n$'s output is saved in the file *pipe-output*. The structure of filters and pipes is used extensively in our image processing system.

3. SYSTEM STRUCTURE

General features. Taking our cue from the UNIX pipe and filter structure, we chose to program our image processing routines as filters. Thus, each program would perform only one type of transformation, and complex transformations would be built up, whenever possible, as pipelines of simpler transformations. This sort of implementation is natural given the UNIX operating system. The use of pipelines in image processing systems was demonstrated by Stevens and Hunt [5] as a means of greater efficiency given that the use of temporary files for the storage of intermediate results is avoided.

When applying a filter to a sequence of frames, one has to specify a number of parameters. Generally, the parameters can be divided into two groups: those that describe the image sequence itself, and those that apply to the operation of the filter. For example, any filter needs to know the number of frames in its input sequence, but only the *reduce* filter needs to know how much to reduce an image. Thus, in order to ease our programming efforts, the image parameters were gathered together in a standardized *image header* which was then made an integral part of each image sequence as stored in the computer.

The image header. The use of standardized image headers is by no means new to image processing. It was initially our intent to use an already defined header in our system in order for our images to be portable and easily transferred to other labs, but we found that other such headers were inadequate for our purposes. One such header is the NATO header [6], which is used for picture transmission by ARPA facilities, among others. Although it gives full description of pictures in pixel matrix format, it is rigid in that graphic data in other formats, such as vector representations, cannot be represented. The NATO header also allows only fixed space for image sequence documentation, and is primarily intended as a format for magnetic tape storage of imagery. Other headers which are more geared for parameter passing include an extension of the NATO header called CVL [7], and the header used in the EIDES system [8].

Our design departs from this approach in allowing variable length textual description of images, and allows more freedom in the documentation of picture sequences. Being of variable size, the header can be expanded with no limit. The rationale behind this approach is that in a research environment (as opposed to a production environment in which large volumes of pictures are processed), considerations of execution efficiency and data integrity are secondary to considerations of modularity, simple and effective documentation of image files, and easy software generation and maintenance.

Our image header scheme provides two facilities: the automatic passing of image parameters through a transformation pipeline, and the ability to automatically document an image sequence integral to how the sequence is stored and manipulated. The image parameters in the header include:

number of frames,

number of rows,

number of columns,

bits per pixel,

bit packing (if filler is used to fill out bytes, or not),

pixel format (byte, integer, floating point, complex, vector plot, histogram, quad-tree and hierarchical encoding,...).

The descriptive information includes:

originator's name,

sequence name,

sequence date,

sequence description (a text of arbitrary length),

sequence history.

The originator's name, sequence name, and sequence date are all text fields of arbitrary length which can be used to describe the imagery from which this data was originally derived. The sequence description is also in free text format, and comments may be added to this description at any point. The sequence history is an interesting and useful feature. It contains an executable command string of all transformations that have been applied to this sequence, including the date and time they were run. For example, the following sequence history describes an image which was read in from the Grinnell, reduced by a factor of 4, cropped to 96 × 64 pixels, converted to floating point format, then linearly scaled back to byte format such that the lowest floating pixel maps to 0, and the highest to 255 (which is a cheap way of contrast enhancement):

```
rframe "-D Thu Mar 18 16:17:46 1982" |
reduce 4 "-D Tue May 4 11:19:27 1982" |
extract 96 64 11 19 "-D Fri Sep 17 16:10:10 1982" |
btof "-D Fri Sep 17 17:01:59 1982" |
scale "-D Fri Sep 17 17:02:03 1982"
```

(this string is actually an executable UNIX command string). The quoted parameters which begin with *-D* are dummy parameters which describe when the transformation was applied to this image, and are ignored by all programs. The \ at the end of each line is the UNIX method of saying that a command continues on the next input line. Thus, if the original image was still in the Grinnell's image memory, the image described by this history could be regenerated by executing the history string as a UNIX command.

The image header can be modified in two ways. After an image filter has read an image header, it can simply assign new values to header parameters, or append text to the descriptive information. Also, an image filter exists, called *adddesc*, which allows one to update any or all of the documentary fields in the image header. For

example, the command

adddesc – a "new descriptive information" < input-sequence > output-sequence

will add the string "new descriptive information" to the sequence description.

A small library has been written which includes routines that manipulate image headers. Routines exist which read an image header from the standard input, update the description, update the history, and write the new header to the standard output.

```
/*
 * logimg - takes log of input image.
 * input image is in byte format, output image is floating point.
 * usage:    logimg <seq >oseq
 * to load:  cc -o logimg logimg.c -lhipl -lm
 */
#include <stdio.h>
#include <hipl_format.h>
#include <math.h>
float logtab[256];

main(argc,argv)

char *argv[];

{
        int factor,fr,f,r,c,b,i,j,k;
        char   *pic,*p;
        float  val;

        struct header hd;

        read_header (&hd);
        if(hd.pixel_format != PFBYTE) {
                fprintf(stderr,"logimg: pixel format must be byte\n");
                exit(1);
        }
        r = hd.rows;
        c = hd.cols;
        hd.pixel_format = PFFLOAT;
        update_header(&hd,argc,argv);
        write_header(&hd);
        if ((pic = (char *) calloc(r*c,sizeof(char))) == 0) {
                fprintf(stderr,"logimg: can't allocate core\n");
                exit(1);
        }
        for (i=0;i<256;i++)
                logtab[i] = log((double) (i+1));
        for (f=0;f<hd.num_frame;f++) {
                if (pread(0,pic,r*c*sizeof(char)) != r*c*sizeof(char)) {
                        fprintf(stderr,"logimg: error during read\n");
                        exit(1);
                }
                p = pic;
                for (i=0;i<r;i++) {
                        for (j=0;j<c;j++) {
                                val = logtab[*p++ & 0377];
                                write(1,&val,4);
                        }
                }
        }
        return(0);
}
```

FIG. 1. Logimg program.

Programming a filter. The typical image transformation consists of the steps

(1) interpret the command arguments,

(2) read in the image header,

(3) update the sequence history and other header parameters,

(4) write the header,

(5) read in, transform, and output the transformed image sequence.

The actual implementation of this can be seen in the sample program in Fig. 1. This program, called *logimg*, will take any image with 8-bit byte-formatted pixels and yields a floating pixel image by taking the natural log of each pixel (plus 1, to avoid log(0)). The program *includes* several standard files, one of which describes the image header structure. After reading the input image's header, a check is made that the image is truly in byte format. This particular filter outputs a floating image from an input byte image, leaving all other image parameters constant, therefore the entry in the image header describing the pixel format is changed to indicate floating point pixels, and the header is output. A lookup table is then computed for the logarithm, for efficiency's sake. Lastly, the actual computation is performed, reading in a frame at a time, and outputting each new pixel as it is computed. The program is written in the language *C* [9], which is the standard language in use at UNIX installations, and bears a slight similarity to Pascal.

4. SYSTEM OVERVIEW

In this section we will describe the programs which are currently available in the system. As our research is ongoing, new tools are always being developed, and the list is thus merely a snapshot of the current state of affairs. Since the format of these programs is often much the same from one program to another, we have developed also a tool for developing image transformation programs (see *calcpix*, described under single pixel transformations). The functions in the system include peripheral interface, manipulation of headers, frame generation, frame-by-frame operations, simple frame transformations, single pixel transformations, format conversion, statistics computation, filtering, convolution, transforms, edge enhancement and detection and line drawing manipulation, a 3-dimensional vector plotting package, and digital transmission methods. The programs listed under peripheral interface are the only equipment-dependent programs in the system, making the system more easily transportable. A list of all currently available programs can be found in the Appendix.

Peripheral Interface. Only a small number of programs actually deal with input and output of images to and from peripheral devices. It is here that any machine-dependent code is to be found, and it is these programs which would need to be changed in order to drive an alternative image processor. These include programs to read and write single frames from the Grinnell (*rframe* and *wframe*), erase the Grinnell (*grerase*), output sequences of frames as a video movie (*movie* and *bmovie*), output strings of characters to the Grinnell (*grstring*), output graphs to the Grinnell (*grplot*), and to digitize input from a video camera (*tvc*).

A number of routines exist which control film and video equipment in order to allow sequence digitizing and sequence output to be fully automated. The output lines of a slow-speed parallel interface (DR11-C) are connected to the equipment's

remote control lines in order to effect this control. These programs can control a video tape recorder (a Betamax: *betacucnt*, *betacuew*, *betacurec*, *betapause*, *betaplay*, *betastop*), a video disk recorder (a Sony video motion analyzer: *sonyfwd*, *sonyfwdst*, *sonyrec*, *sonyrv*, *sonyrvst*), and a 16 mm film projector (a Lafayette motion analyzer: subroutine *pstep*, which is used in program *rseq*). A fully automated sequence input routine for conversion from film is also available (*rseq*).

Lastly, two routines exist which attempt to display images as halftones on an Anadex dot-matrix printer by controlling individual dots. The first, *prthlf*, uses zero to sixteen of the dots in a 4 × 4 grid with one possible overstrike in order to give 32 gray levels. The other, *prtdth*, uses the "dithering" technique [10] to give up to 256 levels of gray.

Header manipulation. Three routines allow for the manipulation of image headers. *Seeheader* outputs the header in a readable format, allowing one to examine image documentation. An example of seeheader output is given in Fig. 2. Two other utilities, *grabheader* and *stripheader*, allow one to separate headers from image sequences, and vice versa. Lastly, *adddesc* can be used to update the informational portions of the sequence header (the description, name, etc.).

Frame generation. Genframe and *fgenframe* can be used to generate homogeneous fields with byte and floating pixels, respectively. *Checkers* generates checkerboard patterns. Sinewave gratings can also be generated by creating a power spectrum with *genframe* and *pad*, converting to complex with *btof* and *ftoc*, and then applying the inverse Fourier transform *inv.fourtr*.

Frame-by-frame operations. Catframes can be used to concatenate single frames or short sequences into longer image sequences. *Subseq* allows the user to extract

```
Original name:          Nancy - Sentence 2
Sequence name:          B.I.128.30.6
Number of frames:       1
Original date:          10/5/81
Number of rows:         96
Number of columns:      64
Bits per pixel:         8
Bit packing:            No
Pixel format:           Bytes

Sequence history:

rseq "-D Thu Mar 18 16:17:46 1982" :\
reduce 4 "-D Thu Mar 18 17:05:05 1982" :\
extract 96 64 11 19 "-D Fri Sep 17 16:10:10 1982" :\
subseq 20 20 "-D Fri Sep 17 17:01:58 1982" :\
btof "-D Fri Sep 17 17:01:59 1982" :\
scale "-D Fri Sep 17 17:02:03 1982" :\
mask -f 261 "-D Fri Sep 17 17:04:45 1982" :\
thresh 6 "-D Fri Sep 17 17:04:46 1982" :\
neg "-D Fri Sep 17 17:04:49 1982"

Sequence description:

One frame was taken from the original sequence.
It was reduced, cropped, contrast enhanced.
Then it was convolved with a mask which approximates a
Laplacian operator, thresholded so that 6% of the pixels
become white, and then negated so that the drawing
appears as black-on-white.
```

FIG. 2. Sample *seeheader* output.

subsequences from a given sequence (including skipping frames). *Repframe* simulates the use of frame repetition or frame interpolation (by pixel averaging) as a means of image compression. A sequence of images can be compressed into one averaged image with *strobe*. Two sequences can be compared, yielding a sequence of differences between frames with *diffseq*. Lastly, a sequence consisting of differences between successive frames in a single sequence can be created with *autodiff*.

Simple frame transformations. The routines *reduce*, *enlarge*, *reflect*, *rotate180*, and *pictranspose* do exactly what their names imply. An image sequence can be inserted in a fixed gray-level background with *pad*. A sequence of smaller subpictures can be extracted from a sequence using *extract*. Lastly, the frames of a sequence can be multiplied pixel by pixel by a given fixed frame using *mul*. This last program can provide a means of generalized filtering if performed in the Fourier transform domain.

Single pixel transformations. The routine *neg* produces a photographic negative, converting black to white and vice versa. *Shiftpix*, *powerpix*, and *stretchpix* allow one to shift the gray scale in various ways. *Logimg* takes the natural log of an image. *Thresh* thresholds an image sequence, yielding an image which consists entirely of black and white pixels.

In an image processing lab, writing simple single-pixel-oriented transformations is a fairly rote process, and such transformations are often needed for special one-time-only purposes. The program *calcpix* allows the user to create a new filter which can transform a sequence of byte-formatted frames where a user-supplied series of *C* statements are applied to each pixel in the images. Variables are supplied so that the user can refer to neighboring pixels, the row and column number, use local variables, call *C* subroutines, and so on. For example, the line

```
calcpix "if (ipix > 50 && ipix < 100) opix = 255; else opix = 0" < in > out
```

will transform the sequence "in" into a black and white image where pixels which ranged from 51 to 99 in the input image become white (255), and all others become black (0). The resulting sequence is stored in file "out." *Calcpix* also leaves a copy of the specially tailored filter in the user's directory, which can then be applied to other images.

Pixel format conversion. Btof converts byte images to floating format, and *ftoc* converts floating images to complex format. *Bpack* and *bunpack* convert byte images to and from bit-packed one-bit-per-pixel images, respectively. Scale takes a floating image, and linearly scales the gray scale such that the smallest pixel value maps to 0, and the largest to 255, yielding a byte formatted image. The simple pipeline "btof | scale" thus provides a means of linearly stretching image contrast.

Image statistics. The mean and variance of pixels in a given image are computed by *framevar*. Two programs perform entropy calculations. *Pixentropy* computes first and second order entropy on byte-formatted imagery. *Entropy* computes the entropy of sub-blocks in a sequence of single bit-per-pixel imagery (i.e., black and white imagery), including three-dimensional sub-blocks. Two programs compute image gray-level histograms, *histo* and *disphist*. *Histo* computes the gray-level histogram of an image or sequence, creating a new sequence complete with an image header in which the pixel format denotes that this image is in histogram format. *Disphist* displays such histograms, creating an image sequence in byte format which can then

be displayed on the Grinnell or printed. The output of *histo*, on the other hand, may be used by programs which need the values of the histograms (such as thresholding and contrast enhancement programs).

Image noise. Bit reversal noise can be added to a sequence with *noise*, and Gaussian noise with *gnoise*. *Fgnoise* is a faster, less accurate version of gnoise.

Filtering, convolution, transforms, edge enhancement, and edge detection. Several programs and library functions exist for transform domain processing. Library subroutines *fft*, *fftn*, and *fft_2d* perform fast Fourier transforms, and *dct_2d* and *dctinv_2d* perform the two-dimensional discrete cosine transform. *Fourtr* and *inv.fourtr* transform image sequences to and from the spatial frequency domain. In that domain, *highpass*, *lowpass*, and *bandpass* may be employed to perform filtering, where the filters are characterized as ideal, exponential, or Butterworth, and by their slopes and cutoff frequencies. *Dog* can be used to filter images with Gaussian filters, or the difference of two Gaussians of different variance (a "dog" filter, used as an approximation of the Laplacian of a Gaussian). *Fourtr3d* applies a three-dimensional Fourier transform to a sequence of images, where time is the third dimension. *Walshtr* and *inv.walshtr* apply the forward and back Walsh transforms [11] to an image sequence.

Convolution of byte-formatted image sequences with fixed masks is performed with the program *mask* (the comparable function for floating pixel images is carried out by *fmask*). The program is actually capable of performing several convolutions at each pixel, and combines these results at each pixel with a specified function, such as maximum mask output, sum of mask outputs, sum of absolute value of mask outputs, and so on. The set of masks and function are specified in a mask descriptor file, and a large number of descriptors are available in a library, including most mask-oriented edge enhancement algorithms, such as those of Prewitt [12]. Roberts [13], Sobel (in [14]), Kirsch [15], Abdou [16], Kasvand [17], Eberlein and Weszka [18], and Robinson [19], and Laplacian approximations (Prewitt [12]). For single mask outputs, the convolution is precisely equivalent to a linear filter, and the program *maskseq* will convert the mask description into an image so that the power spectrum of the corresponding filter can be computed by *fourtr*. Recently, Marr and Hildreth [20] suggested an edge-detection scheme wherein zero crossings are located in an image which has been filtered by a Laplacian. In our system, the zero crossing computation is carried out by *zc*.

Other nonlinear filters include *median* and *extremum* [21] which replace pixels with the median and closest extremum value, respectively, in a given neighborhood around the pixel. *Discedge* applies the discrete edge-fitting procedure of Shaw [22] to an image sequence. *Discedge2* applies the same algorithm to a series of overlapping neighborhoods, and gives for each pixel, the thresholded output of each of those applications at that point. Another edge-fitting algorithm described by Abdou [16] is available (*abdou*).

We have also been working on methods for transforming edge-detected images into line drawings. So far, two programs have been developed along these lines. *Bclean* is intended to aid in noise cleaning of binary imagery. It deletes white pixels (in white-on-black images) which are in 8-connected components of extent smaller than a user-specified size (e.g., it can remove isolated pixels). *Thin* can both thin an image, resulting in the same 8-connected groups which are generally thinned to one-pixel breadth, and then categorize the pixels as to being endpoints, branch

points, isolated points, and so on, in a manner similar to that described by Sakai *et al.* [23]. Lastly, *thicken* can take thinned images and increase their "contrast" by thickening the remaining pixels to two-pixel width lines, or more, yielding an image which is sometimes more visually acceptable.

3D plotting package. Programs to generate and manipulate three-dimensional graphs were added to the image processing system in order to be able to generate synthetic line drawings. In addition, edge-detection and boundary-following schemes can result in point and vector representations, so this package is useful in our line drawing research. The plotting package employs a special image format and allows one to manipulate graphs in space and over time.

There are two programs that generate simple graphs: *gpoly* generates polygons (including points and lines), and *gcube* generates cubes. Three families of programs, each consisting of three programs, manipulate graphs in space and over time. The *g*-family (programs *gmag*, *gshift*, and *grot*) scale, translate and rotate graphs, respectively; they all apply a fixed transformation to the coordinates of the input graph. The *t*-family (programs *tmag*, *tshift*, *trot*) "stretch" a graph over time. *Tshift*, for example, will create a specified number of new frames, each consisting of the original graph shifted by a constant distance relative to the preceding frame. Thus, the result of applying *tshift* to a frame is a dynamic sequence which depicts movement of the objects at a constant velocity. The third family, the *v*-programs (programs *vshift* and *vrot*), manipulate the coordinate system in a specified number of frames of the input sequence, corresponding to shifts in the viewer's reference frame.

In addition to these three families, *pstrobe* collapses a graphic sequence into a single frame; *psubseq* allows extraction of a sub-sequence from an input sequence, and *gsync* synchronizes several sequences, i.e., combines several dynamic "graphic worlds" into one. Polar projection of a graph, with a specified focal point, can be created by *view*. Graphic sequences are obtained from images in pixel-by-pixel point representation with *pixto3d*, and 3D graphs can be converted to the standard *plot* format of UNIX (in which 2D graphs are represented) by *plot3tov*. The numerical representation of the graphs can be inspected by the user via the *seeplot* program.

Digital transmission methods. Several programs exist which compress sequences using traditional digital transmission schemes. Berkeley UNIX already provides an adaptive Huffman coding [24] program called *compact*. In addition, the image processing system now includes *hier_r* and *hier_t* (hierarchical coding), *dpcm_r* and *dpcm_t* (DPCM encoding [11]), and *btc* (block truncation coding [25]). Compression of binary (i.e., one bit per pixel) images using hierarchical coding into quad-trees [26] is accomplished with *quad*, and the inverse transformation with *quad_r*.

5. USING THE SYSTEM

As we have mentioned, one of the strong points of the system is its flexibility and ease of use as an image processing research tool. In order to give some feel for the flavor of the system, this section will attempt to give some idea of a typical use of the system.

Let us assume that we have a film of a sentence in American Sign Language, which is to say, imagery of a frontal view of someone gesturing for a period of about three seconds. Furthermore, assume that the intent is to come up with a new image, derived from the original, which is more of a line drawing and yet, for speakers of

ASL, preserves the information inherent in the gestures. This, of course, is not an arbitrary example, but comes from one of the lines of research being followed in our lab [3]. The plan of action will involve reading the sequence into the computer, applying a variety of operators to the image, and previewing the results on a video monitor.

In order to read in the sequence, the film projector is set up to project the sequence under computer control, and a video camera is focused on the image and connected to the Grinnell. Note that this requires that the film projector be capable of showing single frames for extended periods: we use a film motion analyzer for this purpose. The Grinnell is initialized, and instructed to begin digitizing the input from the camera by typing

```
grerase; tvc.
```

At this point, the first film image will appear on the video monitor, and the equipment can be adjusted for best resolution. A single frame can now be stored by typing

```
rframe >fframe; grerase.
```

The screen erase also stops digitization. Now one can try various combinations of *extract* piped to *wframe*, in order to best crop the image to include only the required information. The image will also be reduced in size in order to save space. This allows us to show sequences at real-time speed by loading the entire sequence into computer memory and outputting to the Grinnell as quickly as possible (synchronized with the video vertical sync pulses). Once the proper cropping and reduction have been determined, the sequence may be read in by typing

```
grerase; tvc
rseq 120 -l "ASL sent. 1" | reduce 4 | extract 96 64 20 12 > seq.
```

This will read in 120 frames of film, and reduce and crop it to 96 × 64 pixels per image. The sequence may be previewed by typing

```
grerase; movie < seq.
```

At this point, it is time to play with various image operators in order to gauge their effects. Since filtering, convolution, and other such operators tend to be time consuming, it is often best to examine a single frame at first, and then apply the operator that appears to be the most useful to the entire sequence. Hence, we pull out a frame from the middle of the sequence:

```
subseq 60 < seq > frame60.
```

The simplest types of operators involve convolution of the image with a small number of masks, and then applying some function (such as the sum, sum of squares, maximum, etc.) to the output of these masks. All of these functions are incorporated into the *mask* program, which takes as an argument a description file (or built-in description) of a sequence of masks and function choice. The output of

the *mask* program can then be thresholded, yielding a purely black and white image. For example,

mask -f 14 < frame60 | thresh 10 | wframe

will display the effects of a 2×2 Roberts edge-detection technique, thresholded such that approximately ten percent of the pixels are classified as edge (i.e., above threshold). By repeating the command with different values for the $-f$ parameter, other convolution techniques may be tried. We have found that approximations to the Laplacian have a good appearance, and several mask approximations can be called up in the above manner. In addition, a program exists which computes the difference of two Gaussians of unequal variance as applied to the image, which constitutes a good approximation to the Laplacian of an image convolved with a Gaussian (see Marr and Hildreth [20]), and can be computed more quickly. Thus,

dog .6 7 < frame60 | thresh 10 | wframe

will apply a difference of Gaussians as approximated by a seven-pixel wide mask, where the standard deviation of the narrower Gaussian is 0.6 pixels, and of the wider Gaussian is 0.6 * 1.6 or 0.96 pixels (the ratio of 1.6 is the default), and then threshold the enhanced image, yielding a binary picture.

After trying several other techniques, such as *abdou*, *discedge*, and so on, let us assume that the difference of Gaussians appeared to be the most promising. The next step is to apply the same transform to the entire sequence. Thus,

dog .6 7 < seq | thresh 10 | neg | bpack > threshseq

will store the binary sequence in bit-packed form. The sequence was photographically negated because edge pictures often look more pleasing to the eye as black-on-white rather than the opposite. The bit-packing increases the efficiency of storage and of real-time display. The new sequence may now be viewed by typing

bmovie < threshseq.

A sequence of frames from the movie may be printed for posterity by typing

subseq 0 120 10 < threshseq | bunpack | prthlf | lcat.

The *subseq* yields every tenth frame of the sequence, the *bunpack* converts it back to byte-format, the *prthlf* formats it for printing, and *lcat* sends it to the printer. Lastly, if at some future time you wish to see how threshseq was generated, type

seeheader < threshseq

which would yield something similar to Fig. 2.

Several features of the system can be noted from this foray into the use of the system. Note that most programs require no parameters to govern their operation; most of the needed parameters are obtained from the image header. Other parameters are often derived from useful defaults (such as *movie* and *wframe*, which output their images centered in the video screen, unless otherwise instructed). Image filters

can be combined easily in a variety of ways given the UNIX pipeline facility, and as noted in [5], this saves on needless hassle and an explosion of unneeded temporary files, in addition to being more time-efficient. For more production-oriented tasks, such as applying the same set of transformations to a large series of image sequences, the programming language-like features of the UNIX command language combined with the image processing filters allow for a fully automated process of image production and processing.

6. CONCLUSIONS

The system we have described, HIPS, has been used for over a year, and has proven itself to be both easy to use and flexible. Programming new image filters is a very easy task, and allows us to continually develop new tools as needed without spending the bulk of our time programming. The ability to compound primitive image transformations through the use of pipes allows the user to quickly examine the capabilities of each transform under various conditions and applied to various originals. This makes preliminary research in the area of image processing an interactive process, and quite enjoyable. Combined with automatic control of film and video equipment, and the use of command language scripts (another UNIX feature), the automatic generation of tapes and films of images and image sequences is also an easy task. Finally, the automatic documentation of images is enormously helpful as the number of images saved in the computer increases (and your average user, the authors included, loses track of what is what).

The development of this software is an ongoing task, and new tools are continually being developed. The work was performed by people interested in using the tools, rather than for development of a distributable package. Thus, the software is not utterly complete. Several programs do not handle all the pixel formats that they might. For example, *neg* can negate byte, bit-packed, and floating point, but not complex images. These sorts of features, along with the addition of obviously useful tools, are added as needed.

We regard the flexibility of the system as its main virtue. For example, we are considering extending the system to be able to handle images which are not in pixel matrix representation. One step in that direction was accomplished when the hierarchical coding programs were written simply by adding format codes for that case. All programs check for the pixel format that they can handle, and so needless confusion is easily avoided. We are now considering extensions of the plotting package, and various schemes which produce line drawings from binary pictures.

APPENDIX

The following programs are available as of the time of this writing:

Program	Synopsis
abdou	Edge-fitting technique [16].
adddesc	Add descriptive information to an image header.
autodiff	Generate differences between successive frames.
bandpass	Parameterized bandpass filtering.
bclean	Clean binary images.
betacucnt	Control the Betamax.

Program	Synopsis
betacuew	Control the Betamax.
betacurec	"
betapause	"
betaplay	"
bmovie	Display a binary sequence.
bpack	Pack a sequence as one bit per pixel, bit packed format.
btc	Block truncation coding [25].
btof	Convert byte to floating point format.
bunpack	Unpack from bit packed to byte format.
calcpix	Create new image filters.
catframes	Concatenate separately stored frames into a sequence.
checkers	Generate a checkerboard pattern.
diffseq	Difference of two image sequences.
discedge	Edge-fitting technique [22].
discedge2	Overlapping neighborhoods of discedge.
disphist	Display a histogram.
dispwbasis	Display the Walsh transform basis functions.
dog	Apply a Gaussian filter or difference of Gaussian filters.
dpcm_r	Dpcm encoding receiver.
dpcm_t	Dpcm encoding transmitter.
enlarge	Enlarge an image.
entropy	Compute image sub-block entropy on 1-bit-per-pixel imagery.
extract	Crop an image sequence.
extremum	Nonlinear filter for edge sharpening.
fgenframe	Generate a homogeneous floating point image.
fgnoise	Add Gaussian noise (fast version).
fmask	Floating point image convolution.
fourtr	Fourier transform.
fourtr3d	Fourier transform in three dimensions (including time).
framevar	Image statistics.
ftoc	Convert floating to complex.
gcube	Generate a vector plot of a cube.
genframe	Generate a homogeneous byte-formatted image.
gmag	Globally scale a vector plot.
gnoise	Add Gaussian noise.
gpoly	Generate a vector plot of a polygon.
grabheader	Pull the header from an image.
grerase	Erase the Grinnell screen.
grot	Globally rotate a vector plot.
grstring	Write text on the Grinnell.
gshift	Globally translate a vector plot.
hier_r	Hierarchical coding receiver.
hier_t	Hierarchical coding transmitter.
highpass	Parameterized highpass filtering.
histo	Compute an image gray-level histogram.
inv.fourtr	Inverse Fourier transform.
inv.walshtr	Inverse Walsh transform.
logimg	Take the natural log of an image.
lowpass	Parameterized lowpass filtering.
mask	Image convolution.
maskseq	Convert a mask to an image.
median	Nonlinear filter for image smoothing.
movie	Display an image sequence in real time.
mul	Multiply a sequence by a fixed frame.
neg	Negate an image.
noise	Bit reversal noise.

Program	Synopsis
pad	Pad an image with a homogeneous background.
pictranspose	Transpose an image.
pixentropy	Compute entropy of an image.
pixto3d	Convert a byte image to vector plot format.
plot3tov	Convert vector plot image to Unix plot format.
powerpix	Stretch contrast with a power function.
prtdth	Print halftone using a dither matrix.
prthlf	Print halftone using dot density.
pstrobe	Collapses a sequence of vector images to a single image.
psubseq	Extracts a subsequence from a sequence of vector plot images.
quad	Hierarchical coding compression.
quad_r	Hierarchical coding receiver.
reduce	Reduce an image using pixel averaging.
reflect	Reflect an image.
repframe	Simulate compression using frame repetition.
rframe	Read a frame from the Grinnell.
rotate180	Rotate an image 180°.
rseq	Read in a sequence from film.
scale	Linearly scale a floating image to fit in byte format.
seeheader	Print out image header information in a readable format.
seeplot	Inspect the numerical representation of a vector plot.
shiftpix	Binary shift pixel values.
sonyfwd	Control the Sony motion analyzer.
sonyfwdst	"
sonyrec	"
sonyrv	"
sonyrvst	"
stretchpix	Stretch pixel contrast.
stripheader	Strip the header from a sequence.
strobe	Collapse a sequence to a single frame by averaging.
subseq	Extract a subsequence from an image sequence.
thicken	Thicken a thinned binary image.
thin	Thin a binary image and categorize the remaining pixels.
thresh	Apply a threshold to an image.
tmag	Scale a vector plot over time.
trot	Rotate a vector plot over time.
tshift	Translate a vector plot over time.
tvc	Start the Grinnell digitizer.
view	Compute polar perspective for a vector plot.
vrot	Rotate the viewer coordinates in a vector plot over time.
vshift	Translate the viewer coordinates in a vector plot over time.
walshtr	Forward Walsh transform.
wframe	Write a frame on the Grinnell.
zc	Compute zero crossings.

The following subroutines are available as of the time of this writing:

Subroutine	Synopsis
dct	Discrete cosine transform.
fft	Fast Fourier transform.
ffwt	Fast Walsh transform (floating point).
fwt	Fast Walsh transform (integer).
init_header	Initialize an image header.
pstep	Step the film projector.

Subroutine	Synopsis
read_header	Read an image header from the standard input.
update_desc	Add information to the sequence description.
update_header	Update the sequence history.
write_header	Write an image header to the standard output.

ACKNOWLEDGMENTS

The preparation of this article and the work on image processing of American Sign Language was supported by the National Science Foundation, Science and Technology to Aid the Handicapped, Grant PFR-80171189. M. Pavel provided the technical expertise needed to coordinate the many facets of the system and made helpful comments on the manuscript. We wish to acknowledge the assistance of Thomas Riedl and Robert Picardi, and also O. R. Mitchell, who made available his computer programs for block truncation coding.

REFERENCES

1. G. Sperling, Bandwidth requirements for video transmission of American Sign Language and finger spelling, *Science* **210**, 1980, 797–799.
2. G. Sperling, Video transmission of American Sign Language and finger spelling: Present and projected bandwidth requirements, *IEEE Trans. Comm.* **COM-29**, 1981, 1993–2002.
3. G. Sperling, M. Pavel, Y. Cohen, M. Landy, and B. Schwartz, Image processing in perception and cognition, in *Physical and Biological Processing of Images*; Rank Prize Funds International Symposium at the Royal Society of London (O. J. Braddick and A. C. Sleigh, Eds.), Springer, Berlin, 1982.
4. D. M. Ritchie and K. Thompson, The UNIX time-sharing system, *Bell System Tech. J.* **57**, 1978, 1905–1929.
5. W. R. Stevens and B. R. Hunt, Software pipelines in image processing, *Comput. Graphics and Image Process.* **20**, 1982, 90–95.
6. J. S. Dehne, The NATO RSG-4/SGIP tape format, in *Proceedings of the Workshop on Standards for Image Pattern Recognition*, National Bureau of Standards Special Publication 500-8, Washington D.C., 1977.
7. R. L. Kirby, R. Smith, P. Dondes, S. Ranade, L. Kitchen, and F. Blonder, Interfaces, Subroutines, and Programs for Grinnell GMR-27 Display Processor, University of Maryland Computer Science Technical Report TR-810, 1979.
8. H. Tamura, Image database management for pattern information processing studies, in *Pictorial Information Systems* (S. K. Chang and K. S. Fu, Eds.), Lecture Notes in Computer Science, Vol. 80, pp. 198–227, Springer-Verlag, New York, 1980.
9. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice–Hall, Englewood Cliffs, N.J., 1978.
10. J. F. Jarvis, C. N. Judice, and W. H. Ninke, A survey of techniques for the display of continuous tone pictures on bilevel displays, *Comput. Graphics and Image Process.* **5**, 1976, 13–40.
11. R. C. Gonzalez and P. Wintz, *Digital Image Processing*, Addison–Wesley, Reading, Mass., 1977.
12. J. M. S. Prewitt, Object enhancement and extraction, in *Picture Processing and Psychopictorics* (B. S. Lipkin and A. Rosenfeld, Eds.), pp. 75–149, Academic Press, New York, 1970.
13. L. G. Roberts, Machine perception of three-dimensional solids, in *Optical and Electrooptical Information Processing* (J. T. Tippett *et al.*, Eds.), pp. 159–197, MIT Press, Cambridge, Mass., 1965.
14. R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis*, Wiley, New York, 1973.
15. R. Kirsch, Computer determination of the constituent structure of biological images, *Comput. Biomed. Res.* **4**, 1971, 315–328.
16. I. Abdou, Methods of Edge Detection, University of Southern California, Image Processing Institute Report, No. 830, 1978.
17. T. Kasvand, Iterative edge detection, *Comput. Graphics and Image Process.* **4**, 1975, 279–286.

18. R. B. Eberlein and J. S. Weszka, Mixtures of Derivative Operators as Edge Detectors, *Comput. Graphics and Image Process.* **4**, 1975, 180–183.
19. G. S. Robinson, Edge Detection by Compass Gradient Masks, *Comput. Graphics and Image Process.* **6**, 1977, 492–501.
20. D. Marr and E. Hildreth, Theory of Edge Detection, MIT Artificial Intelligence Lab. Memo, No. 518, 1979.
21. J. M. Lester, J. F. Brenner, and W. D. Selles, Local transforms for biomedical image analysis, *Comput. Graphics and Image Process.* **13**, 1980, 17–30.
22. G. B. Shaw, Local and regional edge detectors: Some comparisons, *Comput. Graphics and Image Process.* **9**, 1979, 135–149.
23. T. Sakai, M. Nagao, and H. Matsushima, Extraction of invariant picture sub-structures by computer, *Comput. Graphics and Image Process.* **1**, 1972, 81–96.
24. D. A. Huffman, A method for the construction of minimum redundancy codes, *Proc. IRE* **40**, 1952, 1098–1101.
25. O. R. Mitchell and E. J. Delp, Multilevel graphics representation using block truncation coding, *Proc. IEEE* **68**, 1980, 868–873.
26. M. Shneier, Two hierarchical linear feature representations: Edge pyramids and edge quadtrees, *Comput. Graphics and Image Process.* **17**, 1981, 211–224.