A PRIMER ON IMAGE MANIPULATION USING A HIGH RESOLUTION COLOR DISPLAY TERMINAL

by

Pat Hanrahan
Randy Schulz
Leonard Uhr

Computer Sciences Technical Report #476

May 1982

A Primer on Image Manipulation Using a High Resolution Color Display Terminal.

Pat Hanrahan

Randy Schulz

Leonard Uhr

Image Processing Laboratory
University of Wisconsin
Madison, Wi.

Introduction

This primer has been written to introduce new users to the graphics and display capabilities at the University of Wisconsin Image Processing Laboratory. The major hardware components of this laboratory are a Stanford Technology Corporation (STC) Color Display System, with a PDP 11/45 as host, and a high-speed link to a VAX 11/780. The software comprising the system has been closely integrated into the UNIX* operating system. This combination of hardware and software provides a powerful system supporting the basic processes of image manipulation. These include:

(i) The digitization or synthesis, and subsequent archival and/or display of high resolution color imagery.

(ii) The application of a wide range of standard pre-programmed operations to single images, or to sequences of images.

This primer teaches use of the graphics and display capabilities of the laboratory by actually running the major command line programs that control the display terminal. To get started we will emphasize those operations common to a range of applications, from image enhancement, image understanding and pattern recognition to computer graphics and image

*UNIX is a Trademark of Bell Laboratories.

synthesis. The tools provided form a nucleus of techniques which are used over and over in any application. But, this core of techniques only touches on the capabilities of the hardware and software. In particular, the image processor has a pipelined arithmetic logic unit, which essentially gives the user a general-purpose computer that can, in only 30 milliseconds, execute a single operation on every one of the quarter million cells of a 512 by 512 image; this is not described here.

But to begin, it is only necessary to read and experiment with the examples in the text. During this period you will be using a large package of programs written in C and many features of the UNIX operating system. However, to use these programs it is not necessary to appreciate or understand what an operating system does, or for that matter what it is. It will not be necessary to be able to code your own programs. The entire primer is meant to be self contained, so no commands are needed except for those indicated in the text. But be forewarned, to really appreciate the complexity of image processing and computer graphics you will need to study in more detail the operating system and programming packages available. At the end of this primer are several very good references to guide you to this information.

Logging on

First, you must make arrangements to use the computer. Somebody must put the login name and password you choose into the computer, tell you what terminals you can use, and perhaps give you some special instructions about the peculiarities of the system not mentioned in this introduction. Also ask to have the images and other files referred to in this primer put in your directory. Once this is done you are ready to login.

Go to the laboratory and find a free terminal next to the color monitor. If the terminal and monitor are off, turn them on (the <power> switch for the monitor is in the lower left hand corner; the terminal's power switches are located on the side or the back panels). Most systems will then present you with the words (if not, push <carriage-return> several times):

```
Login:
```

to which you should respond with your login name, e.g.:

```
Login: solaris<return>
```

That is, simply type in your login name, followed by a carriage-return.

If the login name requires a password the system will respond:

```
Password: lem
```

in which case you should type in your password and carriage-return. (When typing the password you will not see the characters that you type. This is to protect your password from discovery by others.) You should now be logged into UNIX. After a short pause you will probably receive some messages, e.g.,

```
you have mail
there are new messages
```

Information about the mail and message services of UNIX are not discussed in this primer but can be found in the references at the end.

Then after another short pause your terminal may print several blank lines and respond:

```
Erase set to control-H
```

<Control-H> stands for a single character which is formed by simultaneously holding the <control> and the <H> keys. This

can be done at any time to correct typing mistakes while entering commands. Normally this will cause a backspace over the previous character and when you retype the character it will replace the mistake. On some terminals there is another key labeled <backspace> which is equivalent to the <control-H>. It is also possible to erase an entire line by typing the character <@>.

The login sequence will end with the display of a standard "prompting" symbol, which indicates that UNIX is waiting for your next command. The typical prompt is:

```
%
```

(There are two common prompts, which vary depending which "shell" you are using. The standard UNIX shell, _sh_, uses the prompt "$"; the Berkeley shell, _csh_ or C shell, uses the prompt "%".)

If you plan on working through this primer (which is highly advised) you should now run the command:

```
% primer
```

which insures all the programs and files that are needed for the demonstrations in this document are available and ready to use. If this command cannot be found or returns with an error it is best to immediately consult a local wizard.

A quick overview of the graphics terminal.

Let´s start with a brief overview of the Stanford Technology Corporation graphics terminal (we often abbreviate this as STC, STC-70F, Model 70 or m70) shown schematically in Figure 1. It has several major parts:

i) a set of three refresh memory _channels_ which contain the information which is to be displayed or operated on.

ii) three _pipelines_ which can be programmed to transform the data in the memories before being displayed.

iii) three _digital-to-analog converters_ which change the numbers from the pipelines into voltages that are used to control the brightness osf the red, green and blue phosphors on the color monitor.

iv) a _feedback arithmetic logic unit_ (ALU) that executes logical and arithmetic operations on the output from one of the pipelines and the _accumulator_ (the first two refresh memory channels) and feeds back the results storing them in a memory channel;

Figure 3., contained in Appendix A, shows a detailed diagram of the current system. Also listed in that appendix are all the subunits that are currently configured as part of the STC at the U.W. Image Processing Laboratory. These figures are very helpful, so remember to refer to them frequently while reading this primer.

Loading images into refresh memory channels

Whenever you begin working with the terminal the safest thing to do is to reinitialize the system. You should type:

```
  % ginit
  % gclr
```

Ginit initializes the graphics terminal to standard default conditions. (E.g., the cursor is set to on, scroll registers are set to 0, zoom is set to power = 1. More on this later.) _Gclr_ clears the graphics memory channels, i.e simply sets the contents of the memory equal to 0.

Next, we want to load an image into the terminal´s memory: Try running the sequence:

```
  % seeimg randall.b 1
```

This command copies the image array, named "randall.b", into memory or channel number 1. The image should appear blue (if it doesn´t try repeating the sequence from _ginit_ onwards) and hence the mnemonic ".b" at the end of its name. An image is simply a large rectangular array of numbers indicating the intensity of the light recorded (or possibly computed) at each point. Images stored in files can have almost any dimension, but an image being viewed on the display cannot be larger than the contents of a memory channel, since each memory channel is a fixed size square matrix of 512 by 512 8-bit values. The memory is sometimes referred to as consisting of 8 512 by 512 bit planes, emphasizing that the unit of image manipulation is two dimensional. Each individual memory location is called a _picture element_ or _pixel_ for short. With 8-bits per pixel it is possible to encode 256 different intensity levels at each and every location in the image.

Continue on with the following commands:

```
  % seeimg randall.g 2
```

```
% seeimg randall.r 3
```

What we have now done is load three bands or color separations, one into each channel. The final full color image, results from the mixture of the red, green and blue components which are stored in the display memory of the terminal. Channel 1 of the refresh memories is setting the blue component on the television monitor, 2 the green and 3 the red.

Monochrome or black and white images can also be viewed by placing a single image in all three memory channels. This can be easily done with.

```
% seeimg testpattern.bw all
```

Notice the naming conventions for channels. Each channel is designated by a single number from 1 to 3, since there are three memory channels*. In addition there are special keywords; _all_ for all three channels and _none_ for no channels.

At this point the image should appear in shades of grey. If we type

```
% gclr 1 2
```

We now see the same image displayed in shades of red. The display changed from shades of grey to shades of red because of the way the memory channels drive the color display. Channel 1 is initially set to provide the blue component of the screen image, 2 the green component and 3 the red component. (At least, as we will see in the next section, this is the default set by _ginit_.) Since the same picture was in all three channels, the amounts of red, green, and blue light at each location was equal -- so we saw a grey-level picture. When we cleared channels 1 and 2 the blue and green components were set to 0 and as a result we see only shades of red.

Command usage and man pages

We will discuss only the most common uses of the commands. But for convenience most of the commands contain other options that are very useful at times. Usually, if a command is typed with _no_ parameters a single line reminder will be printed showing how to use the command. (But be wary, this is not always true!) E.g.

```
% seeimg
```

* It is possible for the terminal to contain up to 12 refresh memories. If the system were to contain more channels they would simply be numbered sequentially.

```
  Usage: seeimg <image> [-w <l> <r> <t> <b>] [-S <l> <r> <t>
<b>] [-s <xs> <ys>] [-c] <channels>
```

Square brackets, "[" and "]", indicate that the text within them is entirely optional, the dash, "-", followed by a letter pick a particular option. Following that is a list of parameters relevant to that option. Options are normally omitted because the command is designed to be executed without them. Angle brackets, "<" and ">", mean that you are to fill in that slot with the requested information. For example, <channels> needs to be filled with a channel name such as "1" or "all", and <image> requires the name of an image file such as "testpattern.bw".

This one-liner is meant to be just a reminder of the syntax of the command, not an explanation of all the options and parameters. For more information consult the "man"ual-page entry. These are up-to-date descriptions of the details of the command -- its purpose, the meanings of the options and their parameters, and even known problem areas or "bugs" -- written by the person who wrote the original version of the program. The manual entry is stored on the computer at all times and can be accessed with the command _man_ followed by the name of the program of interest.

```
  % man seeimg
```

will give several screenfulls of information describing this command. (After each screen it may be necessary to type a <space> to advance to the next screen of information.) As you gain expierence you will want to study these pages in depth. For now it is best to ignore them unless you desperately need more information.

Sometimes a situation arises where you are not sure what the name of a command is, or you want to know what programs are available to deal with a particular subject. In these cases consult the permuted index of available graphics programs. This contains all the manual entries indexed by keywords contained in their descriptions. This index can also be accessed online with the _apropos_ command.

```
  % apropos seeimg
  seeimg (local)      - load images
```

Displaying refresh memory channels

With the above aside, let´s return to the display.

As you may have noticed, the appearance of the image depends on the colors in which it is displayed. The last image that was loaded is now in channel 3; if the same image were in channel 1 then instead of appearing red it would appear blue. One of the most powerful features of the graphics terminal is its flexible display modes. It is possible to route the data from any channel to any color. Displaying a single image on all three colors has the same effect as making three copies of the image, one in each channel and displaying each of these on a different color, but without actually making two extra copies of the picture in memory. Typing

```
  % dsp 3 blue
```

switches channel 3 from being displayed in red to being displayed in blue. Even more powerful is the capability to display a channel simultaneously on several colors. This simulates placing multiple copies of the same image in different channels.

```
  % dsp 3 blue green red
```

places the pattern on all three colors. We designate which colors we want an image displayed on by simple mnemonics. The three primary colors are named "red" or "r", "green" or "g", and "blue" or "b". Each of these as we will see later corresponds to a single <u>pipeline</u>. For convenience, combinations of the above are also given keywords: "magenta" or "m" (for both the red and blue pipelines), "yellow" or "y" (the red and green pipelines), "cyan" or "c" (the blue and green pipelines), "white" or "w" or "colors" (for all three pipelines), and "black" (for none of the pipelines). These conventions allow the last command to be more compactly expressed as:

```
  % dsp 3 w
```

That is, just as before, take the contents of channel 3 and display it in white, a combination of the blue, green and red color pipelines.

Now lets look at what is in channel 2 and then load another image.

```
  % dsp 2 w
  % seeimg cat.bw 2
```

The old image in channel 3 has not been destroyed; we can view it again by simply changing the display mode.

```
  % dsp 3 white
```

Such sequences of loading images and switching the display, are often repeated many times while working with the terminal. Different channels are used to store different images or the results of processing those images. If there are several people using the system it is even possible for each to be using a refresh memory independently of the others (although this takes a certain amount of cooperation).

Typing the command:

```
  % dsp none colors
```

will display "no" channels on "all" the pipelines, therefore in effect turning off the display. This can also be done with the

```
  % dsp all black
```

or

```
  % off
```

Finally, if we only type dsp then the current state of the display is printed. For example after a ginit:

```
  % ginit
  % dsp
  blue:   1 in entire screen
  green:  2 in entire screen
  red:    3 in entire screen
```

Tells us the defaults set by ginit.

The dsp program is the command most frequently used with the terminal, so it is a command you may want to learn more about. The first place to look are the manual pages describing it.

These examples show some of the possibilities given the capability of being able to reconfigure the display. Besides switching from image to image these different display modes serve three major functions.

i) Full color viewing This is the configuration established by ginit. It is the most common method of displaying multiband imagery. Each channel contains one band of the composite color image.

ii) Black and white, also pseudocolor or false color viewing (e.g. with dsp 1 white). This is the most common viewing mode for monochrome imagery. It is also used for image enhancement and special graphical displays. See the section on color-mapping.

iii) Combination viewing (e.g. with dsp 1 2 3 b). This is useful for performing operations which are a function of several images. Examples include displaying the ratio of two images, or a dissolve, that is, the weighted combination of two images.

Scroll and zoom

Let's examine exactly what happens when an image is displayed: The image is scanned, and fed into the monitor, pixel by pixel, starting in the upper-left hand corner, and moving left to right within a row, then to the left of the next row, and continuing from the top to the bottom. This process is called "raster scanning," and leads to the somewhat strange coordinate system typically used in image processing - with 0,0 designating the upper-left corner, 511,0 the end of the first row, and 511,511 the last picture element, at the bottom-right.

This "raster scan" takes 30 milliseconds (the standard television scan rate) for the entire 512 by 512 array. It takes place continuously 30 times a second (hence the name "refresh memory"). The host computer can execute several thousand instructions in the 30 milliseconds of a single scan; as a result the host can be working in parallel, preparing the next instruction for the terminal.

This scanning operation is controlled by counters which sequentially access the different elements in the raster. It is possible by controlling the starting position and frequency of these counters to selectively scroll and zoom about the image memory.

```
  % ginit
  % seeimg randall.b 1
  % seeimg randall.g 2
  % seeimg randall.r 3
  % scroll 256 256 all
```

This is the picture of the building we looked at before. The command scroll translates the point, specified as a raster coordinate (x,y), so that it is displayed at the screen origin, the upper left hand corner. In the process of scrolling the picture wraps around, as if it was on a sphere, so that it appears folded, with its original origin at the center of the screen. Notice that the scroll command accepts a channel argument. This is because each channel can be scrolled independently of the others.

```
  % scroll 0 0 1
```

will reset the blue band to its original location, causing it to be "out-of-phase" with the other bands.

You may have noticed the trackball next to the monitor. Try rotating it; as the ball is rotated the gimbals supporting it revolve to record its movement. Changes in the position of the trackball are normally indicated by the screen cursor; these may also be sent to the computer. A very useful application of this is to register images that differ by a translation in the x and y direction. This can be done by "linking" the trackball to the scroll coordinates of the channel. The program _link_ is designed for this purpose. _Link_ connects the channels specified to the trackball and continues to scroll them until any button on the trackball unit is pressed. Realign channel 1 relative to the other channels after typing:

```
% link 1
```

In addition to the scrolling capability, it is also possible to zoom in to magnify smaller regions of the picture.

```
% ginit
% zoom 0 0 2
```

This _zoom_ command will magnify the upper left hand quadrant of the image. The first two numbers are the center of the zoom, the last number the power by which to zoom. So

```
% zoom 256 256 4
```

zooms about the center of screen by a factor of 4. Whenever a zoom is performed the point picked is fixed at its current position and the image is expanded about it. The hardware only allows zoom powers of 1, 2, 4 and 8. Unlike the scroll command it is not possible to zoom a single channel; all channels are zoomed simultaneously.

The zoom can be combined with scroll* by going into interactive zoom and scroll, sometimes referred to as roam mode.

```
% roam
```

This command resets the zoom to 1 and the scroll to 0,0 and then allows them to be changed by pushing various buttons on the trackball unit.

"A" - Increases the zoom times 2.
"B" - Decreases the zoom times 2.
"C" - Toggles roam mode (scroll linked to trackball).

* The details of how a scroll and zoom interact to produce the display are rather complicated. For more detailed information consult the references at the end.

"D" - Quits.

Once your are in roam mode then the zoom is applied about the position of the cursor. Experiment with this command.

Image analysis

At this point we would like to explain some of the basic image analysis programs. These are very useful techniques, not necessarily profound, but upon which the more advanced techniques are built.

One of the most useful commands is the sample program. This returns the value stored in the image memory at the point of the cursor. E.g.:

```
% sample
Channel 1, 2, 3 at x = 256, y = 256: 99, 70, 28
```

Image samples can also be extracted interactively with the examine program. When running this program the buttons on the trackball will perform the following functions.

"A" - Sample at the current cursor location
"B" - Cycle the zoom to the next power.
"C" - Toggle continuous sampling mode.
"D" - Quit.

Try it. Have numbers printed out where the picture is red and compare them to the values in a green region.

```
% examine
```

Two other useful related programs are trace and cslice. Trace interactively reads either a horizontal or vertical strip of the image and plots the intensities. Cslice performs a similar function by coloring all the pixels in a certain intensity range. The bounds of this interval can be controlled with the trackball.

Statistical programs provide information about all the pixels in the image. For example, the program minmax prints out the range of numbers contained in an image.

```
% minmax b
Blue: minimum 0; maximum 138
```

Still more information can be gotten from a histogram of the entire image. The histogram can be calculated and displayed with:

```
% histogram b
1525 pixels at 0, Vertical axis ticks every 573.75 pixels
```

which will draw the histogram in graphics memory (more on the graphics memory later). The abscissa contains the intensity values, ranging from the minimum, 0, to the maximum 255. The ordinate displays the number of pixels that have that intensity value. The number of pixels with value 0 is normally suppressed to keep the graph scaled to the screen.

[One quirk about the last two commands is that their arguments are color pipelines not an image channels. This is because they derive their information from special hardware in the pipeline (see the next section on pipeline processing) as the data are routed to the monitor. As we will see in the next section certain transformations can take place in this pipeline and therefore these numbers may only indirectly reflect the true statistics of the image. However, when preceded by a _ginit_ these numbers will be correct.]

Pipeline processing

The terminal can be configured so that the data stored in any channel can be displayed in any color or combination of colors. _Dsp_ can be thought of as a switch which controls the routing of the data to the red, green and blue guns of the monitor. But as was briefly mentioned, the route the data takes to the monitor involves travelling through a _pipeline_ containing special purpose hardware to transform these data before display. The major elements are a series of look-up tables which replace the original pixel values by new values obtained by indirectly referencing a table. This basic idea is summarized in Figure 2. As each number in the stream reaches the table its value is used to select a new value contained in the value´th position of the table; this new number replaces the original in the output stream. The value 0 picks the 0´th entry in the table, the value 255 picks the 255´th entry, and so on.

To introduce these concepts try the following sequence.

```
  % ginit
  % gclr
  % squares
```

The program _squares_ loads a pattern of small squares into channel 1. The value of the extreme upper left square is 0 and the value of the extreme lower right square is 255. (Try testing this by placing the cursor over these small squares and then using the program _sample_ to read the value in memory.) At this point the squares should be different shades of blue. Switch the display so that channel 1 is displayed on the white pipeline combination.

```
  % dsp 1 colors
```

At this point it is possible to change the pipeline's tables to form a photographic negative of the original picture.

```
  % invert 1 colors
```

Notice that the squares that were previously bright are now dark and vice-versa. This is done _not_ by changing the values in memory (convince yourself of this with the _sample_ command) but instead by using a table look-up to substitute the value _maxintensity_ - _intensity_ for the value _intensity_.

The new value can be _any_ function of the input value -- and therefore it is possible to apply any function of a single pixel to the entire image with this table. The advantage of a look-up table is that its function is applied to the image in real-time, continuously. And even though the hardware cannot compute arbitrarily complicated functions, it is possible to precompute these in the host computer, and then store them in the look-up tables of the graphics terminal. This can be done very rapidly (usually during the vertical retrace part of the raster scan) since the size of a look-up table is only as large as the number of different input values.

Some additional common functions applied by look-up table are:

i) Stretch - performs a linear remapping from a range, old minimum to old maximum, to the range, new minimum to new maximum. Closely spaced intensity values can be "stretched" so that they are displayed over a larger intensity range, increasing the contrast. This same function also allows you to threshold a picture.

ii) Equalize - remaps the intensities so that the histogram of the resulting pictures is flat and all intensities have an equal probability of occurring.

iii) Gamma - Perform gamma correction. The amount of light emitted by most monitors is not a linear function of the applied voltage. If the function can be determined then it is possible to undo this distortion by applying the inverse function with a lookup table.

Let's examine the terminal's pipeline in more detail (you may want to refer to Figure 1. at this point). Actually, each pipeline contains two sets of look-up tables. Each of the three pipelines has an _8-bit in 9-bit out_ (the extra bit is a sign bit) lookup table (named the LUT) for each of the memory channels. That is, there is one lookup table per channel per pipeline*. The outputs of the LUT's are then added together and sent through another table, the output function memory (named the OFM). This table has 10-bits in and 10-bits out. The input to the OFM may range from 0 to 1023. [Precisely how this is done is quite complicated if several images are being simultaneously displayed on a pipe and/or the look-up tables perform a function resulting in a negative number. Some of the technical details about this process will be discussed in the section on the range and constant registers; for more detail consult the references at the end.] Finally the unsigned 10-bit OFM output is converted to a voltage by the digital-to-analog converter. (As we will see later, the output may be redirected to the feedback arithmetic logic unit for further processing and storage in a memory channel.)

The LUT's can be saved and loaded with the following commands:

```
  rdlut <file> <channel> <pipeline>
  wrlut [-f <file>] [-e <expression>] <channels> <pipelines>
```

Since there is a LUT for each channel in each pipeline both a pipeline and channel name must be present. Notice that it is possible to load several tables from a file with one command.

There are similar commands for the OFMs

```
  rdofm <file> <pipeline>
  wrofm [-f <file>] [-e <expression>] <pipelines>
```

In this case there is no channel specified since there is only a single OFM per pipeline. Notice also that it is possible to load more then one OFM simultaneously.

Typing the following commands:

```
  % wrlut
  % wrofm
```

sets the tables to their defaults (as does ginit). The default table for a LUT is the identity mapping. Thus the

* In the current configuration with three channels that means that there are a total of 9 LUTs devoted to the refresh memories. If more channels were to be added there would be correspondingly more LUTs. But in addition to the channels there is also a set of LUTs for the video digitizer and graphics memory, bringing the total to 15.

same 8-bit number is simply copied from the input stream to the output stream. The default OFM is slightly more complicated. In the typical viewing configuration a single channel is displayed on each pipeline. If the OFM performed the identity mapping then the maximum output value that would be sent to the DACs would be 255. But as mentioned above the maximum value possible is 1023. Therefore the screen would be very dim, being at only one-quarter full intensity. To remedy this the input values between 0 and 255 are multiplied by 4. Values above this are set to 1023. This may lead to problems if several images are displayed on a single pipeline since their combined value may be greater than 255 -- causing the output picture to saturate.

The _wrlut_ ad _wrofm_ commands are very powerful since they allow very complicated functions to be calculated and stored in the tables without the necessity of writing a special purpose program. For example, the _invert_ command can be expressed as:

```
% wrlut -e "255 - input" 1 colors
```

We will have more occasion to use this command in the examples later on.

Color-mapping.

The image of squares should still be in memory and being displayed in shades of grey. One of the most powerful capabilities of a lookup table is the ability to do color-mapping. Since there is a separate lookup table for each channel in each color pipeline there exists the possibility of independently setting the red, green and blue components selected from a single value. Try:

```
% rgb 0
Channel 1, value=1: Red 0, Green 0, Blue 0
Channel 2, value=1: Red 0, Green 0, Blue 0
Channel 3, value=1: Red 0, Green 0, Blue 0
```

Rgb sets or tells the contents of the three color LUTs associated with the channels depending on the number or arguments. Executing:

```
% rgb 0 255 0 0
```

sets the 0´th entry in the red table to its maximum value while setting the green and the blue entries to 0. The effect is to color all the pixels whose values are 0 to bright red, in the case of the image being displayed the single small square in the upper-left corner.

The command _rgb_ also has access to a _palette_ of color names. So we could have abbreviated the above with

```
  % rgb 0 red
```

This database of colors can be accessed with the _palette_ program. Typing

```
  % palette red
  red 255 0 0
```

prints out the red, green and blue values for that color. _Palette_ by itself prints all the colors in the database.

This method of making colors is limited by the fact that only 256 different colors can be indexed by a single memory channel. However, the range of colors is still just a great as if the image was being displayed in full color. The space of all possible colors that a monitor can display is called its gamut. This is a three dimensional space whose axes correspond to the primary colors, red, green and blue. Picking a combination of primaries localizes a point in this space; the locus of all possible points is a cube which has sides of length 256, therefore the volume of the cube is 2^{24}, approximately 16 million. The program _color-cube_ shows the outer surfaces of this cube when viewed along its white axis.

```
  % ginit
  % colorcube
```

Pseudocoloring, that is, any mapping from a single number to three numbers, can be imagined to consist of a curve in a three dimensional color space. There are many possibilities for choosing such curves some of which are demonstrated by the following script. These programs are described in more detail under the man entries for "colors" and "patterns". (The last command must be interrupted by pressing the <break> or <del> key.)

```
  % dsp 1 white
  % squares
  % ctable1
  % ctable2
  % vramp 1
  % swath
  % rainbow
  % animate 1
```

Transferring images using the feedback unit

Up to now we have been manipulating and changing the image displayed on the monitor. But the image stored in the STC memory has always remained the same. An enormous amount of power is gained from the capability that allows one to make changes to stored images and then perform another operation on the resulting transformed images, and continuing this in an arbitrarily long sequence of computations. To do this requires that we perform the transformation, then store the resulting transformed image. The power of this operation lies in the speed with which it is performed along with the computational capabilities of the pipeline processors and the arithmetic logic unit. The whole sequence is done in one scan over the channel (in 30msec.), involving 1/4 million picture elements, independently of the much slower host computer!

The basic command is *transfer*, which copies an image through a pipeline, optionally applying the transformations contained within it, via the feedback unit back into a refresh memory.

```
% ginit; gclr
% vramp 1
% transfer 1 b 2
```

We placed a ramp of intensities in channel 1, and then transferred it to channel 2. This is *not* the same as just displaying channel 1 on the blue and green pipelines although the visual effect is the same. Notice how much faster this is than rerunning the program or reloading the image. By using the options, "-l" (for lut), "-o" (for ofm) or "-p" (for pipeline) *we can apply the functional mappings currently in the pipeline* to the image before storing it back into the refresh memory.

```
% invert 1 b
% transfer 1 b 1 -lut
% wrlut
```

Will actually form an inverted version of the image and store it in memory channel 1.

Unfortunately, the *transfer* command has several dangers. First it requires that the source channel be displayed on the pipeline that the transfer proceeds through. Second, when copying from the source to the destination the command tries to undo the function being applied by the pipeline without actually changing the display. This is possible because the output of the ALU can be optionally routed through yet another lookup table, named the input function memory (IFM). [See the commands *wrifm* and *rdifm*.] When the transfer command is executed *it uses the IFM* to undo the transformations applied in the pipeline by the LUT and OFM. The options then act by preventing certain transformations from being undone. But this approach will not succeed if the pipeline functions are not invertable or degenerate (that is, more than 1 value in the refresh memories is mapped to the same output value) and may lead to unpredictable results.

Applications

To demonstrate the power of what you already have learned we will show several more complicated examples. These will require several commands, but illustrate how rather powerful operations can be performed without the necessity of writing any programs.

The first will be the use of the feedback unit to create a copy of a zoomed image. This can be done with the following script.

```
  % gclr; ginit
  % seeimg testpattern.bw 1
  % zoom 256 256 2
  % transfer 1 b 3
  % dsp 3 colors
  % zoom 0 0 1
```

The transfer sends the zoomed image to channel 3. (Normally it is possible to transfer a source channel to the same channel, but when the zoom is in effect this is not possible.)

The second example will be to form the first spatial derivative in either the x or y direction of an image. This can be approximated by subtracting an image from a copy of itself displaced by 1 pixel in the appropriate direction. Continuing with the picture above.

```
  % ginit; gclr 3
  % transfer 1 b 2
  % wrlut -e " 1.0 * input" 1 w
  % wrlut -e "-1.0 * input" 2 w
  % wrofm -e " 4.0 * (0.5 * twos10(input) + 128)" w
  % dsp 1 2 b
```

The above group of commands set up for the derivative calculation. We clone a copy of the picture with the transfer. And then we use the capability to display two images on a pipeline to perform the subtraction. One copy is made positive and the other negative with the appropriate wrlut. The trickiest part involves the OFM. After the subtraction, the result may lie in the range from -255 to 255. So we want to rescale this to lie in the range 0 to 1023 the minimum and maximum output values of the OFM. This is what the expression in quotes does with the following complication; the input to the OFM ranges from 0 to 1023 but when computing the expressions value we must treat the input as being a negative number, in 10-bit twos complement form, this conversion is done with the function twos10. At this point the display will be black since an image is being subtracted from itself. To form the derivative we merely scroll one copy 1 pixel relative to the other, and finally transfer it into the last free channel.

```
  % scroll 1 0 1
  % transfer 1 b 3 -p
```

As a final example we will show how it is possible to convert a color photograph into a black and white image. A nice working color image is that of the building we have been using. First we load each band into a separate channel.

```
  % seeimg randall.b 1
  % seeimg randall.g 2
  % seeimg randall.r 3
```

To convert to a black and white image we must combine the three separations weighted by the relative luminosities of red vs. green vs. blue. A standard black and white television set converts a color television signal to luminosity according to the following equation:

$$BW = 0.30R + 0.59G + 0.11B$$

Where BW is the luminosity, and R, G and B are the intensities of three primaries. This function can be computed within the pipeline and then applied to the picture to form the black and white image.

```
  % wrlut -e "0.11 * input" 1 w
  % wrlut -e "0.59 * input" 2 w
  % wrlut -e "0.30 * input" 3 w
  % dsp 1 2 3 white
  % transfer 1 b 1 -pipe
  % wrlut
  % dsp 1 w
```

This same technique can be used to set the display color to be any linear combination of the images stored in the refresh memory channels, in effect performing a matrix multiplication of the image in real time.

Miscellaneous commands

If you feel comfortable with the last few examples you have become quite proficient at controlling the image processor. Just for completeness, we will briefly describe several hardware capabilities of the display terminal not previously covered. These commands are not used nearly as often as those mentioned above.

Cursor

The cursor (typically a thin arrow pointing northwesterly) is normally linked to the trackball, so that by moving the trackball it is possible to position the cursor. Moving past any edge of the screen will cause the cursor to appear at the opposite edge. Normally the cursor is visible and linked to the trackball, but if for some reason it is not you can query its status with the cursor command:

```
% cursor
Cursor on, linked, no blink, beeper enabled
```

To turn the cursor off, then on, type:

```
% cursor off
% cursor on
```

The cursor is stored in a small square of size 64 by 64 of special memory in the terminal. The contents of this area can be changed by loading a new pattern. E.g. another common cursor shape is a "x" which can be set:

```
% cursor x
```

X is the name of a standard cursor file. (For information about creating your own cursor read about the program mkcurs.) One problem with different cursor shapes is that the center of the cursor area is not the same as the position of the cursor. Instead the hardware records the position of the cursor as the position of its upper left hand corner. Certain cursor shapes may suggest that the pointed-to pixel is at the center and may cause errors. To be safe lets reload the small arrow which points to the true cursor location.

```
% cursor arrow
```

The present location of the cursor can be read with:

```
% cursloc
Cursor at x = 2921 (361); y = 2150 (102)
```

It turns out that the cursor can be positioned anywhere on a grid of 4096 by 4096 points but its visible position is confined to an area of the screen which is 512 square. For this reason the position is reported modulo 512, corresponding to the visible display area. Another annoying problem with the cursor is that its position on the screen may not be the same as the pixel which it points to. The cursor does not undergo the same zoom and scroll operations as do the other channels, hence if the image is being zoomed or scrolled then the position of the cursor might not be equal to the coordinate of the pixel under it.

Graphics overlay memory

In addition to the three refresh memory channels there is another block of memory containing only 4-bits of information per pixel. This memory is named "graphics" and has several uses. First, it is used for drawing graphs and maps and to annotate the display. Second, a single bit plane can be used to drive another black and white monitor called the status monitor. Finally, it can be used to create regions of interest (ROI). The values of the bits in this plane are used by the arithmetic logic unit and videometer to control which areas in the memories are operated on.

All the commands that write into memory channels will also write into the graphics memory by using the keyword "graphics" (this includes the gclr command). Nominally the graphics channel is displayed via a special graphics pipeline* which is independent of the three major color pipelines. The graphics output normally replaces, or overlays, the outputs of the standard color pipelines. To control the graphics unit use the command graphics. With no arguments it reports the status of the graphics unit.

```
% gclr; ginit
% graphics
ROI plane is 0, status video off (plane = 0)
```

A useful command to annotate graphics memory is aecho.

```
% aecho "Writing in graphics memory usually appears
green."
```

Causes the sentence following the aecho to be written in the upper left hand corner. Text can be placed at any position by using a positioning option:

```
% aecho -p 300 400 "uwipl 82"
```

Other options are possible, consult the aecho "man" entry for a list of these.

The minimum - maximum, constant and range registers

There are several additional hardware functions performed within the pipeline. These all exist between the lookup tables and the output function memory.

* Actually in the current U.W. system there is a special LUT for the graphics channel. The graphics channel can be displayed on the pipeline by referring to it as channel 6. This is not a standard option with the STC.

The first is the min-max register, which is what the command minmax reads. It is located immediately before the output function memory. The numbers is reads are treated as signed 10-bit quantities. Remember that these values reflect the transformations due to the LUTs and the adder.

It is also possible to add in a constant. There are three constant register, one in each pipeline, whose contents are added to the adder output. These are normally set to 0. These registers can be set and inspected with the const command.

Finally, it is sometimes possible for the adder to produce numbers greater than 10-bits (this usually occurs if the terminal is expanded to include more than three memory

channels); the range shifter selects 10 contiguous bits from the maximum 13 bit output of the adder. This operation is controlled by the range command and is normally set so that the least significant 10-bits are selected.

Arithmetic logic unit

Finally, we will mention a few things about the arithmetic logic unit, which when used in conjunction with the feedback unit and pipeline processing allows the terminal to perform even more complicated image processing tasks. These capabilities come about by using the ALU to combine the pipeline output with data already stored in a 16-bit accumulator (memory channels 1 and 2). Most common arithmetic (but not including multiplies and divides) and logical operations are possible. It is also possible to apply two functions, one in the "region of interest" (i.e. wherever there is a bit set in the ROI), and another everywhere else.

Examples include programs that perform convolutions over the original image by shifting, multiplying and accumulating the results. This has been used to implement David Marr´s edge detector. Another program allows one to emulate a large "simd" array computer to code pattern matches and other parallel algorithms. Unfortunately, the instruction format required by this subunit is quite complicated and therefore it difficult to control by typing simple commands.

Programming the STC

At this point you know most of the capabilities of the STC for image manipulation and display. If you need more information about the STC or intend to write programs there are several other documents you should consult.

All the command line programs are described in section I of the STC programmer´s manual. The lowest level STC primitives are all C-callable procedures and are described in section III of the same manual. Each subunit on the device has a corresponding procedure (usually named the same as the programs you have learned above) that controls it. There are also a large number of utility procedures to perform useful operations for graphics and image processing.

For a general description of the STC terminal, see the STC Manual, "Product Description Model 70/F: Image Computer and Display Terminal." The most detailed reference on the architecture is "I/O Specification: Model 70/F." Both of these are provided by the manufacturer.

Along with the procedures to control the graphics terminal it is also usually necessary read and write image files. These are stored in a standard format which includes a header of important information about the image, e.g. its size and resolution. There exists a package of procedures to manipulate these image files; see the manual entry under image.

Unix, C and Pascal

The Image Processing laboratory computers all use the UNIX operating system. If you are new to UNIX you will want to learn more about what tools are available. For a very good introduction, read "UNIX for Beginners" by Brian Kernighan. While you are logged on under UNIX you communicate with the operating system through a "shell" which provides a large number of useful features that create a friendlier, personalized environment. At the Image Processing Laboratory people commonly use the Csh; this is described in "An Introduction to the C shell" by William Joy. In the process of program development it is essential to have a good editor. Most people prefer "vi", a screen oriented editor (see "An Introduction to Display Editing with Vi" by William Joy) although the older line oriented editor, ed, is still used (see "An Tutorial Introduction to the ED Text Editor" by Brian Kernighan).

Currently there are two programming languages that can be used with the graphics terminal. The major one is C. All the programs described in the manual have been written in C; if you want to learn C, read the book by its inventors, "The C Programming Language" by Brian Kernighan and Dennis Ritchie. We also have a Pascal compiler. The appropriate manual is: the "Pascal Reference Manual" by Whitesmith, Inc.

Acknowledgements

The Image Processing Laboratory is jointly run by the Computer Sciences Department and the Institute of Environmental Studies. We would like to thank the original directors of the laboratory Frank Scarpace and William Havens for contributing many of the ideas and methods described in this primer.

Appendix A - The complete Stanford Technology Corporation terminal

The University of Wisconsin Image Processing Laboratory has (as of April 1982) an STC-70F display terminal with 3 512 by 512 by 8-bit memories, 1 512 by 512 by 4-bit graphic overlay memory, a video digitizer, 3 pipeline processors (each containing a lookup table per channel, and also including the video digitizer and graphics memory, for a total of 15, and an output function memory), split screen, hardware zoom and scroll, histogram generator (videometer), sum processor (records the minimum and maximum), a feedback Arithmetic-Logic Unit, an input function memory, programmable cursor, tablet trackball, and a 19in. Conrac RGB color monitor, model 5411.

This terminal is connected to the Unibus of a PDP-11/45 via a DMA interface. The PDP-11/45 has approximately 240 MegaBytes of disk storage and 9-track tape drive. A high-speed, 1 MegaByte/sec, link connects to a VAX11/780 with 2.5 MBytes of high speed memory and four disks totaling about 400 MBytes (hardware is in place; software is being completed).

Figure 3 shows the schematic of the image processing terminal provided by the manufacturer.

<u>Appendix B</u> - <u>Index of major graphics commands</u>

aecho	- print arguments in refresh memory
class	- display classifications
coloram	- manipulate graphics ram
colors	- color table maps
const	- manipulate constant registers
cslice	- color level slice
cursloc	- cursor location control
cursor	- cursor subunit control
dsp	- display enable control
dissolve	- change display gradually
examine	- interactively print channel contents
fadein	- change display gradually
fadeout	- change display gradually
flash	- view multiple channels
full	- full color display
gamma	- gamma correction
gclr	- clear refresh memory
ginit	- initialize image processor
graphics	- graphics subunit control
gscale	- draw grey scale
histogram	- compute and display histograms
istat	- image header information
link	- scroll channels, split screen
luts	- color lookup table routines
minmax	- examine limit registers
off	- display enable control
on	- display enable control
palette	- search color data base
patterns	- draw shapes in refresh memory
range	- control range register
rdifm	- read input function memory (IFM)
rdlut	- read lookup table (LUT)
rdofm	- read output function memory (OFM)
reflect	- reverse a refresh memory
rgb	- set/tell lookup table entries
rle	- run length encoding
roam	- interactive zoom and scroll
sample	- read pixel values
scroll	- set scroll coordinates
seeimg	- load images
splitc	- split screen coordintates
store	- store pipeline output in refresh memory (RFM)
stretch	- rescale images
take	- store images
tpict	- read tape images
trace	- plot image intensities
transpose	- transpose a refresh memory
transfer	- perform feedback operation
wrifm	- write input function memory (IFM)
wrlut	- write lookup table (LUT)
wrofm	- write output function memory (OFM)
zoom	- magnify images

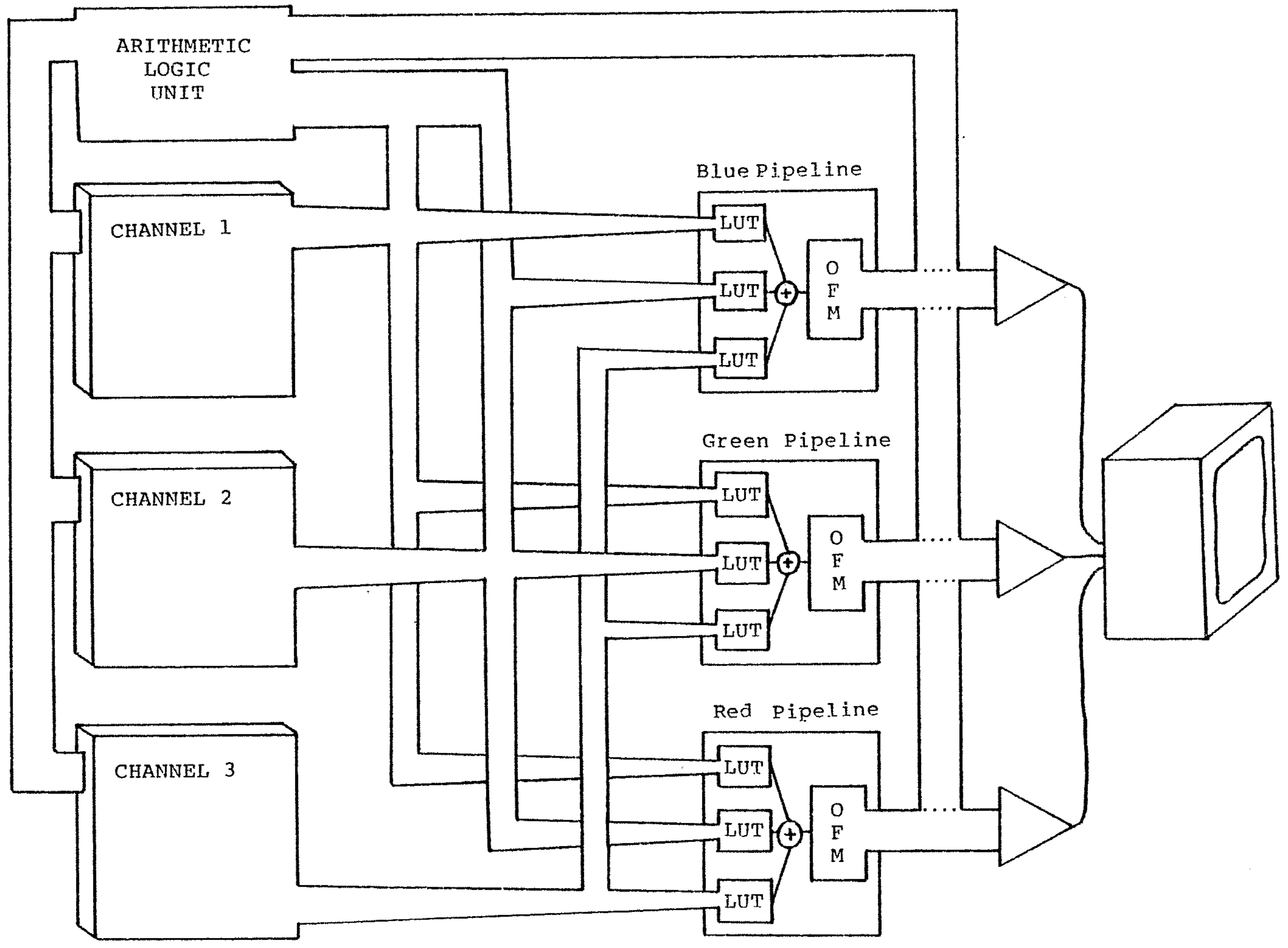

Figure 1. The Stanford Technology Corporation Image Processing Terminal.

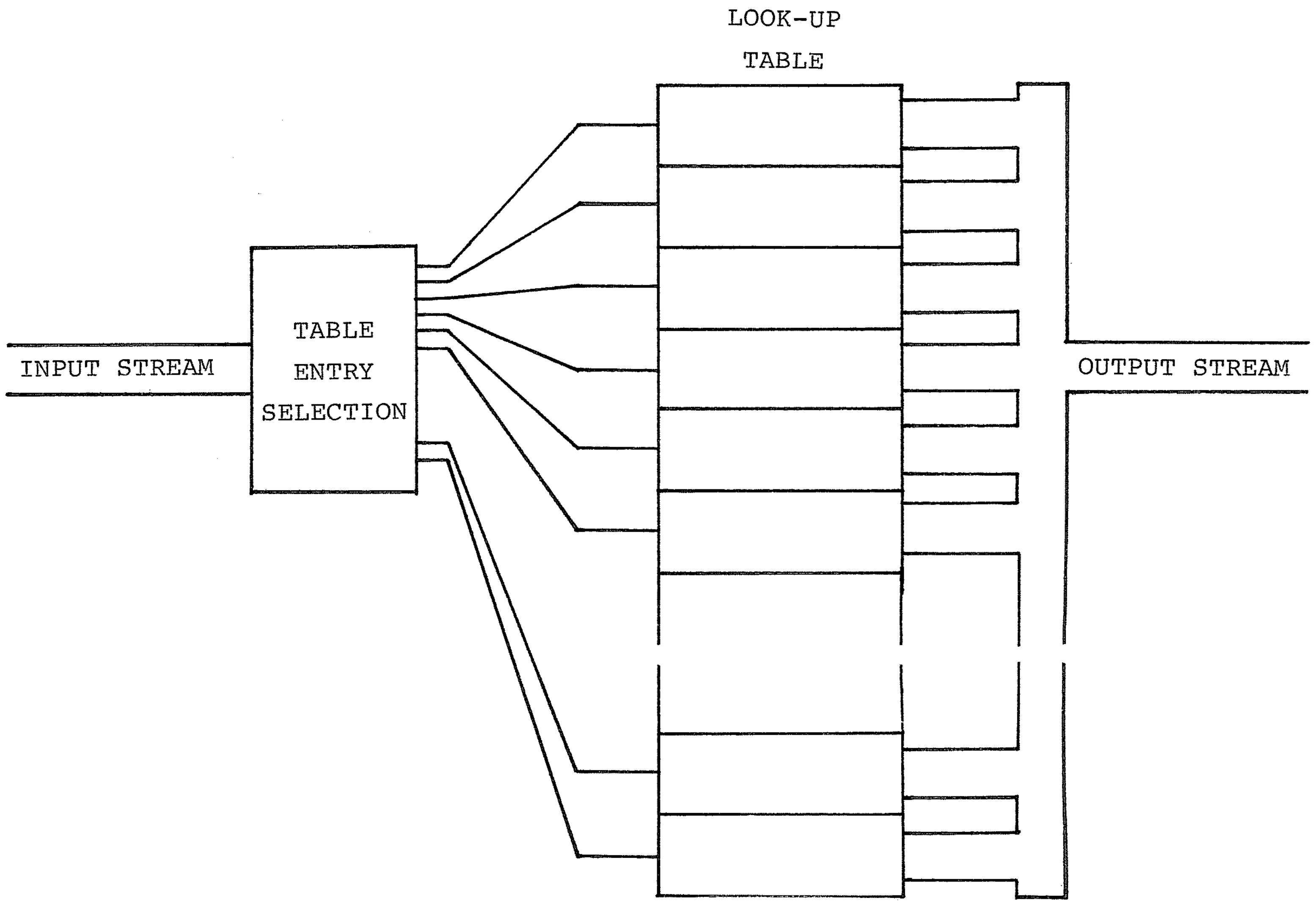

Figure 2.
Data paths shown as wide bars. Enables are shown as
single lines. The values in the input data stream
select an entry in the look-up table. The value stored at this
at this location is then placed in the output stream.

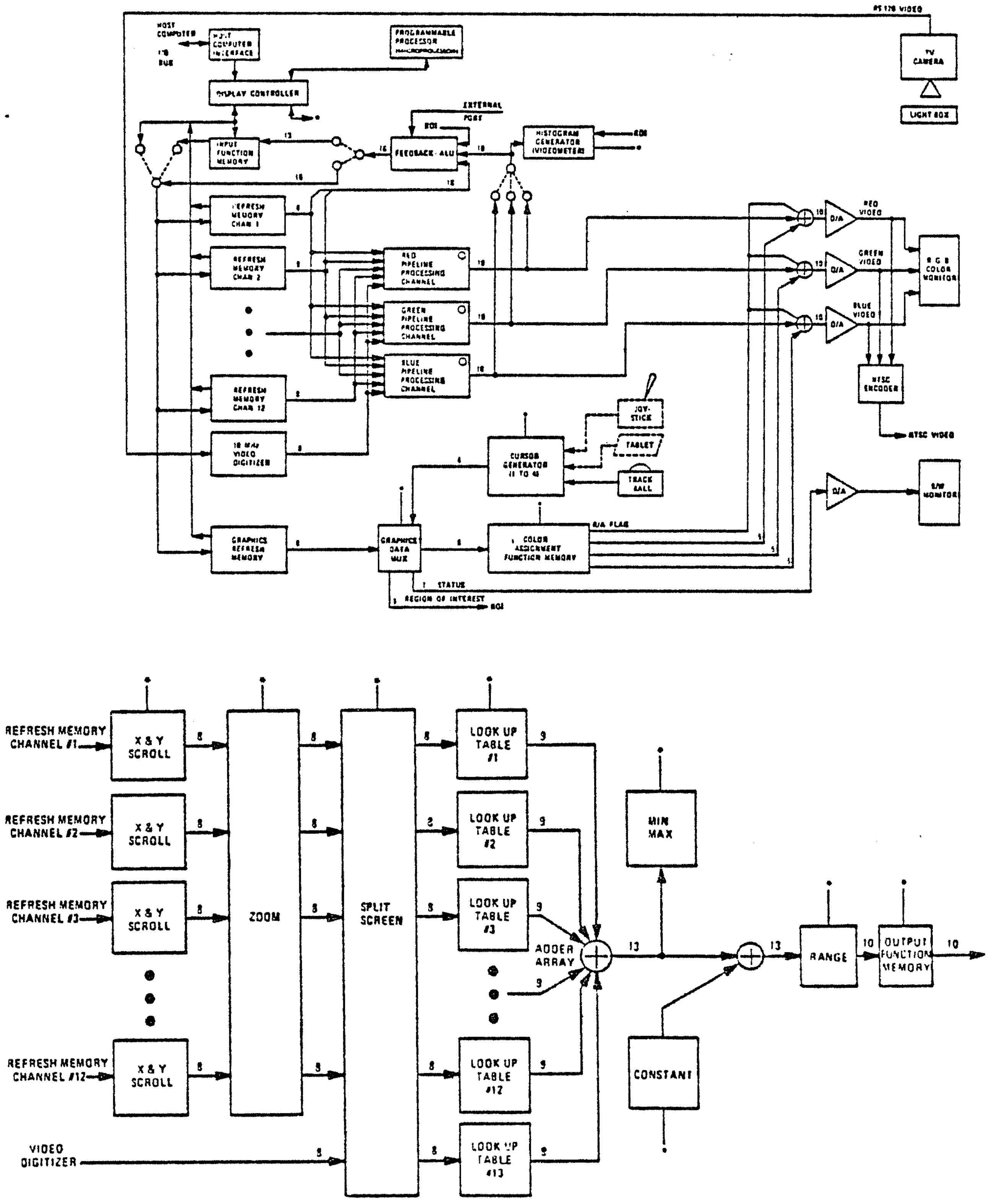

Figure 3. Complete Functional Block Diagram of the STC.