Introduction to Vista Programming

Art Pope, Daniel Ko, David Lowe

for Vista V2.1 — June 1994

Contents

1 Introduction

This document is an introduction to programming in the Vista environment.

Vista includes a library of C-callable routines for performing many operations that are commonly needed in computational vision work. Known as the Vista library, it contains most of Vista's basic routines for manipulating images and other data types. In addition, Vista includes some more specialized libraries. Currently, one specialized library provides support for building interactive image processing applications, and another contains routine for calibrating and applying camera models.

Whereas this document only provides an introduction to two of the libraries, complete documentation for all of Vista exists in the form of manual pages. Section 7Vi man pages, particularly the following ones, provide a good starting point for further exploration:

`Vtutorial(7Vi)` contains a tutorial on using Vista programs

`Vista(7Vi)` provides an overview of Vista

`Vlib(7Vi)` provides an overview of the Vista library

1.1 Overview

This document will develop your understanding of Vista programming through a series of illustrative examples. We start with a simple program for generating test images in order to demonstrate how to parse command line arguments and how to work with the data structure that represents an image.

Then we look at a typical filter program that reads a file containing images, processes the images, and writes the result out as a new file. Many basic Vista programs are of this form so that they can be piped together to perform a series of operations. The program is divided into two parts: a main routine that handles command line options and file I/O, and a routine that processes an individual image. Vista programs are often organized this way so that the routine can be placed in the library and used by other programs. One chapter describes the main program, and the subsequent chapter describes the routine that processes individual images.

Vista can assist you in writing not just filter programs, but also interactive ones that run under X Windows. If your application is of a certain, common form, you can use the VX library, which is described in chapter 5. VX allows you to easily build an interactive application that displays an image while accepting input through menus, dialog boxes, and direct manipulation of the image.

The example code used in this document can be found both in the appendices, and on-line in the directory `$VISTA/examples`. (`$VISTA` denotes the top of the Vista source tree, which at this site is `/usr/src/vista`.) The code makes a good starting point for creating your own programs and routines.

We strongly encourage you to contribute your software to the Vista collection so that it can be used by other researchers. Once you are familiar with the contents of this document, look at the *Vista Contributor's Guide* for some additional guidelines on writing Vista software. The guidelines will help you to ensure that the software you contribute integrates smoothly with the rest of Vista.

1.2 Some preliminaries

As a Vista library user you will need to direct the C compiler to header files located in the `/usr/-include` directory. This is usually done with a `-I` option:

```
cc ... -I/usr/include ...
```

At minimum, each C source code file that uses the library must include a general header file:

```
#include <vista/Vlib.h>
```

and it may have to include others, too, depending on what library features are used. A routine's manual page can be relied upon to tell you what additional header files are needed when using that routine.

When linking, include at least the Vista and math libraries:

```
ld ... -lvista -lm ...
```

plus any other libraries (such as the VX library and various X Window System libraries) mentioned on specific manual pages.

Vista's libraries are written in ANSI C, but in such a way that they can be used from either an ANSI C program or one written in the older "Kernighan and Ritchie" (K&R) dialect. An ANSI C compiler such as *gcc* will find function prototypes in a library's header files, allowing the compiler to check arguments to function calls. We find this to be a real win. For those using a K&R compiler, however, the example code in this guide is written in the K&R dialect; it should work with either a K&R or an ANSI C compiler.

Symbols defined by Vista's libraries follow a simple naming convention: all type names, macro names, and names of externally visible functions and data structures begin with `V`. This makes them easy to recognize, and also, one hopes, unique. (Some external names not meant for you to use directly begin with `V_` or `VX_`, to help ensure their uniqueness while distinguishing them as "private".)

2 A Simple Program

The first example we will look at is a simple program, *ramp*, that generates a square image containing a ramp of intensity values. The program takes a command line option specifying the size of the desired image, and another option specifying the file to which the new image should be written. For example,

```
ramp -size 128 -out image.v
```

produces a 128×128 image, writing it to the file `image.v`. If the `-size` option is omitted it defaults to 256 rows and columns, and if the `-out` option is omitted, the file is written to the standard output stream.

This example will demonstrate the basics of parsing command line options, error handling, and the data structure for representing an image. The entire text of the program is in appendix A.1; here we will examine it in pieces.

2.1 Parsing command line options

The first piece parses command line options:

```
#include <vista/Vlib.h>
#include <vista/mu.h>
#include <vista/option.h>
#include <stdio.h>
```

```
int main (argc, argv)
    int argc;
    char *argv[];
{
    static VShort size = 256;
    static VOptionDescRec options[] = {
        { "size", VShortRepn, 1, (VPointer) & size,
              VOptionalOpt, NULL, "Number of image rows and columns" }
    };

    FILE *out_file;

    VParseFilterCmd (VNumber (options), options, argc, argv,
                     NULL, & out_file);
    ...
}
```

The Vista library includes routines for parsing command line options and for reporting information on what options a program accepts. There are advantages to using these routines: they make it easy for the programmer to parse options, and they provide consistency for users across various programs.

The options a program accepts are described by a table, which is passed to a library routine along with the command line arguments to be parsed. In *ramp*, the table is called `options` and it contains an entry for a single option, `-size`. (Why there isn't also an entry for `-out` will be explained shortly.) The `-size` entry specifies the keyword that signals the presence of the option on the command line (`"size"`), the kind of value that must accompany the option (a short integer), the number of values accompanying it (one), a location to store the value(s) parsed (the `size` variable), whether the option is optional or must be present (it's optional), and a descriptive phrase to print should the user ask for help on how to use the program. Note that the `size` variable has been initialized with the option's default value (256).

(The `options` table is declared static so that it can be initialized at compile time; the `size` variable is declared static so that its address can be used in the initializer for the options table. Also, a header file, *<vista/option.h>*, is included to define the type of an options table entry, `VOptionDescRec`.)

To parse command line options, many Vista programs use the `VParseFilterCmd` routine. It performs three common duties: parsing command line options, printing command usage information, and opening input and/or output files. Its prototype is:

```
void VParseFilterCmd (int noptions, VOptionDescRec options[],
                      int argc, char *argv[],
                      FILE **input, FILE **output);
```

The first two arguments supply the options table, specifying both the number of entries in it and the location of its first entry. It is common practice to use the `VNumber` macro, defined in the *<vista/mu.h>* header file, to supply the number of entries in a fixed-sized table — the macro works by simply dividing the size of the entire table by the size of one table entry.

The next two arguments supply the command line arguments to be parsed, and the last two concern the opening of input and output files. Since *ramp* has no input file, we pass `NULL` for the `input` argument. By passing a non-`NULL` pointer for the output argument, however, we tell `VParseFilterCmd` that an output file is needed. `VParseFilterCmd` will allow that file to be identified on the command line in any of three ways:

1. with an `-out` option (e.g., `ramp -out image.v`);
2. by omitting the `-out` option and simply mentioning the name on the command line (e.g., `ramp image.v`); or
3. by directing the standard output stream (e.g., `ramp >image.v`).

However it finds the output file, `VParseFilterCmd` opens it and returns its `FILE` pointer via the `output` argument. (Note that using this technique, you can't simply parse and return the file-name without opening the file. In cases where you want to do your own file manipulation, use an explicit `"in"` or `"out"` string entry in the options table, and correspondingly use `NULL` in calling `VParseFilterCmd`.)

There are two conditions under which `VParseFilterCmd` will not return but instead exit the program with a status of 1 (indicating an error). It will exit after printing an error message if it detects an error among the command line arguments — e.g., a named file cannot be opened or a keyword isn't recognized. It will also exit, while printing a summary of command line options, if it sees a `-help` option.

2.2 Numeric data types

You will have already seen the use of the type `VShort` to declare one variable, `size`. It is one of several numeric data types defined by the Vista library. Each of the types has some minimum precision, which it is guaranteed to have regardless of machine architecture. Here is the complete set of numeric types:

`VBit`	represents an unsigned integer in the range [0,1]
`VUByte`	represents an unsigned integer in the range [0,255]
`VSByte`	represents a signed integer in the range [-128,127]
`VShort`	represents a signed integer in at least 16 bits
`VLong`	represents a signed integer in at least 32 bits
`VFloat`	represents a floating point number in at least 32 bits
`VDouble`	represents a floating point number in at least 64 bits

Associated with each type is a constant that can be used to refer to it. The constants are values of the `VRepnKind` enumeration, which is defined in *<vista/Vlib.h>* like this:

```
typedef {
   ...,
   VBitRepn, VUByteRepn, VSByteRepn, VShortRepn, VLongRepn,
   VFloatRepn, VDoubleRepn,
   ...
} VRepnKind;
```

The numeric types are used whenever it is necessary to ensure that something will have some minimum precision — as when, for example, an image is declared to have `VLong` pixels so that each pixel will be assured of having at least 32 bits. They are also used whenever it is necessary to refer to one of several possible types — as when specifying what type of value a command line option should have. It is for this second reason that `VShort` and `VShortRepn` are used for the `-size` option. We're not too concerned about the precision of the `-size` value (only small integers are expected), but the option table entry requires that its type be specified and so we have chosen `VShort` for it.

2.3 Error reporting

The Vista library provides a uniform way of reporting errors, both fatal and otherwise. For a fatal error, you can call `VError` with a message:

```
if (size <= 0)
    VError ("Size must be positive; %d isn't", (int) size);
```

It takes arguments in the same way `printf` does, as a format string followed by the values to be formatted. After printing the message, `VError` exits the program with the status code 1.

If the error is a non-fatal one, you would call `VWarning` instead:

```
if (size <= 0) {
    VWarning ("Size must be positive; using 256 instead");
    size = 256;
}
```

The error and warning messages produced by these routines automatically include the program's name, which `VParseFilterCmd` will have noted from the first command line argument.

2.4 Images

After parsing and checking its command line options, the *ramp* program creates an image of the appropriate size in memory. A Vista image contains a three-dimensional array of pixel values (plus some other stuff you needn't worry about right now). Its pixels are all of the same type, which may be any numeric type from `VBit` to `VDouble`.

The three pixel dimensions are called *band*, *row*, and *column*. The band dimension can serve a variety of purposes, including representing such things as the consecutive frames of a motion sequence, the left and right halves of a stereo pair, the color channels of an RGB image, or even some combination of these. (The *ramp* program will create an image with just a single band representing gray-scale intensity.) The remaining dimensions, row and column, index pixels by their vertical and horizontal coordinates. Vista's convention is to number rows and columns from the upper left pixel, which is at row 0, column 0.

To work with images we must include the header file *<vista/VImage.h>*. It defines the type, `VImage`, by which images are represented in memory. (A `VImage` is a pointer to dynamically-allocated data structure.) We then create an image by calling `VCreateImage`:

```
#include <vista/VImage.h>

VImage image;

image = VCreateImage (1, (int) size, (int) size, VUByteRepn);
if (! image)
    exit (1);
```

This creates the image with one band, `size` rows, `size` columns, and `VUByte` (unsigned 8-bit) pixels. Like other library routines, it follows a convention of returning (a pointer to) the resulting object if successful, and returning `NULL` otherwise. This convention dictates that if the routine fails, it will issue its own warning message describing the error. The caller need not issue another message. In the case of the *ramp* program, any error in creating the image is fatal so *ramp* simply exits.

The program then iterates over image pixels, filling them with appropriate values. The library provides a few different ways of accessing and iterating over image pixels. As we will see later in section 4.3, **Computing destination pixel values**, one way is to set a pointer to the first pixel in the image and repeatedly increment it. Here we use a different method, indexing each pixel by its band, row, and column numbers:

```
int row, column, value;

for (row = 0; row < size; row++) {
    value = (float) row / (size - 1) * 255;
    for (column = 0; column < size; column++)
        VPixel (image, 0, row, column, VUByte) = value;
}
```

The intensity ramp filling the image runs from zero (black) at the top of the image to 255 (white) at the bottom. At each row we compute the pixel value, `value`, needed to fill that row. The `VPixel` macro is used to access the pixel at band 0, row `row`, and column `column` of `image`. The macro requires an argument specifying the type of pixel the image has — in this case, `VUByte`.

2.5 Writing images

The program then writes the image to the file opened previously by `VParseFilterCmd`:

```
if (! VWriteImages (out_file, NULL, 1, & image))
    exit (1);
```

It uses `VWriteImages`, whose prototype is:

```
VBoolean VWriteImages (FILE *file, VAttrList attributes,
                       int nimages, VImage images[]);
```

(For now, don't worry about the *attributes* argument, which denotes an attribute list. Attribute lists are described below in section 3.1.)

`VWriteImages` writes out a series of images as a single Vista data file, returning `TRUE` if successful. On failure, it issues its own warning message (e.g., "Write to stream failed") and returns `FALSE`.

2.6 For more information

More information on the topics, data structures, and routines introduced in this chapter can be found in the following manual pages:

`Voption(3Vi)`	table describing command line options
`VParseFilterCmd(3Vi)`	parse command line options and filenames
`Vlib(7Vi)`	numeric data types
`VError(3Vi)`	report a fatal error
`VWarning(3Vi)`	report a non-fatal error
`VImage(3Vi)`	image representation, and a summary of macros and routines for working with images
`VCreateImage(3Vi)`	allocate memory for an image
`VWriteImages(3Vi)`	write a series of images to a stream

3 Implementing a UNIX Filter Program

Like many Vista programs, the one described in this chapter is designed to be used as a filter program. It reads a data file, performs an operation on the objects read, and writes the results out as a new data file. This particular program, *binarize*, thresholds each input image to produce an output image with `VBit` pixels — an output pixel is 1 only if the corresponding input pixel reaches a specified threshold. It might typically be used to flag all non-negative pixels:

```
binarize -threshold 0 < input.v > output.v
```

3.1 Attribute lists

A Vista data file can contain a variety of different sorts of objects, images being just one sort. A file's contents are organized as a list of attributes, each comprising both an attribute name and an attribute value. The attribute name is an alphanumeric string, such as *image1* or *pixel_aspect_ratio*. The attribute value may have any of several forms depending on whether the value is a number, a string, a keyword, a nested list of attributes, an image, or some other object. For example, a file containing two attributes might look something like this:

```
V-data 2 {
    face: image {
        data: 0
        length: 65536
        nbands: 1
        nrows: 256
        ncolumns: 256
        repn: ubyte
        pixel_aspect_ratio: 1.25
        name: "Jean"
    }
    comment: "Taken March 30, 1993"
}
...
```

The first attribute's name is `face`. Its value is an image, represented in the file by `image { ... }`. Between those braces lies another attribute list with elements describing the properties of the image — where its pixel values are to be found within the file, how many bands, rows and columns the image has, what type of pixels it has, and so on. The second top-level attribute has the name `comment` and a value that is a string.

The usual behavior for a Vista filter program is to pass unchanged any attributes in the input file that aren't relevant to the program. Given the above file, for example, the *binarize* program should replace the `face` image with a new, thresholded one while passing the `comment` attribute unchanged. To achieve this behavior a typical filter program will do this sequence of steps:

1. parse command line options
2. open input and output files
3. read the contents of the input file into memory where it is represented as a linked list of attributes
4. step through the list, processing and replacing those attribute values that are relevant
5. write the list to the output file

A list of attributes is represented in memory by a `VAttrList` type, which is a pointer to a dynamically-allocated, doubly-linked list. Each element of the list includes an attribute name, an indication of what form the attribute value takes, and the attribute value itself. Like its counterpart in a data file, an attribute value in memory can have any of several forms. They include:

- *a character string.* Numbers (e.g., 1.25 in the example above), keywords (e.g., `ubyte`), and strings (e.g., `"Jean"`) are all represented in memory as character strings. As they are accessed within a program they may be converted to and from other forms, such as binary integers and floating point numbers.
- *an attribute list.* Because an attribute's value can itself be another attribute list, attributes can have values that are arbitrarily complex.
- *an image.* The image is represented by a `VImage` data structure.

Another type is used to represent a particular position within an attribute list. This type, `VAttrListPosn`, is a structure. When stepping through an attribute list to examine each of its attributes individually, a `VAttrListPosn` variable is used to record the current position within the list.

The Vista library includes routines and macros for creating, copying, and destroying attribute lists; for accessing individual attributes and their components; for reading attribute lists from data files; and for writing attribute lists to data files. We will see a few of these routines and macros in this chapter. For a complete list of them, see `Vattribute(3Vi)`.

3.2 Parsing command line options

Appendix A.2 contains a full listing of the *binarize* program. The portion that parses command line options is much like the *ramp* code we saw earlier:

```
#include <vista/Vlib.h>
#include <vista/mu.h>
#include <vista/option.h>
#include <stdio.h>

int main (argc, argv)
    int argc;
    char *argv[];
{
    static VDouble threshold = 0.0;
    static VOptionDescRec options[] = {
        { "threshold", VDoubleRepn, 1, (VPointer) & threshold,
              VOptionalOpt, NULL, "Value at which pixel is set to 1" }
    };

    FILE *in_file, *out_file;

    VParseFilterCmd (VNumber (options), options, argc, argv,
                     & in_file, & out_file);
    ...
}
```

The `-threshold` option takes a floating point value, which defaults to zero if the option isn't present. In this program, `VParseFilterCmd` will open both input and output files. The input file can be specified by an `-in` option, simply named on the command line prior to any output file name, or allowed to default to the standard input stream. Similarly, the output file can be specified by `-out`, named on the command line, or allowed to default to the standard output stream. `VParseFilterCmd` returns if the command line arguments have been found acceptable, after opening both input and output files.

3.3 Reading an attribute list

The next step is to read the contents of the input file. We use `VReadFile`, whose prototype is supplied in <*vista/file.h*>:

```
#include <vista/file.h>

VAttrList list;

list = VReadFile (in_file, NULL);
if (! list)
    exit (1);
```

The `VReadFile` routine reads the contents of a Vista data file, returning it as an attribute list. On failure it returns `NULL` instead. (The second argument to `VReadFile` can be used to provide a filtering routine that gets to choose those attributes that will be read fully — it can almost always be `NULL`, as it is here, allowing all attributes to be read. For details see `VReadFile(3Vi)`.)

3.4 Stepping through an attribute list

We then step through the attribute list while processing any images it contains:

```
VAttrListPosn posn;
VImage src;

for (VFirstAttr (list, & posn); VAttrExists (& posn); VNextAttr (& posn)) {
    if (VGetAttrRepn (& posn) != VImageRepn)
        continue;
    VGetAttrValue (& posn, NULL, VImageRepn, & src);
    ...
}
```

This fragment of code demonstrates several macros and routines used to work with attribute lists. The `for` statement steps through the attribute list: `VFirstAttr` sets `posn` to refer to the first attribute in `list`; `VAttrExists` tests whether `posn` still lies within `list`; and `VNextAttr` moves `posn` forward to the next list element.

With `posn` identifying an attribute list element, `VGetAttrRepn` is used to determine what sort of value that attribute has. What `VGetAttrRepn` returns is one of the integer codes defined for the enumerated type `VRepnKind`. In section 2.2, **Numeric data types**, we saw a partial list of those codes. Here's the complete list:

```
typedef enum {
    VUnknownRepn,

    /* Integer numbers: */
    VBitRepn,              /* 1-bit integer, [0, 1] */
    VUByteRepn,            /* 8-bit integer, [0, 255] */
```

```
    VSByteRepn,             /* 8-bit integer, [-128, 127] */
    VShortRepn,             /* 16-bit integer, [-32768, 32767] */
    VLongRepn,              /* 32-bit integer, [-2**31, 2**31-1] */

    /* Floating point numbers: */
    VFloatRepn,             /* 32-bit IEEE floating point */
    VDoubleRepn,            /* 64-bit IEEE floating point */

    /* Miscellaneous representations: */
    VAttrListRepn,          /* attribute list */
    VBooleanRepn,           /* TRUE or FALSE */
    VBundleRepn,            /* object of named type */
    VListRepn,              /* list of opaque objects */
    VPointerRepn,           /* pointer to opaque object */
    VStringRepn,            /* null-terminated string */

    /* Standard object types: */
    VEdgesRepn,             /* edge set */
    VImageRepn,             /* image */
    VNRepnKinds             /* number of predefined types */
} VRepnKind;
```

Of these codes, the only one we're concerned with at the moment is `VImageRepn`, which indicates that the attribute value is an image. If `VGetAttrRepn` returns something other than `VImageRepn`, we continue to the next attribute. Otherwise, we have an image attribute value, which we fetch into the variable `src` using `VGetAttrValue`.

3.5 Performing an operation on an image

A Vista library routine that processes one image to produce another usually takes three or more arguments. The first three look like this:

`VImage VProcessImage (VImage` *src*`, VImage` *dest*`, VBand` *band*`, ...)`

The first argument specifies the *source* image, which is the one to be processed. The second specifies the *destination* image, which is where the result will be stored. The third specifies which band or bands of the source image are to be processed.

You generally have three choices for how the destination image is provided to such a routine:

1. You can have the routine (in this case, `VProcessImage`) create a destination image. If you pass `NULL` for *dest*, the routine will create and return a destination image that is appropriate for the source image and for the type of operation being performed on it:

   ```
   result = VProcessImage (src, NULL, band, ...);
   ```

2. You can supply a destination image, provided it has the correct properties. Usually this means it must have a particular size and pixel representation:

   ```
   dest = VCreateImage (1, 256, 256, VBitRepn);
   VProcessImage (src, dest, band, ...);
   ```

3. You can specify the same image as both the source and the destination, in which case the source pixel values will be replaced by destination pixel values. This choice is only available for operations where the source and destination images have the same properties:

```
VProcessImage (src, src, band, ...);
```

Regardless of how the destination image is provided, a typical routine will return that image if it is successful, and it will return `NULL` otherwise. Moreover, if `NULL` is returned, the routine will already have called `VWarning` to issue a message describing the problem.

The *band* argument may specify, by band number, a particular band of the source image to be processed. Usually in this case the destination image will have a single band. Or the *band* argument may be the constant `VAllBands` (defined as -1) to indicate that all bands of the source image are to be processed. In the `VAllBands` case, the destination image usually has the same number of bands as the source image.

In the *binarize* program we use a routine called `BinarizeImage` to threshold the image while allowing that routine to create an appropriate destination image. `BinarizeImage` is not a Vista library routine, but rather one presented as an example in chapter 4. Nevertheless it follows the library's usual calling convention. From *binarize* it's called by this code fragment:

```
VImage result;

result = BinarizeImage (src, NULL, VAllBands, threshold);
if (! result)
    exit (1);
```

This will threshold each band of `src` using the value `threshold`. The new image returned by `BinarizeImage` has the same numbers of bands, rows, and columns as `src`, but it has `VBit` pixels regardless of what kind of pixels `src` may have. The new image is stored in `result`.

This new image is then substituted for the original one in the attribute list, and the original image is discarded:

```
VSetAttrValue (& posn, NULL, VImageRepn, result);
VDestroyImage (src);
```

3.6 Writing an attribute list

Once all attributes have been examined and any images have been replaced by their thresholded versions, the attribute list is written out in the form of a data file:

```
if (! VWriteFile (out_file, list))
    exit (1);
```

And that completes the *binarize* program.

3.7 For more information

More information on the topics, data structures, and routines introduced in this chapter can be found in the following manual pages:

`Vfile(5Vi)`	format of Vista data files
`Vattribute(3Vi)`	representation of attributes in memory, and a summary of macros and routines for working with them
`VReadFile(3Vi)`	read an attribute list from a data file
`VWriteFile(3Vi)`	write an attribute list to a data file
`VGetAttrValue(3Vi)`	fetch an attribute's value
`VSetAttrValue(5Vi)`	set an attribute's value
`VImage(3Vi)`	calling conventions for image processing routines

4 Implementing an Image Processing Routine

In the last chapter we looked at a UNIX filter program that reads images from a data file, thresholds them, and writes out the modified images. The actual thresholding is done by a routine, `BinarizeImage`, that has this prototype:

`VImage BinarizeImage (VImage` *src*`, VImage` *dest*`, VBand` *band*`, double` *threshold*`);`

`BinarizeImage` accepts an image of any type and a threshold; it produces a thresholded image of the same size having `VBit` pixels. It follows the same conventions for argument passing as other image processing routines in the Vista library:

- the *src* argument specifies the image to be processed (the *source* image)
- the *dest* argument may specify a *destination* image, or it may be `NULL` to indicate that a new destination image should be created by the routine
- the *band* argument may specify a particular band of the source image to be processed, or may be the constant `VAllBands` to indicate that all bands of the source image should be processed

If successful, `BinarizeImage` returns the destination image, even in cases where it was supplied by the caller. On encountering an error that the caller could conceivably recover from, `BinarizeImage` issues a warning message and returns `NULL`. For a fatal error, it would call `VError(3Vi)` to issue a message and terminate the program. Any message produced by `BinarizeImage` includes the routine's name as a prefix to identify the source of the message.

In this chapter you'll see how `BinarizeImage` is implemented, and from it learn how to implement other image processing routines that follow the same conventions. Most routines, including `BinarizeImage`, have this general structure:

1. Validate arguments.
2. If a destination image was supplied, ensure that it has the appropriate properties; otherwise, create one.
3. Compute the value of each destination pixel.
4. Return the destination image.

In the following sections each step is covered separately. A complete listing of `BinarizeImage` is in appendix A.3.

4.1 Validating arguments

Routines that, like `BinarizeImage`, have a `band` argument must check that it specifies a band lying within the source image. Often they also need to establish a pointer and counter for iterating over the source pixels in the band or bands identified by their `src` and `band` arguments. One library routine, `VSelectBand`, can be used to do both:

```
VImage BinarizeImage (src, dest, band, threshold)
    VImage src;
    VImage dest;
    VBand band;
    double threshold;
{
    int npixels;
    VPointer src_pixels;

    if (! VSelectBand ("BinarizeImage", src, band, & npixels, & src_pixels))
        return NULL;
    ...
}
```

`VSelectBand` will issue a warning message and return `FALSE` if `band` is neither `VAllBands` nor the number of a band within `src`. If `band` is valid, however, it returns `TRUE` while setting `npixels` and `src_pixels` to the number and address of the source pixels identified by `src` and `band`. The routine's name, `"BinarizeImage"`, is passed to `VSelectBand` to be included in any warning message that must be issued.

It would be silly to threshold an image against a value that the source pixels cannot possibly attain — for example, thresholding 8-bit unsigned (`VUByte`) pixels against the value 1000. As a courtesy, `BinarizeImage` issues a warning but proceeds anyway if the threshold lies outside the range of possible source pixel values:

```
if (threshold < VPixelMinValue (src) || threshold > VPixelMaxValue (src))
    VWarning (
      "BinarizeImage: Threshold (%g) is outside source pixel range [%g,%g]",
              threshold, VPixelMinValue (src), VPixelMaxValue (src));
```

The `VMinPixelValue` and `VMaxPixelValue` macros return the minimum and maximum values permitted by `src`'s pixel representation.

4.2 Establishing a destination image

Usually the library routine `VSelectDestImage` can be used to establish a destination image, either creating it if it doesn't exist or checking its properties if it does. Here is how `BinarizeImage` uses `VSelectDestImage`:

```
int nbands;

nbands = (band == VAllBands) ? VImageNBands (src) : 1;
dest = VSelectDestImage   ("BinarizeImage", dest, nbands,
                           VImageNRows (src), VImageNColumns (src),
                           VBitRepn);
if (! dest)
    return NULL;
```

The last four arguments to `VSelectDestImage` are the properties the destination image should have: the numbers of bands, rows, and columns, and the pixel representation. We choose the number of bands, `nbands`, to be 1 if `band` specifies a single source band; or we choose it to be the number of bands in the source image if `band` is `VAllBands`. The destination image will have the same numbers of rows and columns as the source image, and `VBit` pixels. Also passed to `VSelectDestImage` is a routine name to be included in error messages, and the `dest` argument as it was supplied to `BinarizeImage`. `VSelectDestImage` will return one of the following:

- a newly created image, if no destination image was supplied (i.e., if `dest` is `NULL`); otherwise
- the destination image supplied to `BinarizeImage`, provided it has the correct properties; otherwise
- `NULL`, while issuing a warning message about the inappropriate destination image.

4.3 Computing destination pixel values

One issue to consider at this point is how to iterate over the source and destination pixels in order to compute each destination pixel value from its corresponding source pixel value. Because pixels are stored contiguously within an image, you can set a pointer to the first pixel to be processed and simply increment it for each subsequent pixel. This is in fact the approach taken by `BinarizeImage`.

Another approach is to use nested loops that iterate over the band, row, and column dimensions — this is appropriate when the treatment of a pixel must depend on where it lies in the image (e.g., border versus interior). Within the innermost loop, pixels are accessed with macros like `VPixel`, which computes a pixel address from band, row, and columns number using indices that are recorded with each image. This approach is illustrated by the fragment of *ramp* program code described in section 2.4, bf Images.

A second issue to consider is how to handle various types of pixel representation. Recall that each image pixel can have any numeric type, from `VBit` through `VDouble`. Tasks like iterating over pixels and doing arithmetic with pixel values can be somewhat complicated by the need to handle various pixel types. In designing a routine like `BinarizeImage`, you need to decide what pixel types will be accommodated and how. There are several approaches for you to consider:

- The routine could be restricted to working only with a particular pixel type. In the case of `BinarizeImage`, for example, we could perhaps restrict it to `VUByte` pixels — the caller would then be responsible for ensuring that any source image was of that form. This approach may be quite appropriate in some cases, but in others it creates an extra burden for the caller.
- The routine could accept any pixel type, but initially convert the source image to a single, preferred type. `BinarizeImage`, for example, might convert its source image to `VDouble` pixels before thresholding them. Library routines like `VConvertImageRange` are available for converting images. This approach is simple to program although it uses more memory and result in more pixel accesses.
- The routine can isolate pixel type dependencies to the places where pixels are actually accessed. It can use `VGetPixel` to fetch a pixel value as a `VDouble` regardless of its actual type (as, for example, the *show* program's `SelectCallback` routine does on page 24) and `VSetPixel` to store a pixel value supplied as a `VDouble`. When only occasional pixels are accessed (as by `SelectCallback`) this approach is appropriate; note that `VGetPixel` and `VSetPixel` are not macros, however, so using them entails the overhead of additional function calls.
- The routine can handle the various pixel types by cases, usually by switching to separate code for each pixel type. Although this approach is more difficult to program than the others, it is frequently used throughout the library because of its efficiency. `BinarizeImage` illustrates this approach, as we shall see.

`BinarizeImage` iterates over the source and destination images using a macro parameterized by source pixel type. The macro, `Binarize`, assumes that these variables have been initialized: `src_pixels` points to the first source pixel to be processed, `dest_pixels` points to the first destination pixel, and `npixels` is the number of pixels to process. The macro is instantiated once for each type of source pixel, to create a version of the pixel processing loop customized for that type:

```
int i;
VBit *dest_pixels;

/* Locate the block of destination image pixels: */
dest_pixels = & VPixel (dest, 0, 0, 0, VBit);

/* Define a macro, parameterized by source pixel type,
   that can be instantiated for each possible type: */
#define Binarize(type)                                          \
    {                                                           \
        type *src_pp = (type *) src_pixels;                     \
        type t = threshold;                                     \
                                                                \
        for (i = 0; i < npixels; i++)                           \
            *dest_pixels++ = (*src_pp++ >= t);                  \
    }

/* Instantiate the macro once for each source pixel type: */
switch (VPixelRepn (src)) {
case VBitRepn:      Binarize (VBit);        break;
case VUByteRepn:    Binarize (VUByte);      break;
case VSByteRepn:    Binarize (VSByte);      break;
case VShortRepn:    Binarize (VShort);      break;
case VLongRepn:     Binarize (VLong);       break;
case VFloatRepn:    Binarize (VFloat);      break;
case VDoubleRepn:   Binarize (VDouble);     break;
}

#undef Binarize
```

Using its `type` parameter, the macro declares local variables based on the type of source pixel to be processed. One variable, `src_pp`, is a pointer for accessing and iterating over the pixels. Another, `t`, is a copy of the routine's `threshold` argument, coerced to the source pixel type (to avoid the repeated coercions that may be computed if `threshold` were used inside the loop instead).

4.4 Returning the destination image

One step remains before the destination image can be returned. Prior to describing that step, we must say more about what an image comprises. Besides its array of pixel values, an image includes an attribute list recording various properties of the image. Attributes may be present to describe how the image's bands are to be interpreted (as RGB color channels, say, or the left and right halves of a stereo pair), what shape a pixel has (square, or rectangular to a specified degree), and how the image was acquired. Certain attributes have standard interpretations within Vista, but any others can be added by Vista users. By convention these attributes are usually preserved by image processing routines — they are copied unmodified from source image to destination image, and from program input to program output. When appropriate, however, a routine may delete, modify, or amend the attributes while copying them from source to destination. For example, a routine that converts an RGB color image to a grey-scale one will omit copying the standard attribute that describes an image as having RGB color channels.

The operation performed by `BinarizeImage` is not one that requires modifying any of the standard image attributes. Thus we simply copy attributes from the source image to the destination image (replacing any attributes the destination image might have had previously):

```
VCopyImageAttrs (src, dest);
return dest;
```

That completes the `BinarizeImage` routine.

4.5 For more information

More information on the topics, data structures, and routines introduced in this chapter can be found in the following manual pages:

`VImage(3Vi)`	calling conventions for image processing routines
`VSelectBand(3Vi)`	select bands of image pixels
`VSelectDestImage(3Vi)`	select a destination for an image operation
`VConvertImageRange(3Vi)`	convert an image's pixel representation
`VGetPixel(3Vi)`	get a pixel value
`VSetPixel(3Vi)`	set a pixel value
`VCopyImageAttrs(3Vi)`	copy image attributes

5 Implementing an X Windows Application

Vista includes a VX library to help you create image processing programs that have graphical user interfaces and run under X Windows. If your program is of a particular form, it can use the VX library to simplify the task of dealing with X Windows. Most of this chapter describes the VX library with the aid of an example program. Later in the chapter we'll describe some ways to go beyond or beneath VX when you need to do something that VX cannot handle for you.

A program built with VX has a main window consisting of three parts. From top to bottom those parts are:

1. A menu bar, containing the menus and menu entries of your choice. VX notifies your program each time the user selects a menu entry.
2. A drawing area, in which an image can be displayed along with overlayed lines and text. Your program can arrange to be notified whenever the user enters mouse clicks or key presses in the drawing area. Zooming and panning of the image are under the user's control, and handled automatically by VX.
3. An optional message area that displays text messages produced by your program. The message area is for output only, but it can be scrolled by the user.

Your program can also have VX pop up dialog boxes to display messages, or to solicit from the user information such as filenames and yes/no answers. Most of the user interface components come from the OSF/Motif toolkit, giving VX-based programs a user interaction style that follows the *OSF/Motif Style Guide*.

We will use a program called *show* to illustrate VX programming. The program displays an image while reporting the value of any pixel clicked upon. The last pixel clicked is highlighted on the image by drawing an outline around it. When started, the program reads its image from a file supplied on the command line. However, at any time a new image can be read and displayed by selecting Open from the File menu and entering a filename in a pop-up dialog box that appears. To terminate the program, the user must select Quit from the File menu. Appendix A.4 lists the entire text of the program.

Before examining the *show* program we must first describe the programming model used by VX, the support that VX provides for the main window's drawing area, and some other preliminaries.

5.1 Event-driven programming

Like most programs that have graphical user interfaces, a VX program is event driven in the sense that it is directed by user actions. Once the program has initialized itself it remains in what is called a *main event loop*. In that loop, the program repeatedly awaits and responds to events initiated by the user. Some of these events are handled entirely by VX and the underlying layers of X Windows software (e.g., an event signalling the need to redraw a window due to a change in its state of visibility). Other events — specifically the selection of a menu entry, a mouse click on the drawing area, or a key press with the pointer over the drawing area — are passed on to routines that you supply. These routines are called *callback functions*.

From our point of view, a VX program will be simply a set of callback functions, each of which has one very specific goal: respond to a particular type of user input. Once the program enters its main event loop, its work is accomplished by a series of callback function invocations that occur under the user's direction. The last invocation is usually associated with a Quit menu entry; its action is to terminate the main event loop by exiting the program.

Sometimes you must have your program present what is called a *modal* dialog box. It's called modal because the user is forced to respond to the dialog box before proceeding to interact with other parts of the program's user interface. VX makes it simple for your program to present a modal dialog box: you simply call a VX routine to pop up the dialog box, and you receive control back from the routine once the user has responded. No callback functions are invoked while the modal dialog box is displayed. VX supports modal dialog boxes that request yes/no answers, text strings, and filenames.

A *modeless* dialog box, on the other hand, can remain up while the user proceeds to interact with other parts of the user interface. Later the user can return to the dialog box — in fact, it remains displayed until the user dismisses it. You call a VX routine to pop up the dialog box and that routine returns immediately. Your program is not notified when the user eventually dismisses the dialog box. VX supports two modeless dialog boxes for presenting text messages: one for short messages without scroll bars, and another for longer messages with scroll bars.

5.2 Drawing area

The drawing area displays a Vista image, plus an image overlay consisting of lines and text. Through VX you can specify the image to display; draw and erase overlay markings; and receive callback notification of mouse or keyboard input directed at the drawing area.

User controls for zooming the image in and out and for controlling the portion of the image currently displayed are provided automatically by VX. By default, the controls are tied to these mouse button events:

Middle Button	Zoom in by a factor of two, centering on the pointer.
Right Button	Zoom out by a factor of two, centering on the pointer.
Ctrl Middle Button	Shift the image to center the pixel under the pointer.

The coordinates your program uses when drawing an overlay of lines or text are row/column coordinates, determined by the coordinate system of the image currently displayed. Similarly, when your callback function is notified of user input it is told the location of that input (i.e., the pointer location) in terms of row/column coordinates. The mapping between those and window coordinates, which varies with changes in image magnification and display area, is conveniently handled for you by the VX library.

5.3 Some more preliminaries

Source code files using the VX library must include this header file:

```
#include <vista/VX.h>
```

When linking a VX program, you must include the VX library, the Vista library, an OSF/Motif widget library, various X Windows libraries, and the math library in this order:

```
ld ...-lvx -lvista -lXm -lXt -lXmu -lXext -lX11 -lm ...
```

Symbols defined by the VX library follow and extend the Vista library's naming convention: all VX type names, macro names, and names of externally visible functions and data structures begin with `VX`.

Like other programs that rely on X Windows, a VX program can be configured to some degree by specifying resources. X provides various ways by which resources can be supplied to a program, as described in `X(1)`. To specify resources you need to know the program's class name, and the hierarchy of widgets that the program creates. The class name is something you, the programmer, choose yourself. The widget hierarchy is documented in `VX(3Vi)`. You may use the *editres* program (which comes with the X11R5 core distribution) to view that hierarchy and edit resource values interactively in any VX application.

5.4 Overview of the *show* program

We are now prepared to look at the *show* program. It consists of five routines:

- a `main` routine, which initializes the program and enters the main event loop
- callback functions for the Open and Quit menu entries:

  ```
  extern void OpenCallback ();
  extern void QuitCallback ();
  ```

- a callback function for mouse clicks on the image:

  ```
  extern void SelectCallback ();
  ```

- a utility routine for reading an image from a file and displaying the image:

  ```
  extern VBoolean DisplayImage ();
  ```

The program has a single global variable, `image`, containing the image currently displayed:

```
VImage image = NULL;
```

5.5 Program initialization

The VX library is initialized by a call to `VXInit`, which must be called before any other VX routine. Besides initializing, it parses command line options pertaining to X Windows (e.g., a `-display` option), and it creates the widgets comprising the program's main window. The X-related command line options that `VXInit` recognizes are standard for programs based on the X Toolkit; they're described in `X(1)`. Here's a portion of *show* containing the call to `VXInit`:

```
int main (argc, argv)
    int argc;
    char *argv[];
{
    /* Default resource settings: */
    static VStringConst resources[] = {
        "*messageAreaNLines: 2",
        "*fileBox.textColumns: 50",
        "*fileBox.pattern: *.v",
        "*imageView.cursor: crosshair",
        NULL
    };

    VXInit ("Show", resources, & argc, argv);
    ...
}
```

The string `"Show"` specifies the program's class name. In a user's resource file (e.g., `$HOME/.Xresources`) that class name will be used to distinguish resources meant for the *show* program. For example, the resource file might contain these lines:

```
Show.width:                300
Show.height:               400
Show.foreground:           black
Show.background:           white
```

In your call to `VXInit` you can supply defaults for resources. Those defaults will be used for resources not set in the user's resource file (`$HOME/.Xresources`) or the application's default resource file (e.g., `/usr/lib/X11/app-defaults/Show`).

The *show* program's default resources are in the array `resources`. The `messageAreaNLines` resource specifies the default number of text lines to be shown in the main window's message area. The two `fileBox` resources are for the file selection dialog box that pops up when `Open` is selected: one specifies the number of columns in the box's text entry fields, and the other specifies the initial pattern for determining what files are listed (see `XmFileSelectionBox(3X)`). The resource for the widget called `imageView`, which forms the main window's drawing area, gives it a crosshair cursor. `NULL` marks the end of the array of default resources.

The command line arguments are passed to `VXInit` as `argc` and `argv` but notice that it is `argc`'s address that is passed, not its value. `VXInit` will parse and remove from `argv` those arguments that pertain to X-related options. On return, any unrecognized arguments remain in `argv`, and `argc` is set to the number of such arguments. This sets things up nicely for the program's next step, which is to parse any Vista-related options (e.g., `-in` and `-help`):

```
FILE *in_file;

VParseFilterCmd (0, NULL, argc, argv, & in_file, NULL);
```

Since *show* requires an input file but no output file, we pass only an input `FILE` pointer to `VParseFilterCmd`. To open that file `VParseFilterCmd` will look for an `-in` option, a filename appearing on the command line, or a redirection of the standard input stream. On return, in_file will have been set to the opened input file. We read an image from that file and display it using a utility routine, DisplayImage, that is presented below in section 5.9, **Displaying an image**:

```
if (! DisplayImage (in_file))
    exit (1);        /* no image could be found */
fclose (in_file);
```

We then call VXAddMenu to create the program's File menu:

```
VXAddMenu ("File",
           "Open...", "Ctrl<Key>O", OpenCallback, NULL,
           "Quit", "Ctrl<Key>Q", QuitCallback, NULL,
           NULL);
```

VXAddMenu accepts a variable number of arguments. The first specifies the menu's name as it should appear in the menu bar ("File"). That's followed by sets of four arguments, each specifying a menu entry, ordered as the entries should appear in the menu. NULL marks the end of the menu entries. The four arguments for a single menu entry specify the name of the entry as it should appear in the menu (e.g., "Open..."), an optional *accelerator* that can be used to select the menu entry from the keyboard (e.g., "Ctrl<Key>O"), a callback function to invoke when the user selects the menu entry (e.g., OpenCallback), and an argument to pass to the callback function (in this case, NULL for both entries).

The *show* program only has a single menu. If it had more, however, we would simply define each one with an additional call to VXAddMenu.

Another call is made to register a callback function for mouse clicks in the drawing area:

```
VXAddInputCallback (VXIbuttonPress, SelectCallback, NULL);
```

Through VXAddInputCallback's first argument we can ask to be notified of button presses or releases, pointer motion, or key presses. (You can register any number of different callback functions for each of these different types of input event.) Here we've asked for notification of button presses and we've supplied a callback function, SelectCallback, to be invoked when one occurs. We've also supplied an argument, NULL, to be passed to our callback function.

This completes the program's initialization. To now perform its main event loop, we call VXAppMainLoop:

```
VXAppMainLoop ();
```

5.6 Menu item callback functions

During the program's initialization phase we registered callback functions for each of its two menu entries, Open and Quit. Here is the entire callback function for Open:

```
void OpenCallback (client_data)
    VPointer client_data;
{
    VStringConst filename;
    FILE *in_file;

    /* Pop up a dialog box to get the new file's name: */
    filename = VXPopupFileBox ("Open Image File");
```

```
        if (! filename)
            return;         /* Cancel button was clicked */

        /* Open it and display the image it contains: */
        in_file = fopen (filename, "r");
        if (in_file) {
            DisplayImage (in_file);
            fclose (in_file);
        } else VWarning ("Unable to open %s", filename);
        VFree ((VPointer) filename);
    }
```

This routine uses two types that you may not have encountered previously:

- `VPointer` is defined by the Vista library as a generic pointer to an unspecified type — usually `void *`. It is used here as the type of the argument that is supplied when the callback function is registered, and passed to it whenever it is invoked. In *show* it is not needed, but in other programs you might use it to pass state or context information.
- `VStringConst` is defined by the Vista library as a pointer to a null-terminated string whose contents are guaranteed not to be changed. In ANSI C it is `const char *`, and in K&R C it is `char *`. (The `const` qualifier allows some ANSI C compilers like *gcc* to detect potential bugs, such as the inadvertent overwriting of a string constant.)

Using `VXPopupFileBox`, the Open callback function pops up a modal dialog box requesting the name of a file to open and display. The dialog box that VX puts up includes a text field for entering a filename, plus OK and Cancel buttons. When OK is pressed `VXPopupFileBox` returns the filename in dynamically-allocated storage; when Cancel is pressed, on the other hand, it returns `NULL`. No return is made until the user has dealt with the filename request in either one way or the other.

Once it receives a filename from the user, `OpenCallback` proceeds to open the file and display the image found within it. The `DisplayImage` routine, which reads and displays the image, is described below in section 5.9, **Displaying an image**. Failure to open the named file is signalled by calling `VWarning`. When VX is running, messages issued through `VWarning` are displayed in a pop-up, modal dialog box.

The Quit callback function is rather simple. It merely exits the program:

```
    void QuitCallback (client_data)
        VPointer client_data;
    {
        exit (0);
    }
```

5.7 Mouse input callback function

During the program's initialization phase we registered a callback function for mouse clicks in the drawing area. Here is part of that callback function:

```
    void SelectCallback (input_data, client_data)
        VXInputData input_data;
        VPointer client_data;
    {
        VDouble pixel;
```

```
        /* Only respond to the left button: */
        if (input_data->value != 1)
            return;

        /* Report the pixel clicked upon: */
        pixel = VGetPixel (image, 0, input_data->row, input_data->column);
        VXDisplayMessage (TRUE, "Row %d, column %d: %g",
                          input_data->row, input_data->column, pixel);
        ...
    }
```

One of the callback function's arguments, `input_data`, points to a structure describing the input event. Its fields list the type of event (e.g., a button press), which button or key was pressed, what other modifier keys were pressed at the same time (e.g., the SHIFT key), and where the mouse pointer was at the time of the event. Following a convention established by X Windows, mouse buttons are numbered from 1 to 5, with 1 denoting the leftmost button. The location of the mouse pointer is expressed in row/column coordinates by the structure's `row` and `column` fields.

This portion of the callback function fetches the value of the image pixel under the mouse pointer and reports it in the main window's message area. `VGetPixel` is a Vista library routine that returns the value of an image pixel at a specified band, row, and column. (The pixel is returned as a `VDouble` in order to accommodate any pixel type, from `VBit` through `VDouble`.) `VXDisplayMessage` is a VX routine for displaying text in the display area. Its first argument specifies whether to overwrite or append to existing text in the main window's display area; here we pass `TRUE` to overwrite the text. Its remaining arguments are like those of `printf`: they specify a format string and any values to be formatted. (According to the C language's rules for promoting arguments, `pixel` will be passed to `VXDisplayMessage` as a `double` regardless of whether `VDouble` is actually defined as `float` or `double` on the machine at hand.)

5.8 Drawing an overlay

Besides displaying a message about the pixel selected, `SelectCallback` also outlines the pixel by drawing lines on top of the image. The following code does the drawing:

```
VXClearLines ();
VXDrawLine (input_data->row - 0.5, input_data->column - 0.5,
            input_data->row - 0.5, input_data->column + 1.5);
VXDrawLine (input_data->row - 0.5, input_data->column + 1.5,
            input_data->row + 1.5, input_data->column + 1.5);
VXDrawLine (input_data->row + 1.5, input_data->column + 1.5,
            input_data->row + 1.5, input_data->column - 0.5);
VXDrawLine (input_data->row + 1.5, input_data->column - 0.5,
            input_data->row - 0.5, input_data->column - 0.5);
```

`VXClearLines` clears any lines previously drawn (i.e., those outlining the last pixel selected). The subsequent four calls to `VXDrawLine` draw four lines to form a box around the pixel. The lines will be drawn with a default color and width, although those attributes can be changed using other VX routines (`VXSetLineColor` and `VXSetLineWidth`).

5.9 Displaying an image

The final component of the *show* program is a utility routine, `DisplayImage`, that reads an image from a Vista data file and displays it. The routine is called when the program starts up, and whenever a new file is opened from the Open menu entry. Here it is:

```
VBoolean DisplayImage (in_file)
    FILE *in_file;
{
    VAttrList list;
    VImage *images;
    int nimages, i;

    /* Read the image from the specified file: */
    nimages = VReadImages (in_file, & list, & images);
    if (nimages == 0)
        return FALSE;       /* no image was found */

    /* Discard unneeded stuff: */
    VDestroyAttrList (list);
    if (nimages > 1) {
        VWarning ("Showing only the first of %d images", nimages);
        for (i = 1; i < nimages; i++)
            VDestroyImage (images[i]);
    }

    /* Note the new image while discarding any previous one: */
    if (image)
        VDestroyImage (image);
    image = images[0];
    VFree ((VPointer) images);

    /* Display the new image: */
    VXClearLines ();
    VXDisplayMessage (TRUE, "");
    VXSetImage (image, 0, 1.0, 0, 0);

    return TRUE;
}
```

`DisplayImage` uses the Vista library routine `VReadImages` to read the data file and extract the images from among its contents. (`VReadImages` is useful if you're primarily interested in the images a file contains — but since it discards information about where the images occur in relation to other attribute list elements, it isn't suitable for use in a filter program that needs to preserve the ordering of attributes.) The images are returned by `VReadImages` in a newly allocated array, `images`, and all other attributes are returned as an attribute list, `list`. To `nimages` it returns the number of images found, or 0 if an error occurred. We discard the attribute list, any extra images (after warning about them), and any image previously displayed.

Information about a pixel that was clicked in the old image will no longer be relevant so we call `VXClearLines` to clear the image overlay of lines (in particular, lines outlining a pixel) and `VXDisplayMessage` to erase the main window's message area.

`VXSetImage` supplies the new image to be displayed in the main window's drawing area. Along with the image, we specify the band to be displayed (0), the initial zoom level (1.0), and the row/column coordinates of the image's pan position. (At zoom level 1.0, the entire image is shown and therefore the pan position is irrelevant.) Once set by the `VXSetImage` call, the zoom level and pan position are controlled by the user's mouse commands.

That completes the *show* program. The remaining sections of this chapter describe ways of going beyond or beneath VX to accomplish things it isn't designed to do, and they list sources of additional information.

5.10 Beyond VX

If the VX library doesn't already do just what you need, here are some alternatives you might consider:

- Customize the appearance of the widgets VX uses by specifying resources for them. You'll need to know the widget hierarchy VX creates (documented in `VX(3Vi)`), and the resources a widget uses (documented in the *OSF/Motif Programmer's Reference Manual* and by a section 3X manual page named for each widget class).
- If you need a dialog box that VX cannot create — such as one containing radio buttons or check boxes — create it yourself using the OSF/Motif widget set and the X Toolkit. You can create any additional dialog boxes or shell windows that you need by using the VX main window as a parent widget. Its widget ID can be obtained by calling `VXGetApplicationShell(3Vi)`.
- VX provides routines for drawing image overlays of text, line segments, and sets of edges (see `VXDrawText(3Vi)`, `VXDrawLine(3Vi)`, and `VXDrawEdges(3Vi)`). However, if you need to draw something else you can draw directly on the image displayed in the drawing area using X library calls, bypassing the overlay drawing routines that VX provides. The drawing area consists of a `VImageView` widget, which is implemented by the Vista library (see `VImageView(3Vi)`). You can get that widget's ID by calling `VXGetImageViewWidget(3Vi)`, register your own callback function with its `VxNexposeCallback` resource, and have your callback function draw on the image directly using X library routines.
- Using the `VImageView` widget and any other widget library based on the X Toolkit (e.g., OSF/Motif or the Athena widgets), you can create a program that displays Vista images without relying on VX itself. The `VImageView` widget has built in support for zooming, panning, and dithering the image for any type of display.
- By default, programs based on VX and the `VImageView` widget will display colors using X Windows' default standard colormap. Use of the default standard colormap provides the greatest opportunity for sharing colors between several programs running on one display so that the programs can all obtain their needed colors. You can select an alternate colormap, however, if your needs go beyond what the default standard colormap provides (e.g., you need the greatest possible variety of grey shades at the expense of color hues). See `VColormap(3Vi)` for details.

5.11 For more information

More information on the topics, data structures, and routines introduced in this chapter can be found in the following manual pages:

`VX(3Vi)`	an overview of the VX library
`X(1)`	an overview of the X Window System
`VXInit(3Vi)`	initialize VX
`VXAddMenuCallback(3Vi)`	create a menu
`VXAddInputCallback(3Vi)`	register a callback function for input events
`VXAppMainLoop(3Vi)`	perform a main event loop
`VXPopupModal(3Vi)`	pop up a modal dialog box
`VXPopupModeless(3Vi)`	pop up a modeless dialog box
`VXDisplayMessage(3Vi)`	display text in the main window's message area
`VDrawLine(3Vi)`	draw a line over the displayed image
`VClearLines(3Vi)`	clear lines
`VXSetImage(3Vi)`	select an image for display
`VImageView(3Vi)`	widget for displaying an image
`VColormap(3Vi)`	colormap management functions

6 Other Vista Programming Facilities

Not all of the Vista's facilities can be surveyed in a brief introduction such as this. Here are some whose introductions have been left to the man pages:

- Vista includes many more macros and routines for working with images and attributes. They are summarized in `VImage(3Vi)` and `Vattribute(3Vi)`.
- Vista includes support for edge sets, which represent sets of connected points or lines defined on the image plane. Like images, edge sets can be both stored in data files and manipulated in memory. There are routines for displaying them under X Windows or in PostScript, and for creating them by linking image pixels. See `VEdges(3Vi)` for more information.
- You can define your own types of objects to supplement those that Vista provides such as attributes, images, and edges sets. The Vista library assists you in translating your types between internal (memory) and external (data file) formats. See `Vtype(3Vi)` for more information.
- There are routines to aid you in producing PostScript, including PostScript renditions of images and edge sets. See `Vpostscript(3Vi)` for more information.
- If your program requires a lot of parameters, you may with to organize them by collecting their definitions and values in a text file or Vista data file. See `Vpd(5Vi)` and `vpdc(1Vi)` for a description of a facility to help you do that.

A Program Listings

A.1 The *ramp* program

```
/*
 *  Program: ramp
 *
 *  Generates a test image of a specified size, with a linear intensity ramp.
 */

/* From the Vista library: */
#include <vista/Vlib.h>
#include <vista/file.h>
#include <vista/mu.h>
#include <vista/option.h>
```

```
#include <vista/VImage.h>

/* From the standard C library: */
#include <stdio.h>

int main (argc, argv)
    int argc;
    char *argv[];
{
    /* Command line options: */
    static VShort size = 256;
    static VOptionDescRec options[] = {
        { "size", VShortRepn, 1, (VPointer) & size,
              VOptionalOpt, NULL, "Number of image rows and columns" }
    };

    FILE *out_file;
    VImage image;
    int row, column, value;

    /* Parse command line arguments and identify the output file: */
    VParseFilterCmd (VNumber (options), options, argc, argv, NULL, & out_file);
    if (size <= 0)
        VError ("Size must be positive; %d isn't", (int) size);

    /* Create the image: */
    image = VCreateImage (1, (int) size, (int) size, VUByteRepn);
    if (! image)
        exit (1);

    /* Fill the image with a ramp: */
    for (row = 0; row < size; row++) {
        value = (float) row / (size - 1) * 255;
        for (column = 0; column < size; column++)
            VPixel (image, 0, row, column, VUByte) = value;
    }

    /* Write out the image: */
    if (! VWriteImages (out_file, NULL, 1, & image))
        exit (1);

    fprintf (stderr, "%s: Created a %d x %d ramp image.\n",
             (int) size, (int) size);
    return 0;
}
```

A.2 The *binarize* program

```
/*
 *  Program: binarize
 *
 *  Thresholds pixel values to produce a bit image.
 */

/* From the Vista library: */
```

```
#include <vista/Vlib.h>
#include <vista/mu.h>
#include <vista/option.h>
#include <vista/VImage.h>

/* From the standard C library: */
#include <stdio.h>

int main (argc, argv)
    int argc;
    char *argv[];
{
    /* Command line options: */
    static VDouble threshold = 0.0;
    static VOptionDescRec options[] = {
        { "threshold", VDoubleRepn, 1, (VPointer) & threshold,
              VOptionalOpt, NULL, "Value at which pixel is set to 1" }
    };

    FILE **in_file, *out_file;
    VAttrList list;
    VAttrListPosn posn;
    VImage src, result;
    int nimages = 0;

    /* Parse command line arguments and identify files: */
    VParseFilterCmd (VNumber (options), options, argc, argv,
                     & in_ file, & out_file);

    /* Read the input file: */
    list = VReadFile (in_file, NULL);
    if (! list)
        exit (1);

    /* For each attribute read... */
    for (VFirstAttr (list, & posn); VAttrExists (& posn); VNextAttr (& posn)) {

        /* If it's an image, replace it with a thresholded version: */
        if (VGetAttrRepn (& posn) != VImageRepn)
            continue;
        VGetAttrValue (& posn, NULL, VImageRepn, & src);
        result = BinarizeImage (src, NULL, VAllBands, threshold);
        if (! result)
            exit (1);
        VSetAttrValue (& posn, NULL, VImageRepn, result);
        VDestroyImage (src);
        nimages++;
    }

    /* Write out the results: */
    if (! VWriteFile (out_file, list))
        exit (1);

    fprintf (stderr, "%s: Thresholded %d image(s).\n", argv[0], nimages);
    return 0;
```

```
}
```

A.3 The BinarizeImage routine

```
/*
 *  BinarizeImage
 *
 *  Convert pixel values to 0 or 1 according to whether they are less
 *  than a given threshold.
 */

VImage BinarizeImage (src, dest, band, threshold)
    VImage src;
    VImage dest;
    VBand band;
    double threshold;
{
    int nbands, npixels, i;
    VPointer src_pixels;
    VBit *dest_pixels;

    /* Locate the block of source image pixels: */
    if (! VSelectBand ("BinarizeImage", src, band, & npixels, & src_pixels))
        return NULL;

    /* Grumble about a threshold outside the source pixel range: */
    if (threshold < VPixelMinValue (src) || threshold > VPixelMaxValue (src))
        VWarning (
    "BinarizeImage: Threshold (%g) is outside source pixel range [%g,%g]",
              threshold, VPixelMinValue (src), VPixelMaxValue (src));

    /* Check the destination image, which should have VBit pixels.
       If it is NULL, then create one of the appropriate size and type. */
    nbands = (band == VAllBands) ? VImageNBands (src) : 1;
    dest = VSelectDestImage ("BinarizeImage", dest, nbands,
                             VImageNRows (src), VImageNColumns (src),
                             VBitRepn);
    if (! dest)
        return NULL;

    /* Locate the block of destination image pixels: */
    dest_pixels = & VPixel (dest, 0, 0, 0, VBit);

    /* Define a macro, parameterized by source pixel type,
       that can be instantiated for each possible type: */
#define Binarize(type)                                          \
    {                                                           \
        type *src_pp = (type *) src_pixels;                     \
        type t = threshold;                                     \
                                                                \
        for (i = 0; i < npixels; i++)                           \
            *dest_pixels++ = (*src_pp++ >= t);                  \
    }

    /* Instantiate the macro once for each source pixel type: */
```

```
    switch (VPixelRepn (src)) {
    case VBitRepn:      Binarize (VBit);        break;
    case VUByteRepn:    Binarize (VUByte);      break;
    case VSByteRepn:    Binarize (VSByte);      break;
    case VShortRepn:    Binarize (VShort);      break;
    case VLongRepn:     Binarize (VLong);       break;
    case VFloatRepn:    Binarize (VFloat);      break;
    case VDoubleRepn:   Binarize (VDouble);     break;
    }

#undef Binarize

    /* Let the destination inherit any attributes of the source image: */
    VCopyImageAttrs (src, dest);
    return dest;
}
```

A.4 The *show* program

```
/*
 *  Program: show
 *
 *  Display an image under X Windows.
 */

/* From the Vista library: */
#include <vista/VX.h>
#include <vista/file.h>
#include <vista/option.h>
#include <vista/VImage.h>

/* From the standard C library: */
#include <stdio.h>

/* Later in this file: */
extern void OpenCallback ();
extern void QuitCallback ();
extern void SelectCallback ();
extern VBoolean DisplayImage ();

/* The image currently displayed: */
VImage image = NULL;


/*
 *  main
 *
 *  Program entry point.
 */

int main (argc, argv)
    int argc;
    char *argv[];
{
    /* Default resource settings: */
```

```
    static VStringConst resources[] = {
        "*messageAreaNLines: 2",
        "*fileBox.textColumns: 50",
        "*fileBox.pattern: *.v",
        "*imageView.cursor: crosshair",
        NULL
    };

    FILE *in_file;

    /* Parse command line options and open the input file: */
    VXInit ("Show", resources, & argc, argv);
    VParseFilterCmd (0, NULL, argc, argv, & in_file, NULL);

    /* Read and display an image from the input file: */
    if (! DisplayImage (in_file))
        exit (1);       /* no image could be found */
    fclose (in_file);

    /* Create the File menu: */
    VXAddMenu ("File",
               "Open...", "Ctrl<Key>O", OpenCallback, NULL,
               "Quit", "Ctrl<Key>Q", QuitCallback, NULL,
               NULL);

    /* Add a handler for mouse clicks on the image: */
    VXAddInputCallback (VXIbuttonPress, SelectCallback, NULL);

    /* Process events: */
    VXAppMainLoop ();
}


/*
 *  OpenCallback
 *
 *  Callback for Open menu item.
 */

/*ARGSUSED*/
void OpenCallback (client_data)
    VPointer client_data;
{
    VStringConst filename;
    FILE *in_file;

    /* Pop up a dialog box to get the new file's name: */
    filename = VXPopupFileBox ("Open Image File");
    if (! filename)
        return;         /* Cancel button was clicked */

    /* Open it and display the image it contains: */
    in_file = fopen (filename, "r");
    if (in_file) {
        DisplayImage (in_file);
```

```
        fclose (in_file);
    } else VWarning ("Unable to open %s", filename);
    VFree ((VPointer) filename);
}


/*
 *  QuitCallback
 *
 *  Callback for Quit menu item.
 */

/*ARGSUSED*/
void QuitCallback (client_data)
    VPointer client_data;
{
    exit (0);
}


/*
 *  SelectCallback
 *
 *  Respond to a mouse click in the image.
 */

void SelectCallback (input_data, client_data)
    VXInputData input_data;
    VPointer client_data;
{
    VDouble pixel;

    /* Only respond to the left button: */
    if (input_data->value != 1)
        return;

    /* Report the pixel clicked upon: */
    pixel = VGetPixel (image, 0, input_data->row, input_data->column);
    VXDisplayMessage (TRUE, "Row %d, column %d: %g",
                      input_data->row, input_data->column, pixel);

    /* Outline the pixel: */
    VXClearLines ();
    VXDrawLine (input_data->row - 0.5, input_data->column - 0.5,
                input_data->row - 0.5, input_data->column + 1.5);
    VXDrawLine (input_data->row - 0.5, input_data->column + 1.5,
                input_data->row + 1.5, input_data->column + 1.5);
    VXDrawLine (input_data->row + 1.5, input_data->column + 1.5,
                input_data->row + 1.5, input_data->column - 0.5);
    VXDrawLine (input_data->row + 1.5, input_data->column - 0.5,
                input_data->row - 0.5, input_data->column - 0.5);
}


/*
```

```
 *  DisplayImage
 *
 *  Load and display a new image.
 */

VBoolean DisplayImage (in_file)
    FILE *in_file;
{
    VAttrList list;
    VImage *images;
    int nimages, i;

    /* Read the image from the specified file: */
    nimages = VReadImages (in_file, & list, & images);
    if (nimages == 0)
        return FALSE;       /* no image was found */

    /* Discard unneeded stuff: */
    VDestroyAttrList (list);
    if (nimages > 1) {
        VWarning ("Showing only the first of %d images", nimages);
        for (i = 1; i < nimages; i++)
            VDestroyImage (images[i]);
    }

    /* Note the new image while discarding any previous one: */
    if (image)
        VDestroyImage (image);
    image = images[0];
    VFree ((VPointer) images);

    /* Display the new image: */
    VXClearLines ();
    VXDisplayMessage (TRUE, "");
    VXSetImage (image, 0, 1.0, 0, 0);

    return TRUE;
}
```