A Programmer’s Guide to the Generalized Image Library
Leonard Hamey
Harry Printz
Doug Reece
Ralph Hyre
Steve Shafer
Computer Science Department.
Carnegie-Mellon University.
Pittsburgh, PA 15213

Revised August 9, 1990
Abstract

This document describes the generalized image library from the viewpoint of a programmer who wishes to use the library in his programs. A separate document describes the facilities which the generalized image library provides for the users of such programs. A third document describes the generalized image library from the viewpoint of a developer who wishes to add to the library's capabilities.
Copyright © Leonard G. C. Hamey

This research was sponsored by the Defense Advanced Research Projects Agency, DOD, through ARPA Order No. 4976, and monitored by the Air Force Avionics Laboratory under contract F30602-84-K-1520. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or of the U.S. Government.
Contents

1 Introduction and Notation .......................... 1
  1.1 Compiling Programs ................................ 2
     1.1.1 Loading the Library .......................... 2
     1.1.2 Compilation .................................. 3
  1.2 Data Types and Pixels ............................ 4
  1.3 Initializing the Library ........................... 4

2 Accessing Image Data ............................... 6
  2.1 Multi-Band Images ................................ 6
  2.2 Pixel Access ....................................... 7
  2.3 Multi-band Pixel Access ........................... 9
  2.4 Row Access ......................................... 10
  2.5 Multi-band Row Access ............................ 11
  2.6 Column Access ...................................... 12
  2.7 Multi-band Columns Access ........................ 14
  2.8 Box-Access .......................................... 14
  2.9 Multi-band Box Access ............................. 17
  2.10 Filling an Entire Image ........................... 21
  2.11 Clearing Images ................................... 21
  2.12 Raster-Scan Access ............................... 24
  2.13 Copying Images ................................... 27

3 Opening and Closing Images ......................... 29
  3.1 High-Level Single-Band Image Open Routines ....... 29
  3.2 High-Level Multi-Band Image Open Routines ........ 32
  3.3 Low-Level Image Open Routines .................... 34
     3.3.1 CMU Format Images ............................. 35
     3.3.2 GIF Format Images .............................. 36
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.3</td>
<td>MIP Format Images</td>
<td>36</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Network Connections</td>
<td>37</td>
</tr>
<tr>
<td>3.3.5</td>
<td>Matrox Display</td>
<td>38</td>
</tr>
<tr>
<td>3.3.6</td>
<td>Datacube Display</td>
<td>38</td>
</tr>
<tr>
<td>3.4</td>
<td>Closing Images</td>
<td>39</td>
</tr>
<tr>
<td>3.5</td>
<td>Additional Notes</td>
<td>40</td>
</tr>
<tr>
<td>4</td>
<td>Images and Matrices</td>
<td>41</td>
</tr>
<tr>
<td>5</td>
<td>Changing Images</td>
<td>43</td>
</tr>
<tr>
<td>5.1</td>
<td>Discussion</td>
<td>43</td>
</tr>
<tr>
<td>5.2</td>
<td>Duplicating an IMAGE Descriptor</td>
<td>44</td>
</tr>
<tr>
<td>5.3</td>
<td>Cropping an Image</td>
<td>45</td>
</tr>
<tr>
<td>5.4</td>
<td>Shifting an Image</td>
<td>46</td>
</tr>
<tr>
<td>5.5</td>
<td>Simple Linear Transformation</td>
<td>47</td>
</tr>
<tr>
<td>5.6</td>
<td>Multi-band Image from Single-hand Images</td>
<td>47</td>
</tr>
<tr>
<td>5.7</td>
<td>Converting Multi-Band Images</td>
<td>49</td>
</tr>
<tr>
<td>5.8</td>
<td>Magnification and Reduction</td>
<td>49</td>
</tr>
<tr>
<td>5.9</td>
<td>Image Quantization and Halftoning</td>
<td>50</td>
</tr>
<tr>
<td>5.10</td>
<td>Windowing Facilities</td>
<td>51</td>
</tr>
<tr>
<td>5.10.1</td>
<td>Creating a Window Image</td>
<td>52</td>
</tr>
<tr>
<td>5.10.2</td>
<td>Panning and Tracking Windows</td>
<td>52</td>
</tr>
<tr>
<td>5.10.3</td>
<td>Attaching Windows to Images</td>
<td>53</td>
</tr>
<tr>
<td>5.10.4</td>
<td>Window Options</td>
<td>54</td>
</tr>
<tr>
<td>6</td>
<td>Operations on Images</td>
<td>56</td>
</tr>
<tr>
<td>6.1</td>
<td>Miscellaneous macros</td>
<td>56</td>
</tr>
<tr>
<td>6.2</td>
<td>Miscellaneous Routines</td>
<td>57</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Property List Management</td>
<td>57</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Digitizing Images</td>
<td>62</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Temporary Images</td>
<td>63</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Cursor Positioning</td>
<td>64</td>
</tr>
<tr>
<td>6.2.5</td>
<td>Manipulating Subimages</td>
<td>65</td>
</tr>
<tr>
<td>6.2.6</td>
<td>Controlling the CMU Image Module</td>
<td>65</td>
</tr>
<tr>
<td>6.3</td>
<td>Library Environment Options</td>
<td>66</td>
</tr>
<tr>
<td>7</td>
<td>Simple Graphics</td>
<td>68</td>
</tr>
<tr>
<td>7.1</td>
<td>Line and Figure Drawing Routines</td>
<td>68</td>
</tr>
<tr>
<td>7.2</td>
<td>Test Routines</td>
<td>72</td>
</tr>
</tbody>
</table>
8 Image Sequences
  8.1 The Image Sequence Structure .......................................... 74
  8.2 Opening Image Sequences ............................................. 74
  8.3 Accessing the Images of a Sequence .................................. 75
  8.4 Closing Image Sequences ................................................ 77

9 Error Handling
  9.1 Default Error Handling ............................................. 79
  9.2 Error Trapping ......................................................... 79

10 Examples ................................................................. 84

11 Bugs ................................................................. 89

12 Lisp Interface .......................................................... 90
  12.1 Loading the Lisp Interface .......................................... 90
  12.2 Image Access Operations ............................................. 91
    12.2.1 Single-Band Image Put Operations ......................... 91
    12.2.2 Multi-Band Image Put Operations ......................... 92
    12.2.3 Single-Band Image Get Operations ......................... 93
    12.2.4 Multi-Band Image Get Operations ......................... 94
  12.3 Creating Arrays for Pixel Data ..................................... 95
  12.4 Copying Images ...................................................... 95
  12.5 Opening and Closing Images ........................................ 96
  12.6 Changing Images .................................................... 97
  12.7 Image Descriptive Information ..................................... 98
  12.8 Simple Graphics .................................................... 99
  12.9 Text Routines ........................................................ 100
  12.10 Error Handling ...................................................... 101
  12.11 A Simple Example .................................................. 103
Chapter 1

Introduction and Notation

This document describes the generalized image library from the viewpoint of the programmer who wishes to use the library. Throughout this document, routine declarations are cited as a concise description of their calling conventions. For example, the declaration of `i-open` is given as follows.

```
IMAGE *i_open (name, mode, err)

char *name;
int mode;
int *err;
```

This declaration shows that `i-open` has three parameters, `name`, `mode`, and `err` and that it returns a pointer to an `IMAGE` structure. The interpretation of each parameter is not necessarily clear - it will be expanded upon in the surrounding text. However, the declarations do identify the data type of each parameter. For example, `mode` is identified as an integer parameter. This means that any C expression which evaluates to an integer may be used as the `mode` parameter in a call to `i_open`. Similarly, `name` is identified as a character pointer, or string. This means that any C expression which evaluates to a pointer to a character string may be used as the `name` parameter in a call to `i-open`. In particular, `name` may be a quoted character string or the name of an array of characters. It is also possible for `name` to be a variable which has been declared as a pointer to characters. However, such a variable must have been assigned a meaningful value before `i-open` is called. The following function is incorrect and would probably cause the program to crash.

```
#include <gimage.h>
```

However, not all values may be acceptable to `i-open`. In fact, only the values `IMREAD` and `IMMODIFY` are meaningful.
IMAGE *myopen()
{
    char *myfile;
    return (i_open (myfile, 0, NULL);
}

The last parameter of many routines is the error handling parameter err. In most cases, as in the examples throughout this document. NULL can be passed for the error handling parameter. We strongly recommend that you use NULL for the error handling parameter whenever possible. and that you read section 9 carefully before attempting other methods of error handling.

1.1 Compiling Programs

Every C module which uses the generalized image library must include the header file gimage.h. gimage.h is normally the only include file which is required to be able to use the library. All generalized image library include files are commonly located is /usr/include/vision. The following line is therefore required at the top of your C modules.

#include <vision/gimage.h>

Gimage.h contains definitions for the entire library. Specifically, the following aspects of the library are defined in gimage.h.

- The image access routines (section 2).
- The routines to open and close images (sections 3 and 3.4).
- Routines for creating virtual images (section 5).
- Useful macros (section 6.1).
- The subimage package.

1.1.1 Loading the Library

In order for your program to run, the generalized image library must be loaded with it. There are two ways in which this can be done. One way is to load the library along with your program just
as you would load any other library. This results in very large programs which consume lots of disk space.

The alternative is to load the library into your program at run time. Surprisingly, this is not a slow process and it results in much smaller executable programs. In addition, because the library is loaded at run time your program does not need to be relinked in order to take advantage of some new features as they are added to the library. For example, if a new functionality is added to i-open, your program can take advantage of that new functionality without even being relinked.

If you need additional debugging information in your program, you should compile with the -g and -DDEBUG options and link with the debugging version of the generalized image library. The -DDEBUG option causes additional code to be generated which is capable of performing a cyclic redundancy check on the generalized image structure before each pixel access routine is called. This check assists in detecting problems where your program is corrupting one of the generalized image structures. The cyclic redundancy checks are computationally expensive and are only performed if the IMDEBUG environment variable contains the option letter c. (See the section dealing with the environment variables in A User's Guide to the Generalized Image Library).

It is strongly recommended that you use make(1) to compile your programs. The following sections give details for the construction of the appropriate Makefile.

### 1.1.2 Compilation

When your program is being compiled, you must ensure that the C compiler will find gimage.h in /usr/include/vision. You must also load your programs with the generalized image library. You can achieve this with a Makefile similar to the following example for compiling a program called copy.c.

```
copy: copy.o
    cc -o copy copy.o -lgimage
```

To enable debugging, use a Makefile similar to the following.

```
#CFLAGS for debugging
CFLAGS= -g -DDEBUG

copy: copy.o
    cc $(CFLAGS) -o copy copy.o -lgimage
```
1.2 Data Types and Pixels

A pixel is the value stored at a given location in the image. The generalized image library supports three representations of pixels: unsigned integer quantities, signed integer quantities and floating-point quantities.

Of course, not every device or virtual image will support every pixel representation. For example, the Matrox boards support only unsigned 8-bit quantities. Even when all three representations are supported, as is the case for image files on disk, the particular representation used in a given file must be selected when the file is created. Subsequently, the representation cannot be changed.

So, what if your image is unsigned integers and your program only deals with floating-point? No problem. The generalized image library provides automatic type conversion between pixel representations and the data types which your program manipulates. This means that your program can fetch the pixels as floating-point quantities, perform computations on them and store them as floating-point quantities. The generalized image library will take care of converting the integer pixels into floating-point and converting the floating-point values back to integers.

Your program indicates the data type it requires by invoking the appropriate image access routine (see section 2). The generalized image library knows the pixel representation which is supported by the device or file and automatically performs any necessary conversions. Integer quantities are converted to floating-point in the obvious way. Floating-point quantities are rounded off to the nearest integer. For example, a value of 3.45 would round off to 3 but a value of 3.55 would round off to 4. Values which are exactly half-way between two sequential integers may round either up or down. For example, 3.5 may round off to 3 or 4 depending on the implementation.

In some cases, conversions between pixels and user data types may violate the allowable value range of a data type. For example, -3.5 cannot be converted to an unsigned integer because unsigned integers are non-negative. In such cases, the action of the generalized image library is undefined. For instance, the value may be range-truncated. (In the example, -3.5 would be truncated to 0, the nearest extreme value of the valid range.) The value may also be wrapped into the valid range. A common effect when out-of-range values are stored on a device such as the Matrox. It is even possible that the program will crash.

1.3 Initializing the Library

Before you can use any of the functions in the generalized image library, the library must be initialized. This is achieved by calling `i.init`. It is a good idea to call `i.init` as one of the very first things your program does. In fact, it is essential that you call `i.init` before any `printf` calls if your program is to run correctly on the Suns with the library loaded at run time. The best policy is to be kind to the library and call `i.init` before doing any other computation. `i.init` has no arguments and is called as shown in the example in figure 1.1.
#include <gimage.h>

main (argc, argv)
int argc;
char **argv;
{
    IMAGE *in, *out;
    i_init O;
    ...
}

Figure 1.1: Use of i_init.
Chapter 2

Accessing Image Data

The generalized image library provides many methods of accessing and changing the pixel data stored in images. These range from simple operations which fetch or store individual pixels, to the apply routines which provide raster-scan access to entire images. In this section, we will first describe the pixel access routines which are simple but slow, and then proceed to detail more complex access methods which provide greater speed but are less flexible.

The image access operations are provided in several flavors which correspond to different user-program data types. The generalized image library also supports a number of image pixel data types. The conversion between user-program data types and image pixel data types is fully automatic: floating-point quantities are always rounded when being converted to integers.

2.1 Multi-Band Images

Traditionally, an image is a rectangular array of pixel values, each value representing the intensity of light (or some other scalar quantity) at a point in the image plane. The generalized image library extends this traditional view of images by supporting multi-band images. In a multi-band image, each pixel location is associated with a vector of scalar values—one value for each image band. For example, the hlatro boards on the Suns have three separate values for each pixel: the red, green and blue intensity values. When the hlatro is opened as an image, the generalized image library creates a multi-band image with red, green, and blue bands. The multi-band image access routines access the three bands simultaneously, fetching or storing pixel values which are red-green-blue color vectors.

Multi-band images must be accessed with the multi-band image access routines. If an ordinary image access routine is invoked to access a multi-band image, the error will be detected, a message will be printed and the program will be aborted. Ordinary images, however, may be accessed with the multi-band routines. This means that the multi-band access routines are completely general.
float i - fgetpixel (img, mu, col)
long int i - lgetpixel (img, row, col)
int i - igetpixel (img, mu, col)
short int i - sgetpixel (img, row, col)
unsigned char i - cgetpixel (img, row, col)

IMAGE *img;
int row, col;

Figure 2.1: The getpixel routines.

fact which is very useful for implementing some of the library's facilities.

2.2 Pixel Access

Access to individual pixels is achieved by the five kinds of getpixel and putpixel operations. The five getpixel operations correspond to five different types of data which may be stored in the user's C program. They are: i_fgetpixel for float, i_lgetpixel for long int, i_igetpixel for int, i_sgetpixel for short int, and i_cgetpixel for unsigned char. The getpixel operations are functions which return a single pixel value as their function value. The three arguments specify the image from which a pixel is to be fetched and the row and column coordinates of the pixel to be fetched. The function declarations are in figure 2.1.

For example, the following C statement fetches the pixel in row 5, column 7 of myimg as a floating-point quantity.

x = i_fgetpixel (myimg, 5, 7);

There are also five putpixel operations which correspond to the same five user-program data types: i_fputpixel for float, i_lputpixel for long int, i_iputpixel for int, i_sputpixel for short int, and i_cputpixel for unsigned char. Each putpixel operation accepts four arguments: the image in which a pixel is to be stored, the row and column coordinates of the pixel and the value to be stored. The putpixel operations are declared as shown in figure 2.2.

For example, suppose we need a routine which multiplies all the pixels in a section of an image by a constant and copies them to another image. We can write a doubly-nested loop which fetches

Stricly, the image access operations such as ifgetpixel are macros which invoke functions, but the user may view them as functions.
Figure 2.2: The putpixel routines.

```c
#include <gimage.h>

multimg (outimg, inimg, mult, rstart, rend, cstart, cend)
IMAGE *outimg, inimg;
float mult;
int rstart, rend, cstart, cend;
{
    int row, col;
    float x;
    for (row = rstart; row <= rend; row++)
        for (col = cstart; col <= cend; col++)
            {
                x = i_fgetpixel (inimg, row, col);
                i_fputpixel (outimg, row, col, mult * x);
            }
```

Figure 2.3: A routine to multiply an image by a constant.

Each pixel in turn multiplies it and stores it into the output image, as in figure 2.3. This routine achieves the desired goal, but is not very efficient. In later sections we will see how it can be made faster by using other routines provided in the image library.

The above routine manipulates the pixels as floating-point quantities. Floating-point was chosen because it is the most general type available, being able to represent all the floating-point pixel values and most of the integer pixel values. The images may not actually contain floating-point pixels: if outimg is an integer image then *f*putpixel will round the floating-point values to integers for storing in the image.
i_mfgetpixel (img, row, col, fvec)
i_mlgetpixel (inzg, row, col, lvec)
i_migetpixel (inzg, row, col, ivec)
i_msgetpixel (inzg, row, col, svec)
i_mcgetpixel (img, row, col, cvec)

i_mfputpixel (img, row, col, free)
i_mlputpixel (inzg, row, col, Zvec)
i_miputpixel (inzg, row, col, Zvec)
i_msputpixel (img, row, col, svec)
i_mcpuperpixel (img, row, col, cvec)

IMAGE *img;
int row, col;
float *fvec;
long int *lvec;
int *ivec;
short int *svec;
unsigned char *cvec;

Figure 2.4: The multi-band pixel access routines.

2.3 Multi-band Pixel Access

The multi-band pixel access routines may be used to fetch and store individual pixels of a multi-band image. Since each pixel of the image has a vector of values associated with it, the multi-band getpixel routines return the vector of values in a user-program data vector. Similarly, the multi-band putpixel routines store the vector from a user-program data vector. The vector is passed to the routines as the fourth argument. The function definitions for the multi-band getpixel and putpixel routines are shown in figure 2.4.

As an example of the application of the multi-band pixel access routines, consider writing a procedure mkbwimg which converts a color image to black and white. The routine is in figure 2.5. It fetches the color pixels and computes the average of the three intensity values which is then stored as the black and white intensity value. In this program, rgb is the vector which receives the red, green and blue intensity values from the image. The pixel values are fetched as floating-point quantities because float is the most general data type. I_mfgetpixel is the multi-band floating-


```c
#include <gimage.h>

mkbwimg (bwimg, rgbimg, rstart, rend, cstart, cend)
IMAGE *bwimg, *rgbimg;
int rstart, rend, cstart, cend;
{
    int row, col;
    float rgb[3];
    for (row = rstart; row <= rend; row++)
        for (col = cstart; col <= cend; col++)
            {
                i_mfgetpixel (rgbimg, row, col, rgb);
                i_fputpixel (bwimg, row, col,
                    (rgb[0] + rgb[1] + rgb[2]) / 3.0);
            }
    return;
}
```

Figure 2.5: One method of converting a color image to black and white.

point pixal fetch routine.

### 2.4 Row Access

Instead of accessing individual image pixels, it is possible to access entire rows or portions of rows with one routine call. This can result in appreciable speed increases due to reduced overhead. The `apply` routines, which are described in section 2.12, use row access to provide efficient raster-scan access to an image.

Each row access routine comes in five flavours corresponding to the same five user-program data types which are supported by the pixel access routines. The five `getrow` routines are used to fetch a portion of an image row. They each have five arguments which specify the image. the row coordinate, the first and last column coordinates and the address of a data vector to receive the pixel values. The routine declarations are in figure 2.6.

The five `putrow` routines may be used to store data from a vector into a portion of an image row. The five arguments are the same as the `getrow` arguments: the image, the row, the column
Figure 2.6: The getrow routines.

start and end, and the address of the data vector. The declarations are in figure 2.7.

We can use these routines to rewrite the multimg example as in figure 2.8.

In this routine, we have substituted the more efficient getrow and putrow operations for the pixel access routines. We have also imposed a width restriction on the image area to be processed; this could be removed by a programming trick. The major problem is that the program is becoming a little difficult to understand. In section 2.12 we will show how the apply routines can be used to provide an efficient and clear implementation of multimg.

In addition to the routines for fetching and storing rows of data, there are five fillrow routines. These routines may be used to fill a portion of an image row with a constant value. The five arguments specify the image to be affected, the row, the first and last column coordinates and the fill value. The routine declarations are in figure 2.9.

2.5 Multi-band Row Access

Instead of accessing individual pixels of a multi-band image one at a time, it is possible to access an entire row or portion of a row with one routine call. This has the advantage of increased speed due to reduced overhead. The multi-band getrow and putrow routines transfer the pixel values from each band into a separate user-program data array. The routines expect their fifth argument to be a vector of pointers, one pointer for each band of the image. Each pointer indicates the location of
The data array to be used for a particular image band. The function declarations for the multi-band `getrow` and `putrow` routines are in figure 2.10.

The multi-band row access routines can be used to provide a more efficient implementation of `mkbwimg`. As with the implementation of `multimg` in section 2.4, a width restriction has been imposed on the images which could be removed with a little more effort. The improved `mkbwimg` routine is shown in figure 2.11.

In addition to the multi-band `getrow` and `putrow` routines, multi-band routines are provided which fill a portion of a row with constant values. The constant values are passed in a vector with one value for each band of the image. Figure 2.12 contains the function declarations.

### 2.6 Column Access

Access to portions of individual columns of an image is provided by routines analogous to the row access routines described in section 2.4. The five `getcol` routines fetch columns of pixels into a data vector, the five `putcol` routines store columns of pixels from a data vector and the five `fillcol` routines fill columns of pixels with a constant value. The five arguments are: the image, the first and last row coordinates, the column coordinate and the address of the data vector (or the constant value). The routine declarations are in figure 2.13.
```c
#include <gimage.h>

#define MAXWIDTH 1024

multimg (outimg, inimg, mult, rstart, rend, cstart, cend)
IMAGE *outimg, inimg;
float mult;
int rstart, rend, cstart, cend;
{
    int row, col;
    float x[MAXWIDTH] ;
    if (cend - cstart >= MAXWIDTH)
    {
        printf ('Image too wide.\n');
        return;
    }
    for (row = rstart; row <= rend; row++)
    {
        i_fgetrow (inimg, row, cstart, cend, x);
        for (col = 0; col <= cend - cstart; col++)
        {
            x[col] = x[col] * mult;
        }
        i_fputrow (outimg, row, cstart, cend, x);
    }
}
```

Figure 2.8: A better way to multiply an image by a constant.
2.7 Multi-band Columns Access

The multi-band column access routines are the multi-band equivalent of the routines described in the previous section. The multi-band getcol and putcol routines require a vector of pointers as their fifth parameter. Each pointer indicates a user-program data vector which is used for one of the image bands. The routine declarations are shown in figure 2.14.

2.8 Box Access

Entire rectangular regions of the image can be accessed in a single operation by the getbox, putbox and fillbox routines. The five getbox routines fetch boxes of pixels into a two-dimensional array. the five putbox routines store boxes of pixels from a two-dimensional array. and the five fillbox routines fill boxes of pixels with a constant value. The seven arguments for getbox and putbox are: the image, the first and last row coordinates, the first and last column coordinates, the address of the two-dimensional array and the second dimension of the array. The six arguments for the fillbox routines are: the image, the first and last row and column co-ordinates and the constant fill value. The routine declarations are shown in figure 2.15.

An example of the use of box access, consider writing a routine av3x5 (see figure 2.16). Av3x5 computes, for a specified pixel location, the average intensity over a rectangular region 3 rows high and 5 columns wide surrounding the pixel. It uses a C two-dimensional array to store the block of

```c
i_ffillrow (img, row, cstart, cend, fval)
i_lffillrow (img, row, cstart, cend, lval)
i_iffillrow (img, row, cstart, cend,IVAL)
i_sfillrow (iiiig, row, cstart, cend, sval)
i_cfillrow (img, row, cstart, cend, cval)
```

Figure 2.9: The fillrow routines.
i_mfgetrow (img, row, cstart, cend, fpdata)
i_mlgetrow (img, row, cstart, cend, lpdata)
i_migetrow (img, row, cstart, cend, ipdata)
i_msgetrow (img, row, cstart, cend, spdata)
i_mcgetrow (img, row, cstart, cend, cpdata)

i_mfputrow (img, row, cstart, cend, fpdata)
i_mlputrow (img, row, cstart, cend, lpdata)
i_miputrow (img, row, cstart, cend, ipdata)
i_msputrow (img, row, cstart, cend, spdata)
i_mcputrow (img, row, cstart, cend, cpdata)

IMAGE *img;
int row, cstart, cend;
float **fpdata;
long int **lpdata;
int **ipdata;
short int **spdata;
unsigned char **cpdata;

Figure 2.10: The multi-band getrow and putrow routines.
# include <gimage.h>

mkbwimg (bwimg, rgbimg, rstart, rend, cstart, cend)
IMAGE *bwimg, *rgbimg;
int rstart, rend, cstart, cend;
{
    float red[MAXWIDTH], green[MAXWIDTH], blue[MAXWIDTH];
    float bw[MAXWIDTH];
    float *bands[3];
    register int i, row;
    bands[0] = red;
    bands[1] = green;
    bands[2] = blue;
    if (cend - cstart + 1 > MAXWIDTH)
    {
        printf ('Image too wide.\n');
        return;
    }
    for (row = rstart; row <= rend; row++)
    {
        i_mfgetrow (rgbimg, row, cstart, cend, bands);
        for (i = cend - cstart; i >= 0; i--)
            bw[i] = (red[i] + green[i] + blue[i]) / 3.0;
        i_fputrow (img, row, cstart, cend, bw);
    }
    return;
}

Figure 2.11: Converting a color image to **black and white.**
i_mffillrow (inzg, row, cs, ce, fvec)
i_mlfillrow (zing, row, ce, ce, lvec)
i_mifillrow (img, row, cs, ce, itvec)
i_msfillrow (img, row, cs, ce, svc)
i_mcfillrow (img, row, cs, ce, cvec)

```c
IMAGE *img;
int row, cs, ce;
float *fvec;
long int *lvec;
int *itvec;
short int *svc;
unsigned char *cvec;
```

Figure 2.12: The multi-band fillrow routines.

pixels which it fetches with ifgetbox. The computed average is returned as the function value.

### 2.9 Multi-band Box Access

The multi-band box access routines are similar to the ordinary box access routines described in the previous section, but are designed for accessing multi-band images. Entire rectangular regions of the image can be accessed in a single operation. The getbox and putbox routines have seven parameters: the image, the indices of the first and last row and column to be accessed, a vector of pointers and an array dimension. Each of the pointers in the vector indicates a separate two-dimensional array which is used to store the data for a particular band of the image. The arrays must all have the same dimensions; the second dimension of the arrays is passed as the last parameter.

The fillbox routines have six parameters: the image, the indices of the first and last row and column to be accessed, and a vector of constant values. The vector contains one value for each image band. The declarations of the multi-band box routines are given in figure 2.17.
i_fgetcol (img, rstart, rend, col, fdata)
i_lgetcol (img, rstart, rend, col, ldata)
i_sgetcol (img, rstart, rend, col, sdata)
i_cgetcol (mg, rstart, rend, col, cdata)

i_fputcol (img, rstart, rend, col, fdata)
i_lputcol (img, rstart, rend, col, ldata)
i_sputcol (img, rstart, rend, col, sdata)
i_cputcol (mg, rstart, rend, col, cdata)

i_ffillcol (img, rstart, rend, col, fval)
i_lfillcol (img, rstart, rend, col, lval)
i_ifillcol (img, rstart, rend, col, rval)
i_sfillcol (img, rstart, rend, col, sval)
i_cfillcol (img, rstart, rend, col, cval)

IMAGE *img;
int rstart, rend, col;
float *fdata, fval;
long int *ldata, lval;
int *idata, ival;
short int *sdata, sval;
unsigned char *cdata, cval;

Figure 2.13: The column access routines.
Figure 2.14: The multi-band column access routines.
i_fgetbox (img, rstart, rend, cstart, cend, fdata, dim)
i_lgetbox (img, rstart, rend, cstart, cend, ldata, dim)
i_ggetbox (img, rstart, rend, cstart, cend, idata, dim)
i_sgetbox (img, rstart, rend, cstart, cend, sdata, dim)
i_cgetbox (img, rstart, rend, cstart, cend, cdata, dim)

i_fputbox (inzg, rstart, rend, cstart, cend, fdata, dim)
i_lputbox (img, rstart, rend, cstart, cend, ldata, dim)
i_iputbox (img, rstart, rend, cstart, cend, idata, dim)
i_sputbox (inig, rstart, rend, cstart, cend, sdata, dim)
i_cputbox (img, rstart, rend, cstart, cend, cdata, dim)

i_ffillbox (img, rstart, rend, cstart, cend, fval)
i_lfillbox (img, rstart, rend, cstart, cend, lval)
i_ifillbox (img, rstart, rend, cstart, cend, intval)
i_sfillbox (img, rstart, rend, cstart, cend, sval)
i_cfillbox (img, rstart, rend, cstart, cend, cval)

IMAGE *img;
int rstart, rend, cstart, cend, dim;
float *fdata, fval;
long int *ldata, lval;
int *idata, intval;
short int *sdata, sval;
unsigned char *cdata, cval;

Figure 2.15: The box access routines.
float av3x5 (img, row, col)
IMAGE *img;
int row, col;
{
    register int i, j;
    float box[3][5], sum;
    i_fgetbox (img, row-1, row+1, col-3, col+3, box, 5);
    sum = 0.0;
    for (i = 0; i < 3; i++)
        for (j = 0; j < 5; j++)
            sum += box[i][j];
    return (sum / 15.0);
}

Figure 2.16: A function which computes local average intensity.

2.10 Filling an Entire Image

The fillimg routines may be used to fill an entire image with a constant value. The routine calls' have two parameters: the image and the constant value. The routine declarations are shown in figure 2.18.

There are also multi-band fillimg routines which accept a vector of constant values, one value for each image band. Figure 2.19 contains their declarations.

2.11 Clearing Images

It is often useful to be able to clear a portion of an image before doing some form of display generation. The routines i_clrbox and i_clrimg, shown in figure 2.20, may be used to clear a rectangular region of an image or the entire image. I_clrbox accepts an image and the row and column co-ordinates of the region to be cleared. The selected portion of the image is filled with zero. I_clrbox operates on both single-band and multi-band images. I_clrimg is similar to i_clrbox

2The fillimg operations are actually macros which invoke the fillbox routines, but the user may view them as procedures.
Figure 2.17: The multi-band box access routines.
```c
i_ffillimg (img, fval)
i_lfillimg (img, lval)
i_liffillimg (img, ival)
i_sfillimg (img, sval)
i_cfillimg (img, cval)

IMAGE *img;
float fval;
long int lval;
int ival;
short int sval;
unsigned char cval;
```

Figure 2.18: The image fill routines.

```c
i_mfillimg (img, fvec)
i_mlfillimg (img, lvec)
i_miffillimg (img, ivec)
i_msfillimg (img, svec)
i_mcfillimg (img, cvec)

IMAGE *img;
float *fvec;
long int *lvec;
int *ivec;
short int *svec;
unsigned char *cvec;
```

Figure 2.19: The multi-band image fill routines.
2.12 Raster-Scan Access

Many operations on images can be expressed as a simple operation which is applied at each point in the image. For example, `multimg` involved multiplying every pixel by a constant. Other operations such as Sobel edge detection involve more complex computations but can still be expressed as an operation which is applied at each point of the image.

The usual way to apply a computation at each point in the image is to employ some sort of raster-scan over the image. The `apply` routines are a set of special-purpose routines which support raster-scan access to generalized images. The `apply` routines are optimized for a high degree of efficiency and programs which use them are often easier to understand than programs which have explicit code for raster scanning. For these reasons, the `apply` routines are often the best way to implement raster algorithms.

The simplest of the `apply` routines are `applyi` and `applyf`. Both of these routines provide for pixel-by-pixel processing of an input image to produce an output image. The images are passed as arguments to the routine. The declarations are in figure 2.21.

The first argument to `applyf` is `opf`, a `float` function which is defined in your program. `Out` and `In` are the output and input images respectively. `Verbose` is a Boolean parameter which determines whether or not `applyf` displays its progress as it processes the images. If `Verbose` is true, then `applyf` will display a digit on the user's terminal for every ten rows it processes. `Err` is the error handling parameter (see section 9). `Applyf` fetches each pixel from the input image in turn and passes it to your `opf` function. Your function must accept one `float` argument (being the input pixel value) and return a `float` function value. The value returned by your function is stored by `applyf` in the output image at the pixel location corresponding to the input pixel.

---

1. Strictly, `opf` is a pointer to a float function. In C, function pointers are passed by simply naming the function.
2. The digits are written to `stderr`. This is usually the user's terminal.
applyf (opf, out, in, verbose, err)
applyi (opi, out, in, verbose, err)

int (*opi)();
float (*opf)();
IMAGE *out, *in;
int verbose, *err;

Figure 2.21: The simplest apply routines.

The apply routine can be used to implement the multimg example neatly and efficiently, as in figure 2.22. The scaling constant is stored in a static variable (mult) because it cannot be passed as an argument to the scaling function (domult) that is applied to each pixel. This version of multimg has no rstart, rend, cstart and cend parameters. It does, however, have parameters verbose and err which it passes to apply. Err is the error handling parameter.

Applyi is similar to applyf but it uses integer pixel values. Op is a function which accepts an integer argument and returns an integer value.

The applyf and applyi routines process every pixel which exists in both the input and output images. i.e. the area of overlap between the two images. If the input and output images have different bounds, then the portions of the output image which have no corresponding input pixel will remain unchanged. Also, the portions of the input image which have no corresponding output pixel will not be processed.

In some situations it is useful to be able to restrict the processing to a specific region of the input image. Our early version of the multimg example had four parameters which specified the bounds of the rectangular region to be processed. The routines applyuf and applywi have a single parameter bounds which can be used to specify the bounds of a rectangular region to be processed. The routine declarations are in figure 2.23.

Bounds is a SUBIMAGE structure which contains four integers the row and column start and end bounds of the region to be processed. Applyuf can be used to implement a version of multimg which uses a SUBIMAGE structure to limit the affected area of the output image. The program in figure 2.24 is almost identical to the previous example, so a portion has been omitted.

There are other apply routines which support access to more than one input image and more than one output image. There are also apply routines which are useful for scanning small kernels over the input image, such as is required to implement Sobel edge detection. These are described briefly in the apply(3) manual entry.

25
# include <gimage.h>

static float mult;

static float domult (x)
float x;
{
    return (x * mult);
}

multimg (outimg, inimg, fmult, verbose, err)
IMAGE *outimg, *inimg;
float fmult;
int verbose, *err;
{
    mult = fmult;
    applyf (domult, outimg, inimg, verbose, err);
    return;
}

Figure 2.22: An apply version of multimg.

applywf (*opf, bounds, out, in, verbose, err)
applywi (*opi, bounds, out, in, verbose, err)

int (*opi)();
float (*opf)();
SUBIMAGE bounds;
IMAGE *out, *in;
int verbose, *err;

Figure 2.23: The windowed apply routines.
multimg (outimg, inimg, fmult, bounds, verbose, err)
IMAGE *outimg, *inimg;
float fmult;
SUBIMAGE bounds;
int verbose, *err;
{
    mult = fmult;
    applywf (domult, bounds, outimg, inimg, verbose, err);
    return;
}

Figure 2.24: Another version of multimg.

2.13 Copying Images

Image copying is an important part of the Generalized Image Library. Copying images is useful not only for moving data from one file or device to another, but also in combination with the virtual image operations described in section 5 for computing transformed images. The routines i-copy, icopyw and iscopyw provide efficient copying of image data from one image to another. The procedure calls are in figure 2.25.

i-copy copies pixels from fromimg to toimg in the area of overlap between the two images. If verbose is true, then progress marks are printed for every block of pixels transferred.

i-copyw copies a selected window of pixels from fromimg to toimg. Pixels are copied if they are in the area of overlap between the two images and also inside the specified subimage region.

i-scopyw copies a selected window of pixels from fromimg to toimg. During copying, the pixel co-ordinates are shifted by adding rowoffset to the row co-ordinate and coloffset to the column co-ordinate; i.e., the pixel at row i, column j of image fromimg will be copied to row i+rowoffset, column j+coloffset in toimg. The bounds of the portion to be copied, subimg, is taken relative to the image fromimg.
i_copy (toimg, fromimg, verbose, err)
i_copyw (toimg, fromimg, subimg, verbose, err)
i_scopew (toimg, fromimg, subimg, rowoffset, coloffset, verbose, err)

IMAGE *toimg, *fromimg;
SUBIMAGE subimg;
int verbose, rowoffset, coloffset;
int *err;

Figure 2.25: Image Copying Routines.
Chapter 3

Opening and Closing Images

Before the data in an image can be accessed, the image must be opened. The generalized image library provides two levels of facilities for opening images. The high-level open routines are described in section 3.1. They provide all the image naming facilities described in A User’s Guide to the Generalized Image Library. The low-level open routines are described in section 3.3. They each support access to a particular image format or device. The low-level open routines should not normally be used in writing programs since the high-level routines provide the same functionality with greater flexibility.

It is important to realize that the image access routines have exactly the same names and the same calling sequences irrespective of the way an image is opened. The implementations of the image access routines may differ from one image to another. However.

3.1 High-Level Single-Band Image Open Routines

There are two high-level single-band image open routines: i-open and i-creat [sic]. I-open is used to open a single-band image which already exists. I-creat is used to create a new single-band image.

The function declaration for i-open is in figure 3.1.

The parameters for i-open are name, mode, and err. Err is the error handling parameter. Its use is described in section 9. You will normally use NULL for the error handling parameter as is done in figure 3.3. Name is a pointer to a character string containing the image name to be opened. The image name may be any of the expressions described in A User’s Guide to the Generalized Image Library. An error will be detected if, for any reason, the named image cannot be opened. This will occur, for instance, if the image name refers to a file which does not exist or does not have the necessary permission bits set. It will also occur if a syntax error is detected in the name.

Mode is an integer which specifies the desired access to the image. Useful values of mode are
IMAGE *i_open (name, mode, err)

char *name;
int mode;
int *err;

Figure 3.1: The high-level single-band image open routine.

IMAGE *i_creat (name, pixtype, pixbits, bounds, err)

char *name;
int pixtype, pixbits;
SUBIMAGE bounds;
int *err;

Figure 3.2: The high-level image create routine.

IMREAD and IMMEDIATE. Mode IMREAD means that the image is opened for reading only. You should use mode IMREAD whenever you open an image which will be read but not modified. This allows your program to access images which you may not have permission to modify. Mode IMMEDIATE means that the image is to be opened for modifying. Use mode IMMEDIATE if your program will be making changes to the image.

If your program is not opening an existing image but rather creating a new image, then i_creat is the routine to use. i_creat always creates a new image. It will destroy an existing image of the same name unless the file is protected.

Figure 3.2 contains the procedure declaration for i_creat.

The parameters to i_creat are name, pixtype, pixbits, bounds and err. Name is the name of the image. Pixtype, pixbits and bounds specify the characteristics of the image to be created. Err is the error handling parameter.

The image name name is a pointer to a character string which contains the name of the image to be created. The image name may be any of the image name expressions described in A User’s Guide to the Generalized Image Library. An error will be detected if, for any reason, the image cannot be created.
typedef struct {
    int rs, re, cs, ce;
} SUBIMAGE;

Figure 3.3: Declaration of the SUBIMAGE structure.

Pixtype is an integer which specifies the pixel type of the new image. It must be one of the values: IM-UNSIGNED, IM-SIGNED or IM-FLOAT. Piirbits is an integer which specifies the number of bits required for each pixel in the new image. Bounds is a SUBIMAGE structure which specifies the first and last row and column indices of the new image. The SUBIMAGE structure is declared as shown in figure 3.3.

Although the parameters pixtype, piirbits and bounds are required, there is no guarantee that the image which is returned by i-creat will have exactly the desired properties. Depending on implementation details, the actual pixel type and bits per pixel may be different. If the image name refers to a physical device, the image bounds may be reduced to conform to the limits of the device. For example, consider an image being created on the hlatrox display. The hlatrox only supports 8 bit unsigned pixels, so the image will always have 8 bit unsigned pixels irrespective of pixtype and piirbits. The Matrox is also limited to 480 rows and 512 columns. so the created image will never have more then 480 rows or more than 512 columns. Your program should do something reasonable if i-creat returns an image which does not have the requested characteristics.

An example of the use of i-open and i-creat in a simple image copying program may be found in figure 9.1. The relevant portion is reproduced in figure 3.4.

I-open is used to open the input image. Since the image will only be read and not modified, mode is IMREAD. I-creat is used to create the output image. The pixel type, pixel bits and image bounds are taken directly from the input image. However, if the created image does not have the requested bounds, the program will act reasonably because i-copy only copies the pixels which are in the area of overlap between the input and output images.

I-open aid i-creat always return single-band images. This means that a multi-band device such as the Matrox is converted by software into a single-band device. It is therefore impossible to obtain color on the Matrox if it is opened by i-open or i-creat. To obtain color, the multi-band open routines must be used.
include <gimage.h>

main (argc, argv)
int argc;
char **argv;
{
  IMAGE *in, *out;
  ...
  in = i_open (argv[1], IM-READ, NULL);
  out = i_creat (argv[2], i_pixtype (in), i_pixbits (in),
                si_imglimits (in, NULL), NULL);
  i-copy (out, in, NULL);
  ...
}

Figure 3.4: Use of i_open and i_creat.

3.2 High-Level Multi-Band Image Open Routines

There are two high-level routines for opening multi-band images: imopen and imcreat. Imopen
opens an existing multi-band image and imcreat creates a new multi-band image.

Imopen has a similar calling sequence to i-open. The difference is the addition of the bands
parameter (figure 3.5). The bands parameter is a character string containing the bands specification
of the multi-band image to be opened. For example, figure 3.6 illustrates a program which opens
a color input image. The syntax of band specifications is described in A User's Guide to the
Generalized Image Library.

The bands parameter has two special meaningful values. If bands is NULL then the generalized
image library will open any arbitrary multi-band image. It is the user's responsibility to specify
the image hands using keywords such as color:. This is the method used in imgcp to achieve the
capability to copy arbitrary multi-band images. When bands is NULL, the generalized image library
assumes a single-band image unless the image name includes a bands specification as described in

The bands parameter may also be an empty string. In that case, the generalized image library
opens a single-band image exactly as if i-open had been called.

To assist programmers, some commonly used bands specifications have macro definitions. IM-COLOR
represents a standard color specification. The preferred method of opening a color image is shown
IMAGE *i_mopen (name, bands, mode, err)

char *name, *bands;
int mode;
int *err;

Figure 3.5: The high-level multi-band image open routine.

#include <gimage.h>

main (argc, argv)
int argc;
char **argv;
{
  IMAGE *in;
  in = i-mopen (argv[1], 'red,green,blue', IM-READ, NULL);
  ...
}

Figure 3.6: Opening a color image.
# include <gimage.h>

main (argc, argv)
int argc;
char **argv;
{
  IMAGE *in;
  in = i_mopen (argv[1], IM-COLOR, IM-READ, NULL);
  ...
}

Figure 3.7: The preferred method of opening a color image.

#define IM_SINGLE ''
#define IM-COLOR 'red,green,blue'
#define IM_STEREO 'left, right'
#define IM_CSTEREO n
  'left.red, left.green, left.blue, right.red, right.green, right.blue'

Figure 3.8: hlacros for useful multi-band specifications.

In figure 3.7, IMSTERO represents a standard stereo specification IM_CSTEREO represents a color stereo specification and IMSINGLE is defined as the empty string representing a single-band image. Figure 3.8 displays these macro definitions.

i_mcreat has a similar calling sequence to i-creat as shown in figure 3.9. Again, the difference is the addition of a bands parameter. This parameter has exactly the same interpretation for i_mcreat as it does for i-creat.

3.3 Low-Level Image Open Routines

The low-level image opening routines can be used directly to open images which are stored in particular formats or correspond to particular physical devices. However, using these routines severely restricts the flexibility of your program, so you should use i-open or i-creat instead
Figure 3.9: The high-level multi-band image create routine.

```c
IMAGE *i_mcrea (name, bands, pixtype, pixbits, bounds, err)
char *name, *bands;
int pixtype, pixbits;
SUBIMAGE bounds;
int *err;
```

Figure 3.10: The CMU image open routines.

```c
IMAGE *i_imgopen (name, mode, err)
IMAGE *i_imgcrea (name, pixtype, pixbits, bounds, err)
char *name;
int mode, pixtype, pixbits;
SUBIMAGE bounds;
int *err;
```

whenever possible. This section is included more in the interests of completeness than as an incentive to the programmer to use these low-level interfaces.

### 3.3.1 CMU Format Images

The routines *i_imgopen* and *i_imgcrea* open CMU format images. Their function declarations are in figure 3.10.

The parameters for *i_imgopen* are identical to the parameters for *i-open*, except that *name* must be the name of a disk file which contains a CMU format image. The parameters for *i_imgcrea* are identical to the parameters for *i-creat*, but *name* must be the name of a disk file which will be created as a CMU format image.
3.3.2 GIF Format Images

The routines `i-gifopen` and `i-gifcreat` open GIF format images. Figure 3.11 contains the function declarations.

The parameters for `i-gifopen` are identical to the parameters for `i-open`, except that `name` must be the name of a disk file which contains a GIF format image. The parameters for `i-gifcreat` are identical to the parameters for `i-creat`, but `name` must be the name of a disk file.

`i-gifpopen` and `i-gifpcreat` are used to open piped images. `i-gifpopen` opens the standard input channel `stdin` as a piped GIF format image. `i-gifpcreat` creates a piped GIF format image on the standard output channel `stdout`.

3.3.3 MIP Format Images

The routines `imipopen` and `imipcreat` open MIP format images. Their function declarations are in figure 3.12.

The parameters for `imipopen` are identical to the parameters for `i-open`, except that `name` must be the name of a disk file which contains a MIP format image. MIP format images always have 8 bit unsigned pixels, so `imipcreat` does not have `pixtype` or `pixbits` parameters. In addition, MIP format images have exactly 480 rows and 512 columns. If `bounds` has more than 480 rows or more than 512 columns, then the created image will not be as large as specified. If `bounds` is smaller than 480 rows by 512 columns, the unused portion of the MIP file will be filled with zeroes.
IMAGE *i_mipopen (name, niode, err)
IMAGE *i_mipcreat (name, bounds, err)

char *name;
int niode;
SUBIMAGE bounds;
int *err;

Figure 3.12: The MIP image open routines.

IMAGE *i_netopen (machine, name, kinds, mode, err)
IMAGE *i_netcreat (machine, name, bands, pixtype, pixbits, bounds, err)

char *machine, *name, *bands;
int mode, pixtype, pixbits;
SUBIMAGE bounds;
int *err;

Figure 3.13: The network image open routines.

3.3.4 Network Connections

The routines inetopen and inetcreat may be used to open images on remote machines using the network service. The declarations of the network open routines are in figure 3.13.

The parameter machine is a character string which contains the network name of the remote host on which the image is to be opened. Both inetopen and inetcreat communicate with the network server on the remote machine, and instruct it to open or create the image. Name is a character string which contains the name of the image on the remote machine. The image is opened remotely by calling i_mopen or i_mcreat as appropriate. Bands is a character string which contains a specification for the bands of a multi-band image (see A User's Guide to the Generalized Image Library). The parameters mode, pixtype, pixbits and bounds are identical to their counterparts in the i-open and i-creat routine calls.
IMAGE *i_matroxopen (board, err)
IMAGE *i_matroxcreat (board, bounds, err)

int board;
SUBIMAGE bunds;
int *err;

Figure 3.14: Routines which open the Matrox.

3.3.5 Matrox Display

The Matrox displays are multi-band devices each pixel has associated with it a red, green and blue intensity value. The Matrox open routines return multi-band generalized images. Figure 3.14 contains the function declarations.

The parameter Guard is an integer which indicates the Matrox board number of the Matrox to be opened. Currently, this parameter is ignored and is conventionally passed as zero. When more than one Matrox is attached to a single Sun, this parameter will be used to identify a particular Matrox.

I_matroxopen always returns an image which has 480 rows and 512 columns of 8 bit unsigned pixels. There is no mode parameter: the Matrox is always opened for both reading and writing. I_matroxopen does not destroy the image currently held in the Matrox. That image may be read if you so desire.

I_matroxcreat has no pixtype or pixbits parameters because the Matrox boards always have 8 bit unsigned pixels. In addition, the image bounds of the created image may not be as large as the requested bounds because the Matrox only supports 480 rows and 512 columns. If the requested bounds are smaller than 480 rows by 512 columns, the image will be centered on the screen. I_matroxcreat always clears the Matrox screen.

3.3.6 Datacube Display

The routines for opening the Datacube display are analogous to the routines for opening the Matrox display. The Datacube is treated as a color device which can accommodate up to 480 rows and 512 columns of 8 bit unsigned pixels. The routine declarations are shown in figure 3.15.
Figure 3.15: Routines which open the Datacube display.

\begin{verbatim}
IMAGE *i_cubeopen (board, err)
IMAGE *i_cubecreat (board, hounds, err)
int board;
SUBIMAGE bounds;
int *err;
\end{verbatim}

Figure 3.16: The image close routine.

\begin{verbatim}
i_close (img, err)
IMAGE *img;
int *err;
\end{verbatim}

3.4 Closing Images

Just as the pixel access routine calls are the same for every image, so every image is closed in exactly the same way. The routine \texttt{i-close} is the normal method of closing an image. \texttt{i-close} does everything that is necessary to close the image. It ensures that the image data is written out to disk and frees all the computer memory that was associated with the image. In addition, it will delete the image from disk if the image's \texttt{scr}p flag has been set.

The procedure declaration for \texttt{i-close} is in figure 3.16.

If your program creates a temporary image file and then wishes to delete it when it is closed, the routine \texttt{i-scr}ap may be used instead of \texttt{i-close}. \texttt{Is}crap sets the \texttt{scr}p flag on the image and calls \texttt{i-close}. Calling \texttt{Is}crap is equivalent to closing the image and then deleting the image file, but it is simpler and faster. The procedure declaration for \texttt{i-scr}ap is in figure 3.17.

In addition to \texttt{i-scr}ap there is a routine \texttt{i-setscr}ap which sets the \texttt{scr}p flag but does not close the image. If \texttt{i-setscr}ap is called on an image, then the image will be scrapped when it is later closed with \texttt{i-close}. \texttt{I}setscr\texttt{ap} is particularly useful when a temporary image is being created \texttt{i-setscr}ap may be called immediately after the image is created and the file will be deleted when the image is closed normally. The declaration for \texttt{i-setscr}ap is shown in figure 3.18.
(defun checkers ()
  ; Create the image file.
  (let ((img (icreat 'checkers.gif'
                   :unsigned 32 '(-100 100 -100 100) nil)))
    ; Store the pattern in it.
    (make-pattern img)
    ; Close the image
    (i-close img nil))
)

(defun make-pattern (img)
  (let ((r0 (i-rstart img))
        (c0 (i-cstart img)))
    ; Process all rows and columns.
    (dotimes (i (i-rows img))
      (dotimes (j (i-cols img))
        ; Put the value at each pixel.
        (putval img (+ i r0) (+ j c0))
      )))

(defun putval (img x y)
  ; Defines a checkerboard pattern.
  (if (evenp (* (round (/ x 10)) (round (/ y 10))))
      (i-putpixel img x y 0)
      (i-putpixel img x y 255)
  ))

Figure 12.16: Creating a pattern with the GIL Lisp interface.
i-scrap (inzg, err)

Figure 3.17: The image scrap routine.

i_setscrap (img, err)

Figure 3.18: Routine to set the image scrap flag.

3.5 Additional Notes

The following comments on i-close, iscrap and i_setscrap will be elaborated upon in later sections of this document. They are included here in the interests of completeness.

- If the image which is being closed is built upon other images, then those underlying images will also be closed. I-close is used to close the underlying images even if i-scrap was called to close the main image. This means that the underlying images will only be deleted if their scrap flags have individually been set. ¹

- If an image has been duplicated by i-dup, the physical image will not be closed until the last duplicate is closed.

- If i_setscrap or iscrap is invoked on any duplicate of an image then the physical image will be deleted when all the duplicates are closed.

¹This seems to be inappropriate semantics as it means that a multi-band temporary file image, which consists of i-multi applied to the individual files, is not able to be scrapped.
Chapter 4

Images and Matrices

Matrices as defined by the `matrix(3)` package are often useful in image understanding applications. A matrix is a convenient way of storing a temporary image, provided that the image is not excessively large. Matrices also provide fast, convenient access to the individual pixels by means of C language subscripting. This can be especially useful in implementing algorithms which have complex patterns of pixel access. and can result in significant speed improvements.

The generalized image library provides facilities to interface between matrices and images. The `i_dmat`, `i_fmat`, `i_imat` and `i_ucmat` routines take a matrix and return a generalized image which provides access to the matrix. This allows images stored in matrices to be passed to library routines such as `applyf`, `smooth`, etc. The bounds of the generalized image returned by `i_dmat`, `i_fmat`, `i_imat` and `i_ucmat` are the same as the bounds of the matrix. When the generalized image is closed the matrix is not freed. The function declarations are in figure 4.1.

The routines `ineudmat`, `ineufmat`, `ineuimat` and `ineuucmat` create a matrix and return a generalized image which gives access to the matrix. When the generalized image is closed, the matrix is freed. The `bounds` argument is a `SUBIMAGE` structure which contains the desired bounds of the image. Both the image and the underlying matrix will be created with the same bounds. Figure 4.2 contains the function declarations.

The routine `imatcreat` is a cover for the `ineufmat`, `ineuimat` and `ineuucmat` routines. It selects the appropriate type of matrix to create based on the requested pixel type and bits per pixel. The function declaration is in figure 4.3.
IMAGE *i_dmat (dmat, err)
IMAGE *i_fmat (fmat, err)
IMAGE *i_imat (imat, err)
IMAGE *i_UCmat (UCmat, err)

dmat dmat;
fmat fmat;
imat imat;
UCmat UCmat;

Figure 4.1: Generalized images from matrices.

IMAGE *i_newdmat (bounds, err)
IMAGE *i_newfmat (bounds, err)
IMAGE *i_newimat (bounds, err)
IMAGE *i_newUCmat (bounds, err)

SUBIMAGE bounds;
int *err;

Figure 4.2: Routines to create matrix images.

IMAGE *i_matcreate (pixtype, pixbits, bounds, err)

int pixtype, pixbits;
SUBIMAGE bounds;
int *err;

Figure 4.3: Routine to create a matrix image.
Chapter 5
Changing Images

The generalized image library contains a number of routines which accept one or more existing generalized images and create from them a new virtual generalized image. The original images are, in the process, closed so that your program can no longer access them. Although the routine i-dup may be used to keep some of the original images open if your program still needs to access them.

This section describes the routines which create new virtual images from old images. Section 5.1 is a discussion of the implementation of new virtual images from old. Section 5.2 describes the i-dup routine and how the original images can be preserved if necessary.

5.1 Discussion

In the previous sections we have seen that the generalized image library provides consistent access to many different types of images. We have seen that the library supports three different formats of images on disk and three different pixel types. It even supports image access over the network. In short, the generalized image library is very versatile.

The key to the library's versatility is the implementation of the image access routines. When your program is accessing a CMU format image with floating-point pixels, one set of image access routines is used. When the image is CMU format with signed pixels, another set of image access routines is used. And when the image is GIF format, still other image access routines are used.

All these different image access routines perform different computations. Yet they all have the same calling sequence and they all perform similar functions either storing or fetching pixel data so they all look the same to your program. In fact, any arbitrary computation can be performed by an image access routine provided that the calling sequence is correct and the routine does something which can be called 'storing' or 'fetching' pixel data.

However, the images are not physically closed, and the library is still able to access them internally.
For example, consider an image access routine which 'fetches' pixels from an image as follows. If the value of the actual pixel in the image is less than a constant, then the value 0 is returned. Otherwise, the actual pixel value is returned. When this image access routine is used to fetch data from an image, the pixels which are less than the constant will all be seen as 0. The remaining pixels will be unaltered. This procedure thus performs a type of thresholding on the image. The thresholding is done on the fly as the pixels are being used. This is called lazy evaluation.

As another example, consider an image access routine which 'stores' pixels in the following manner. If the pixel is zero, then the routine does nothing to the actual image. If the pixel is not zero, then the routine stores a white pixel value in the actual image. Thus, this image access routine makes the actual image white wherever it is told to store a non-zero pixel value. In particular, if it was given a rectangular array of values to store it would have the effect of producing a graphical overlay, making the actual image white wherever the value was logically true.

In both of these examples, we have assumed that there is an underlying image upon which our special image access routines operate. Although we could write such routines to operate specifically on a matrix or a Chili image, there is no reason at all why they should not operate on any generalized image. To achieve this, our special image access routine access routine would simply use the generalized image access routines on the underlying image. We can thus create a new or different generalized image which is built upon another generalized image.

This concept of a generalized image which is built upon one or more underlying generalized images is the basis of a very powerful set of library routines which take a generalized image and give you back another generalized image which is different but still refers to the same physical disk file or device new images for old.

### 5.2 Duplicating an IMAGE Descriptor

The IMAGE structure provides access to a physical image, device or virtual image. When an IMAGE structure is closed, that point of access to the image is no longer available. However, it is possible to duplicate the IMAGE structure and create another point of access. Then, when one of the IMAGE structures is closed, the other remains open. The physical image also remains open until the last duplicate structure is closed.

Duplicate structures must be created by the i-dup routine, which is described in figure 5.1. I-dup accepts an IMAGE structure and returns a new IMAGE structure which provides another point of access to the same image. There is no difference between the two points of access every putpixel, putrow, etc operation which is applied to one IMAGE structure will affect the data seen from the other IMAGE structure. However, either IMAGE structure can be closed without closing the other IMAGE structure or affecting it in any way.

The most important use of i-dup is when a virtual form of an image is to be created, but access to the original image is also required. For example, figure 5.3 shows the use of i-dup when creating...
IMAGE *i_dup (img, err)

IMAGE *img;
int *err;

Figure 5.1: Routine to duplicate an IMAGE structure.

IMAGE *i_crop (img, bounds, err)

IMAGE *img;
SUBIMAGE bounds;
int *err;

Figure 5.2: Routine to crop an image.

A cropped version of an image. The image in this case is the display, and the program wishes to retain access to the whole display. Therefore, instead of passing the display image to i-crop, the program uses i-dup to create a duplicate image which is passed to i-crop. The duplicate image is cropped and the display image remains untouched.

5.3 Cropping an Image

One of the simplest ways to change an existing generalized image into something new is to crop it. The i-crop routine accepts a generalized image and a set of bounds. It returns a new generalized image which is the same as the original image, but only provides access to the cropped portion of the image. There is no additional execution time involved in accessing a cropped image, but there is no protection to ensure that your program will not access outside the selected image bounds. The function declaration is in figure 5.2. Figure 5.3 is an example of the use of i-crop. In this example, a window is being cropped from the display device.
# include <gimage.h>

IMAGE *display, *window;
SUBIMAGE uinbounds;
display = i-open ('display', IM_MODIFY, NULL);
winbounds.rs = 0; winbounds.re = 239;
winbounds.cs = 0; winbounds.ce = 255;
window = i_crop (i_dup (display, NULL), uinbounds, NULL);
...

Figure 5.3: Cropping a window of the display

IMAGE *i_shift (img, rstart, cstart, err)

IMAGE *img;
int rstart, cstart;
int *err;

Figure 5.4: Routine to shift an image.

5.4 Shifting an Image

The i_shift routine may be used to create a virtual image which has a different origin than the original image. The resulting image will have the same size as the original image, but will have a new origin. The image is shifted entirely in software when your program performs an image access operation based on the new co-ordinates. The library will compute the corresponding old co-ordinates and access the original image. This imposes a small additional cost on each image access.

The function declaration for i_shift is in figure 5.4. img is the original input image which is closed by i_shift so that your program can no longer access it. The routine i_dup should be used if your program also requires access to the unshifted image. Rstart and cstart are the row and column origin co-ordinates for the shifted image. Err is the error handling parameter, usually NULL.
5.5 Simple Linear Transformation

The routine iltrans creates a virtual image which is a linear transformation of an underlying image. When pixels are fetched from the virtual image, the pixel values of the underlying image are fetched and transformed. When pixels are stored into the virtual image, the inverse linear transformation is applied and the converted values are stored in the underlying image. Iltrans can thus be used to adjust the value range of pixels in an image. This is useful, for example, when a 1 bit image is being copied to an 8 bit display device. The function declaration for iltrans is in figure 5.5. The floating-point arguments multiply and add are the parameters of the linear transformation. When pixels are fetched, they are first multiplied by multiply and then have add added to them. When pixels are stored, they first have add subtracted from them and then divided by multiply. It is an error if multiply is zero.

As with the other routines for creating virtual images, the input image img is closed and cannot be accessed after the call to iltrans. I_dup can be used to maintain access to the untransformed image if this is necessary.

5.6 Multi-band Image from Single-band Images

The routine imulti can be used to combine a number of single-band images into a single multi-band image. A typical use of this facility is to create three matrices representing the red, green and blue bands, then convert them into generalized images and finally combine them into a color image. The function declaration for imulti is shown in figure 5.6.

The parameter inlist is a pointer to a vector of IMAGE pointers. The vector contains the images which are to be included in the multi-band image, in their correct sequence. The vector is terminated with a NULL pointer. The parameter bandspec is a character string containing the multi-band specification. Figure 5.7 shows the construction of a multi-band color image from three single-band images. IM-COLOR is the standard macro for the band specification of a color image, described in section 3.2.
IMAGE *i_multi (imglist, bandspec, err)

IMAGE **imglist;
char *bandspec;
int *err;

Figure 5.6: Constructing a Multi-band Image.

IMAGE *red, *green, *blue, *rgb;
IMAGE *imglist[4];
...
imglist[0] = red;
imglist[1] = green;
imglist[2] = blue;
imglist[3] = NULL;
rgb = i-multi (imglist, IM-COLOR, NULL);

Figure 5.7: Example of the use of i-multi.
5.7 Converting Multi-Band Images

Certain types of multi-band images can be converted into other multi-band images in obvious ways. For example, a color image can be converted to a single-band image by computing the average intensity of the red, green and blue components. The routines `i_avbands` and `i_threeed` provide simple conversions between types of multi-band images. Their function declarations are in figure 5.8.

`i_avbands` accepts a multi-band image and returns a single-band image which is the average of the bands of the multi-band image. The averaging is performed on the fly. When a pixel is fetched from the single-band image, the corresponding pixel in the multi-band image is fetched and the average value is computed and returned. When a pixel is stored to the single-band image, the same value is stored in all the bands of the underlying multi-band image. `i_avbands` is used by the library to provide black-and-white access to color display devices such as the Matrox and Datacube.

`i_threeed` can be used to generate three-dimensional displays. The argument to `i_threeed` is a color image. The returned image is a stereo image. When pixels are stored into the stereo image, the right band value is assigned to the red band of the color image and the left band value is assigned to the blue and green bands. This produces a color image which can be viewed with red/blue stereo glasses.

5.8 Magnification and Reduction

The routines `i_magnify` and `i_xmagnify` create virtual images which are magnifications or reductions of an underlying image. Magnification is performed by replicating pixel values: reduction is performed by simple resampling. When pixels are fetched from a magnified image, the underlying

\[ \text{IMAGE } *i\text{-avbands} \ (img, \ err) \]

\[ \text{IMAGE } *i\text{-threeed} \ (img, \ err) \]

\[ \text{IMAGE } *img; \]

\[ \text{int } *err; \]

Figure 5.8: Routines to Convert Multi-Band Images.
Figure 5.9: Magnification Routines.

Pixel values are replicated as necessary to provide pixel values for the virtual image. When pixels are stored to a magnified image, the supplied pixel values are resampled at fixed co-ordinates and stored to the underlying image. Some co-ordinates in the magnified image have no corresponding position in the underlying image; values for these co-ordinates are discarded.

Pixels are fetched from a reduced image by resampling the underlying image at fixed co-ordinates. Storing pixels to a reduced image involves replication of the supplied pixel values.

Figure 5.9 shows the function declarations for both imagnify and ixmagnify. Imagnify provides for magnification or reduction of an image. The parameter img is the image to be magnified or reduced. Num and denom are the numerator and denominator respectively of a rational number which is the magnification factor to be applied. For example, values of 2 and 2 for num and denom respectively can be used to indicate a magnification of 2/3, which is, of course, a reduction. The magnified image will have the same origin as the underlying image.

Ixmagnify provides additional capabilities not provided by imagnify. These extended facilities are provided by the additional parameters. Numr and denomr are the numerator and denominator of a rational number which is the magnification factor to be applied to rows of the image. Numc and denomc represent the magnification factor which is applied to the columns of the underlying image. It is thus possible to change the aspect ratio of an image. Rs and cs specify the origin of the magnified image. Options is intended to be used to implement alternative methods of magnification (such as anti-aliasing and pixel interpolation). The appropriate value is zero for the default method of pixel replication and resampling.

5.9 Image Quantization and Halftoning

The routines ihalftone and icquantize are used for halftoning and color quantization respectively. This is halftoning or color quantization which is done on the fly, enabling a binary display
device to be used for grey-level display or an eight-bit frame buffer to be used as a color image.

`ihalftone` accepts a binary image (such as a binary Sun screen display) and creates a virtual eight-bit image. When pixels are stored to the virtual image, error propagating halftoning is performed on the fly and the binary image is updated. The results of this halftoning process are 'ideal' if the pixels are written in a top-left to bottom-right sequence. In particular, i-copy produces the best possible results. The quality is slightly reduced if the pixels are written randomly.

The first parameter to `ihalftone` is the binary image on which the halftoning is to be performed. The next three parameters provide control over the algorithm. `prop` and `overlap` control the recomputation of the halftoning when a pixel is changed. The default values are used for these parameters if zero is passed. The default values are eight and four respectively. `options` is used for special halftoning options. Zero is the normal value. The only special option value is one which indicates Sun screen halftoning.

The function declarations for `ihalftone` and `icquantize` are in figure 5.10.

### 5.10 Windowing Facilities

A window image is a special type of generalized image. Window images are generalized images which can be used as viewing windows to display the contents of another image. For example, a window image which refers to the display could be attached to an image within a program. The display would then monitor the contents of the image to which it was attached.

The following sections describe the routines which provide the windowing facilities of the generalized image library. Throughout these sections, the term `window` refers to a window image returned by `i_window`.

---

5 Development of `icquantize` is incomplete, so it is not fully described here.
5.10.1 Creating a Window Image

I-window accepts a generalized image and converts it into a window image. This involves creating a virtual image which provides the additional functionality required for its use as a window. A small overhead is introduced into each image access operation. The original image is closed and cannot be accessed unless i-dup is used. The function declaration of i-window is shown in figure 5.11.

Window images have the following special capabilities.

- Windows can be panned, i.e. Their origin co-ordinates as seen from within the program can be changed. The position of the window on the physical display device cannot be changed; rather, the portion of a larger image which is seen on the display is changed.
- Windows can track changing image co-ordinates.
- Windows provide full protection against writing outside the image bounds. Attempts to write pixels outside the visible bounds of the window are either ignored or result in automatic panning. In contrast, a cropped image (i-crop) does not prevent an erroneous program from writing outside the cropped bounds of the image, possibly destroying other image data or causing the program to crash.
- Attempts to read pixels outside the visible bounds of the window result in undefined data (but do not crash the program) and may cause automatic panning.
- Automatic panning can be selectively enabled and disabled. It is disabled by default.
- Windows can be attached to other images and detached from them.

5.10.2 Panning and Tracking Windows

The routine i-pan (see figure 5.12) can be used to explicitly pan a window image. The argument wimg must be a window image returned from i-window, or an error will be detected. The parameters
Figure 5.12: Window Panning and Tracking Routines.

\texttt{i_wpan (wimg, rc, cc, err)}
\texttt{i_wtrack (wimg, rc, cc, err)}
\texttt{i_wtrackbox (wimg, rs, rc, cs, cc, err)}

\begin{verbatim}
IMAGE *wimg;
int rc, cc, rs, re, cs, cc;
int *err;
\end{verbatim}

\texttt{rc} and \texttt{cc} specify the desired co-ordinates of the centre of the window. After a successful call, a pixel stored at row \texttt{rc}, column \texttt{cc} would normally appear in the centre of the window image. However, if the window is attached to an image then \texttt{i_wpan} does not allow a window to be panned beyond the bounds of the image to which it is attached.

\texttt{i_wpan} moves the origin of the window image but it does not move the data which is already stored in the window image. If the window is attached to an image then it will be refreshed by copying the corresponding pixels from the image to the window. Thus, explicit panning of unattached window images is not very useful but window which are attached to an image can usefully be panned.

Explicitly panning windows can result in time-consuming data copying as the window is moved by small increments. The routines \texttt{i_utrack} can be used to cause a window to track a moving point. The argument \texttt{wimg} is the window image which is to track the point. The arguments \texttt{rc} and \texttt{cc} specify the current co-ordinates of the moving point. Each time \texttt{i_utrack} is called, the point is compared with the window bounds. The window is panned as necessary to ensure that the point is visible and is not too close to the edge of the window.

In cases where it is desired to track an area of several pixels, the alternative routine \texttt{i_utrackbox} can be used. \texttt{i_utrackbox} ensures that the entire box specified by \texttt{rs}, \texttt{rc}, \texttt{cs} and \texttt{cc} is visible within the window. It also ensures that the centre of the box is not too close to the edge of the window.

\textbf{5.10.3 Attaching Windows to Images}

Window images are most useful when they are attached to other images. The window image then becomes a moveable window for viewing the contents of the other image. This is useful for displaying the results of tracking-type algorithms (such as road tracing in aerial images) and for

\hspace*{1cm} 'A border approximately $1/8\text{th}$ of the size of the window is maintained and the tracked point is kept out of that border as much as possible.'
Figure 5.13: Window Attachment Routines.

cursor positioning at high resolution. The routine `i_attach` (see figure 5.13) attaches a window to an image. The argument `img` is the image to which the window `wimg` is attached. The initial position of the window on the image is specified by the `pan` co-ordinates `rc, cc`. Attaching a window to an image involves some additional overhead in pixel accesses to the image. If the window is later detached, that overhead is removed completely.

A window can only be attached to one image at a time. If an attempt is made to reattach a window which is already attached to some image, the first attachment is automatically broken and the new attachment is made. However, many windows may be simultaneously attached to one image. This is useful when separate windows are tracking different computations.

The routine `i_detach` may be used to explicitly detach a window from its attached image. Windows are also detached automatically when they are closed. Windows are generalized images so they are closed in the normal manner with `i_close`.

### 5.10.4 Window Options

The routine `i_option` can be used to selectively enable and disable optional features of window images. The argument `wimg` is the window for which the options are to be adjusted. The argument `option` is an enumerated constant which indicates the specific option to be changed. `Value` specifies the desired new value of the option. `Err` is the error handling parameter. The allowable values of `option` and their meanings are as follows.

- **IM.WPANGET**: Pan on image data get operations. `Value` is a Boolean quantity. If it is true (i.e., non-zero) then the window will be automatically panned as necessary so that it tracks the location of image data get operations: i.e., the `getpixel`, `getrow`, `getcol` and `getbox` operations.

This option is used to automatically track the data which is being read into a program. Automatic panning is only effective when the window is attached to an image.
IM_WPANPUT: Pan on image data put operations. Value is a Boolean quantity. If it is true then the window will be automatically panned as necessary so that it tracks the location of image data put operations: i.e. the putpixel, putrow, fillrow, putcol, filrcol, putbox and fillbox operations. This option is used to automatically track the output of a program. Automatic panning is only effective when the window is attached to an image.

IM_WPANSAVE: Save pan options. Value is ignored. When iwoption is called with this parameter, the current automatic panning option values are saved and the default parameters established. Saves and restores of the pan options can be nested up to 30 deep.

IM_WPANRESTORE: Restore pan options. Value is ignored. The previously saved pan parameters are restored.
Chapter 6
Operations on Images

6.1 Miscellaneous Macros

The gimage.h include file provides a number of general-purpose macros which can be used in your program. These macros provide simple services such as fetching fields of the IMAGE structure, displaying progress marks or checking the validity of an IMAGE structure.

The macros i-start, i-end, i-cstart and i-cend may be used to obtain the individual bounds of an image. The macros i-rows and i-cols determine the number of rows and columns in the image. i-pixtype and i-pixbits return the pixel characteristics of the image, and i-filename returns the file name of an image on disk. These macros should not be used to store information in the image. The function declarations for these macros are shown in figure 6.1.

The macro i-progress displays progress marks. It prints the hundreds digit of its argument whenever the argument is a multiple of ten. Typically, i-progress is used to report progress as rows of an image are processed. The progress marks are displayed on the standard error channel stderr so that they do not interfere with piped images. An example of the use of i-progress is contained in figure 6.2. This example is a routine which computes the average intensity of an image and displays progress marks as it processes the rows of the image.

The macros i-check and imcheck may be used to check the validity of an IMAGE structure. Imcheck attempts to ensure that the IMAGE structure is valid by checking a special field of the structure. I-check performs the same function but additionally ensures that the image is a single-band image. Therefore, you should use i-check if your routine expects single-band images and imcheck otherwise. Both of these macros also invoke a cyclic redundancy check which attempts to detect corruption of the IMAGE structure.

The arguments to i-check and imcheck are the image, the name of your routine and the error handling parameter. If the image does not pass the test, both i-check and imcheck invoke error handling. They also return boolean true when an error is detected and boolean false when no error
is detected. Of course, the macros do not even return to your routine if the error results in an abort. This allows your program to use the check macros in an if statement and simply return if the macro value is true. See figure 6.2.

6.2 Miscellaneous Routines

This section describes additional routines which are provided by the generalized image library. These operations can be applied to any image, although they may not do anything useful depending on the capabilities of the display device or file which underlies the image. As is the case for the image access operations, the actual computation performed by the routines described in this section may vary from image to image.

6.2.1 Property List Management

A property list is a list of property/value pairs which are associated with an image. Both the property and the value are character strings.

The property list can be used to store arbitrary information about the image. The value may be C string, including the empty string. The property name may not be the empty string. The following property names have special meanings: pixel type, title and description.

The property list is stored in the image file along with the image data.1 Five routines are

1CMU and GIF format image files are capable of storing the property list. MIP format images are unable to store
# include <gimage.h>

float imgaverage (img, verbose, err)
IMAGE *img;
int verbose;
int *err;
{
    float sum = 0.0;
    register int r, c;
    if (i-check (img, 'imgaverage', err))
        return (0.0);
    for (r = i_rstart (img); r <= i_rend (img); r++)
    {
        if (verbose)
            i-progress (r);
        for (c = i_cstart (img); c <= i_cend (img); c++)
            sum += i-fgetpixel (img, r, c);
    }
    return (sum / ((float) (i_rows (img) * i_cols (img))));
}

Figure 6.2: A simple example using some macros.
int i_getprop (inzg, property, outvalue, err)
i_putprop (inzg, property, value, err)
i_delprop (img, property, err)
i_getproplist (img, props, vals, err)
i_cpyproplist (toimg, fromimg, err)

IMAGE *img, *toimg, *fromimg;
char *property, *value, outvalue[IM_MAXPROP];
char ***props, ***vals;
int *err;

Figure 6.3: The property list management routines.

provided for manipulating the property list: i_getprop, i_putprop, i_delprop, i_getproplist
and i_cpyproplist. The function declarations for these routines are shown in figure 6.3.

i_getprop fetches a property value from an image. If the requested property is not found in
the property list of img, then i_getprop returns Boolean true. Failure to find property in the
property list is not an error condition errors are detected if there are problems when accessing the
property list or if the arguments to i_getprop are invalid.

If property is found, then i_getprop returns Boolean false. The property value string associated
with property is copied to the string value, which should be large enough to receive the result. The
library silently limits property values to IM_MAXPROP-1 characters. If i_getprop is being called
only to check for the presence or absence of a property, then value may be NULL and no attempt
will be made to return the property’s value.

i_putprop stores a property and value pair in the property list of inzg. Property is the name of
the property it must be a non-empty character string. Value is the value string to be associated
with property. Value may be any character string including the empty string. If NULL is passed for
value then the empty string will be stored as the property value. The value string may not exceed
IM_MAXPROP-1 (4095) characters.

i_delprop deletes properly and its associated value from the property list. It is not an error if
property is not found in the property list.

i_getproplist returns all the property/value pairs in an image’s property list. When the
routine returns. props and vals are set to point to NULL-terminated arrays of pointers to strings.
tie property list, so it is lost when the image is closed.

"This counter-intuitive convention is inherited from ages past and it is too late to correct it."
```c
#include <gimage.h>

main (argc, argv)
  int argc;
  char **argv;
{
    char **props, **vals;
    IMAGE *img;
    int i;
    img = i_open (argv[1], IM-READ, NULL);
    i-getproplist (img, &props, &vals, NULL);
    i_close (img, NULL);
    for (i = 0; props[i] != NULL; i++)
      props[i], vals[i] = printf ('%s' = '%s
exit (0);
}
```

Figure 6.4: A program to display the property list.

The strings which are pointed to are the original copies in the image so they must not be altered in any way. The arrays of pointers themselves are created by `malloc(3)` so you should `free` them when you are done with them. Figure 6.4 displays an example of the use of `i-getproplist`. The example is a simple program which prints the property list of an image on the terminal. Note that the arguments to `i-getproplist` are actually the addresses of the variables which are set to point to the NULL-terminated arrays.

`i-cpyproplist` copies the property list of one image to another. All the properties and values of fromimg (except for pixel type and its value) are copied to toimg. The existing properties of toimg are not deleted by `i-cpyproplist`.

For historical reasons, the pixel type information for CMU format images is stored as the value of the pixel type property. For this reason, both `i-getproplist` and `i-cpyproplist` ignore the pixel type property. You should never manipulate the pixel type property directly.

The properties title and description are used when printing images. The value of the property title is printed centred above the image. The value of the property description is printed left justified under the image. The value strings may contain newline characters which are correctly handled when the image is printed. The example in figure 6.5 demonstrates the use of `i-putprop` to set the title and description of an image to be printed. The title is set by copying the property list of the input image to the output image. The description is generated by the program and reflects the processing which it will perform.
# include <gimage.h>
# include <sys/time.h>

main (argc, argv)
int argc;
char **argv;
{
    IMAGE *inimg, *outimg;
    long now;
    char describe[512];
    float delta;

    /*
    * Open input image. Create output image.
    * Parse delta parameter.
    */
    inimg = i_open (argv[1], IM_READ, NULL);
    outimg = i_creat (argv[2], IM_UNSIGNED, 8,
                     si_imglimits (inimg, NULL), NULL);

    /*
    * Create descriptive text including processing
    * parameter and date string (for BSD 4.2).
    */
    now = time (0);
    i_cpyproplist (outimg, inimg, NULL);
    sprintf (describe, 'Texture image with delta %g
        delta, asctime (localtime (&now));
    describe. NULL); i_putprop (outimg, 'description...}

Figure 6.5: Setting the Descriptive Text.
i_live (img, camera, err)
i-grab (img, err)
i_set_camera (img, gain, offset, err)

Figure 6.6: Digitization Operations

6.2.2 Digitizing Images

Before considering using the digitization routines described in this section you should be familiar with the image sequence facilities described in section 8. The image sequence abstraction provides for digitization and also for processing of canned image sequences. If your application is a repetition of digitize and processing steps then image sequences will probably meet your requirement; and may provide unexpected benefits.

Three extended operations routines are provided for using the digitizing facilities of display devices such as the Matrox or Datacube. The routine i_live causes the device to display the current live image coming in from the camera. The three parameters of i_live are the image which is to become live, the camera number from which input is to be taken and the error handling parameter. The img parameter may be any generalized image which refers, either directly or indirectly, to a physical display device with digitization capabilities. This means that img may be a virtual image based on a physical display device; for example, a scaled version of the Matrox display. Img may also be a network image representing a physical device on another machine. The network facilities of the generalized image library are capable of handling the i_live request. It is important to realize that i_live causes all the data currently held in the Matrox to be lost. The effects of i_live are not restricted to a cropped region even if it is applied to a cropped image. I_live has no effect if the image does not refer to a physical display device capable of digitizing.

The routine i_grab may be called after i_live to grab the live image and freeze it in the frame buffer. The camera gain and offset parameters may be adjusted (prior to calling i_grab) by calling i_set_camera. Reasonable defaults are selected for the camera parameters when the device is opened. The procedure declarations for the digitization routines are in figure 6.6.

A higher-level interface for digitizing images is provided by the routine i-digitize. The declaration of i_digitize is in figure 6.1. The parameter img is a generalized image. I-digitize has no effect if img does not refer, either directly or indirectly, to a display device with digitization ca-
i_digitize (img, camera, internet, err)

IMAGE *img;
int camera, interact;
int *err;

Figure 6.7: A Higher-level Digitization Routine.

IMAGE *i_temp (pixtype, pixbits, bounds, err)

int pixtype, pixbits;
SUBIMAGE bounds;
int *err;

Figure 6.8: Routine to Create a Temporary Image.

digitize invokes i_live and i_grab to digitize an image. It also provides interaction facilities to enable the user to set the camera parameters.

Camera is the number of the camera to use, normally zero. Interact is an interaction flags. If internet is Boolean true (iioii-zero) then i_digitize enters a simple command interpreter after activating the live image. The command interpreter allows the user to adjust the camera parameters and to select the appropriate moment to grab the image. The user can also change the camera number. If interact is Boolean false (zero) then there is no interaction and an image is digitized immediately.

6.2.3 Temporary Images

Some algorithms require temporary images to store intermediate results. I_temp creates a temporary image with selected pixel characteristics and image bounds. The routine declaration is in figure 6.8. I_temp first attempts to create the temporary image as a matrix in memory. If that is not successful then it attempts to create a CMU format image in temporary disk space. An error is returned if a temporary image cannot be created at all. If a CMU format image file is created, then it will be scrapped when the image is closed.

If your program needs to create a temporary image file to be passed to another program, you
**6.2.4 Cursor Positioning**

A User's Guide to the Generalized Image Library describes the cursor positioning facilities of the generalized image library. These facilities are implemented by the routine `igetpos`. The procedure declaration of `igetpos` is shown in figure 6.11. This routine is still under development, so this section of the manual is incomplete.
i_getpos (img, row, col, err)

IMAGE *img;
int *row, *col;
int *err;

Figure 6.11: Cursor Positioning Routine.

i_imgsoft (size)
i_imgfdlimit (nfiles)

int six, nfiles;

Figure 6.12: Routines to control the CMU image module.

6.2.5 Manipulating Subimages.

6.2.6 Controlling the CMU Image Module.

The module of the library which accesses CMU format images provides two special-purpose routines which can be used to control the operation of the module. i_imgsoft controls the hardware and software paging features of the module.\footnote{Software paging is implemented only on the Vax at this time, so i_imgsoft does nothing on the Suns.} i_imgfdlimit controls the use of file descriptors by the module.

Figure 6.12 contains the declarations of both i_imgsoft and i_imgfdlimit. i_imgsoft accepts an integer parameter which may be any non-negative value. This parameter specifies an image size in kilo-bytes. Images which are smaller than the specified size will be hardware paged if possible. Images of the given size or larger will always be software paged. The library will print a warning message if software paging is selected and the threshold specified was greater than zero.

i_imgsoft is particularly useful when programs are being written which access only a few pixels of a very large image. In such situations, the cost of reading the entire image far exceeds the cost of using software paging to access the needed pixels. and it is appropriate to select a meaningful software paging threshold.

i_imgsoft can also be used as a crude way of preventing the library from using all the available
dynamic memory for storing large images and not leaving enough space for your program to operate. However, the control provided by \texttt{i-imgsoft} is very crude and has no effect on the use of memory by modules which access images other than CMU format images. It is much better to allocate the buffers which your program needs before opening the images if possible.

\texttt{i-imgfdlimit} sets a limit on the number of files which the CMU image access module will have open at any given time. This can be useful if your program tends to run out of file descriptors.

Neither \texttt{i-imgsoft} nor \texttt{i-imgfdlimit} have any effect on the library except with respect to CMU format images. They may be called whether or not the program is actually operating on CMU format images. It is therefore unnecessary to attempt to force the user of your program to use CMU format images, and it is unnecessary for your program to check whether or not CMU format images are involved.

\section{Library Environment Options}

The generalized image library uses a number of environment variables to control its action. The \texttt{IMDEBUG} environment variable controls debugging checks and information. The \texttt{IMSYNC} environment variable controls whether physical display devices are internally or externally synched. Normally, these options are selected by the user prior to invoking your program. However, some interactive programs may wish to provide the user with the capability to change these options during execution of the program. The routines \texttt{i-getoption} and \texttt{i-putoption} provide an interface to the generalized image library options. The function declarations are in figure 6.13. \texttt{i-getoption} accepts a character string \texttt{optionname} and returns a pointer to the string value associated with the specified option. The pointer will be \texttt{NULL} if the option has no defined value or, equivalently, if the value is the empty string. \texttt{i-putoption} associates value with the option \texttt{optionname}. It should be noted that changing an option is not guaranteed to have immediate effect. Changes to \texttt{IMDEBUG} start to take effect immediately but may only affect new \texttt{IMAGE} structures. Changes to \texttt{IMSYNC} only affect display devices opened after the change is made. \texttt{i-putoption} has no effect on the system environment, either in subprocesses or in the parent shell process.
char *i_getoption (optionname)
i_putoption (optionname, value, err)

char *optionname, *value;
int *err;

Figure 6.13: Generalized Image Library Options Routines.
Chapter 7

Simple Graphics

Simple graphics are useful for visually inspecting the processing of a program. For example, in a segmentation program, these routines may be used to outline and label regions in an image. Similarly, a path planning program may draw an overhead projection of all of the obstacles in the environment and then draw the resulting path plan. Both of these displays can be generated by using the following graphics routines for generalized images.

Graphics are drawn on a generalized image in the same manner as graphics are drawn on a raster display device by coloring individual pixels. An advantage of drawing on a generalized image rather than directly using the display device is that a generalized image allows more flexibility. The generalized image can be the display device or it may be a file which can be used to save results shown with graphics. This file can be printed later for a hardcopy of the results.

7.1 Line and Figure Drawing Routines

The basic graphics function is the line drawing function. Many graphical entities, such as circles, can be drawn as series of lines. The line drawing routine shown in figure 7.1 is used to draw a specified line segment of a specified color on a generalized image. Lines can be drawn on both single-band and multi-band images.

Line is a generic routine which accepts pixel data in any of the five forms normally accepted by the Generalized Image Library. Img is the image on which the line segment is to be drawn. The location of the line segment is represented by its endpoints which are specified by row and column coordinates in the image. R1 and c1 specify one endpoint and R2 and c2 specify the second endpoint. Type is an enumerated value which indicates the data type of data. Type is one of the following: IM_FLOAT, IM_LONG, IM_INT, IM_SHORT or IM_CHAR. Data is a pointer to a vector of pixel values. One for each image band of the type indicated by type. For drawing lines on single-band images, the address of a variable of the appropriate type may be passed as in figure 7.2.
i_line (inzg, r1, c1, r2, c2, type, data)

IMAGE *img;
float r1, c1, r2, c2;
int type;
generic-pointer data;

Figure 7.1: Line Drawing Routine.

The line drawing routine automatically clips the line segment to fit within the bounds of the image given in the first argument. Therefore, the user need not worry about writing outside the bounds of the image.

A simple rectangle drawing function is also provided. This routine draws horizontally oriented rectangles on the image by drawing the appropriate four sides. The declaration of i_rectangle is shown in figure 7.3. It is similar to i_line and uses the same method of handling the different pixel data types supported by the Generalized Image Library.

Img is the image on which the rectangle is drawn. The next four parameters represent two diagonal corners of the rectangle. The rectangle is created by drawing lines from (r1, c1) to (r1, c2) to (r2, c2) to (c2, r2) and back to (r1, c1). The color or intensity of the lines in the rectangle is given by the parameter data which is a generic pointer. The type of the data is indicated by type.

I-circle is provided for drawing circles. It is described in figure 7.4. Img is the image on which the circle is to be drawn. Rowncentre and colcentre are the co-ordinates of the centre of the circle. Radius is the radius. Type is the type indicator for data. The drawn circle is automatically clipped to the size of the image.

The graphics routines can draw lines of varying thicknesses on a generalized image. To set the width of the graphics to be drawn, the i_set_width routine, shown in figure 7.5, is used.

The routine has two parameters: the image where the width is to be altered and the width in pixels of the all graphics to be drawn. All successive calls to the graphics routines for this particular image will draw lines with the width set by this subroutine. When an image is opened, the width is automatically set to one. When an image is duplicated by i-dup the line width of the duplicate image is the same as the line width of the original image. However, the line width for the duplicate image may then be changed without affecting the line width for the original image. Similarly, changing the line width for the original image after calling i-dup will have no effect on the line width for the duplicate image.
# include <gimage.h>

main (argc, argv)
int argc;
char **argv;
{
    IMAGE *display;
    unsigned char color;
    float fcolor;
    /*
    * Open the display.
    */
    display = i_create ('display', IM_UNSIGNED, 8,
        si-subimage (0, 479, 0, 511, NULL), NULL);
    /*
    * Draw a couple of white lines.
    */
    color = 255;
    i_line (display, 0.0, 0.0, 500.0, 500.0, IM-CHAR,
        (generic-pointer) (&color));
    i_line (display, 0.0, 500.0, 500.0, 0.0, IM-CHAR,
        (generic-pointer) (&color));
    /*
    * Draw gray lines.
    */
    fcolor = 128.0;
    i_line (display, 250.0, 0.0, 250.0, 500.0, IM-FLOAT,
        (generic-pointer) (&fcolor));
    i_line (display, 0.0, 250.0, 500.0, 250.0, IM-FLOAT,
        (generic-pointer) (&fcolor));
    /*
    * Close the display.
    */
    i_close (display, NULL);
}
i_rectangle (img, r1, c1, r2, c2, type, data)

IMAGE *img;
float r1, c1, r2, c2;
int type;
generic_pointer data;

Figure i.3: Rectangle Drawing Routines.

i_circle (img, rowcentre, colcentre, radius, type, data)

IMAGE *img;
float rowcentre, colcentre, radius;
int type;
generic_pointer data;

Figure i.4: Circle drawing routine.

i_set_width (img, width)

IMAGE *img;
int width;

Figure i.5: Routine to Set the Line Width.
Another basic graphic routine is the ability to draw text on a generalized image. The text drawing routines allow text to be drawn on single-band or multi-band images. Their function declarations are shown in figure 7.6.1

The first parameter of these routines is the image where the test is to be drawn. The second two parameters specify the location of the test. They contain the pixel location in image coordinates of the upper left hand corner of the test. The fourth parameter is a string containing the test to be written. The last parameter specifies an intensity value for single-band images and an array of values for multi-band images.

The desired size of the text is set by the \texttt{i.set.text.size} routine and the current test size can

\footnotetext[1]{Soon to be changed into generic routines similar to \texttt{i-line}}
be read by the \texttt{i-get-text-size} routine. Both these routines are shown in Figure 7.7.

The first parameter of these routines is the image in which the test size is to be altered or read. The next two parameters represent the test size in pixels. The \texttt{height} and \texttt{width} parameters specify the desired size of the test in pixels. The integers pointed to by the \texttt{pheight} and \texttt{pwidth} parameters will contain the current test size. When the test size is set, the image will round off the number of pixels specified to fit some integer multiple of the allowed test size. The default value set upon opening an image is the smallest allowable size of the test characters. As is the case for the line width parameter, duplicate images created by \texttt{i-dup} initially have the same test size as the original image. Once \texttt{i-dup} has been called, the test size may be changed in either image without affecting the other. It is thus possible to duplicate an image and then change the test size so that intermingled calls can draw test of two different sizes without numerous calls to \texttt{i-set-text-size}. 

\begin{verbatim}
int height, width;
int *pheight, *pwidth;
\end{verbatim}

Figure 7.7: Routines for Adjusting the Test Size.
Chapter 8

Image Sequences

Vision tasks such as robot guidance and motion analysis often involve processing sequences of images. During development, such programs are usually run with canned image sequences. At a later time, they may also be run with live images. Although the two operations are very different, they are also essentially the same. In both cases, a sequence of images are being processed.

The generalized image library provides a useful abstraction in which an entire sequence of images can be manipulated as a single object. Routines are provided for opening and closing image sequences and for obtaining the individual images of the sequence. These routines provide a consistent interface to canned and live image sequences while also allowing the researcher to exercise interactive control over his program.

8.1 The Image Sequence Structure

An image sequence is represented by an IMAGESEQ data structure. The sequence routines operate on pointers to IMAGESEQ objects. The details of the IMAGESEQ structure are unimportant for this discussion, but it is important to distinguish the image sequence structure from the generalized image structure IMAGE.

8.2 Opening Image Sequences

Because image sequences are different from images, they have their own open and create routines. The routine declarations are in figure 8.1. Notice that the parameters are identical to the parameters of the corresponding image open and create routines. The only difference is that these routines return image sequences instead of images.
8.3 Accessing the Images of a Sequence

The routine iseqnext, declared in figure 8.2, returns a selected image from an image sequence. It normally returns an IMAGE pointer, but returns NULL when there are no more images to be processed. Normally, the image returned is the next image in the sequence, but the user can interactively alter the sequence of the images if your program permits interaction. iseqnext can be used in a loop as shown in figure 8.3. Only one image of a sequence can be in use at a given time.

The first argument to iseqnext is the image sequence. The second argument is an interaction flag. If interact is one, then iseqnext enters an interaction cycle every time that it is called. This interaction allows the user to skip backwards or forwards in a canned image sequence and to set the camera parameters for live images. See A User’s Guide to the Generalized Image Library for a detailed description of image sequence interactive control.

If interact is zero, then no interaction takes place and the images are processed in strict sequence.
# include cgimage.h

main (argc, argv)
int argc;
char **argv;
{
    /*
     * Declare input image sequence pointer.
     */
    IMAGESEQ *in;
    /*
     * Declare individual generalized image pointer.
     */
    IMAGE *img;
    /*
     * Initialize the Generalized Image Library.
     */
    i_init ();
    /*
     * Report correct usage if insufficient arguments.
     */
    if (argc < 2)
    {
        fprintf (stderr, 'Usage: process image-sequence.n
');
        exit (1);
    }
    /*
     * Open the input image sequence.
     */
    in = i_seqopen (argv[1], IM-READ, NULL);
    /*
     * Process images in the sequence. While i-seqnext returns
     * a non-null image pointer, processing continues.
     */
    while ((img = i_seqnext (in, 1, 'Input sequence', NULL)) != NULL)
    {
        /*
         * Process the image.
         */
        my-process (img);
        /*
         * Close the sequence image and loop.
         */
        i-close (img, NULL);
    }
}
i_seqclose (seq, err)

IMAGESEQ *seq;
int *err;

Figure 8.3: Routine to Close an Image Sequence.

If interact is minus one (-1) then interaction takes place only when the current image sequence is exhausted. This allows the user to interactively start processing on another image sequence.

The heading parameter to iseqnext is used when prompting the user interactively. It is a string which is printed at the start of the prompt to briefly explain the use of the image sequence in the program. This is especially important in programs which process more than one image sequence.

As is shown in the example, your program must close the image returned by i-seqnext before obtaining another image from the sequence. 1

8.4 Closing Image Sequences

i-seqclose closes an image sequence. It is necessary not only to close the individual images as described above but also to close the image sequence. i_seqclose is declared as shown in figure 1.3.

1This is scheduled to be fixed soon.
Chapter 9

Error Handling

The routines of the generalized image library employ a consistent convention for handling errors. This section describes that convention and how it can be of use to you, the programmer.

With the exception of the pixel access routines, every routine in the library has an error handling parameter err. Err is always the last parameter and it is always an integer pointer. For example, err is the third parameter to i-open. as is shown in the following declaration.

```c
IMAGE *i_open (name, mode, err)
```

double name;
int mode;
int *err;

When a library routine such as i-open is called, the error handling parameter can be used to control the action which is taken when an error is detected. There are two available methods of error handling.

Default error handling is indicated by passing NULL for the error handling parameter. When an error is detected, a message will be printed and processing will be aborted. Section 9.1 describes default error handling in more detail. The pixel access routines, which have no error parameter, always handle errors by default error handling. We strongly recommend that you use the default error handling whenever possible.

Error Trapping is the alternative to default error handling. It is indicated by passing the address of an integer variable as the error handling parameter. When an error is detected, no message will be printed but the integer variable will be set to a non-zero status value which indicates the error that occurred. The library routine will then return to your program. Your program must initialize the integer variable to zero before passing it to the GIL routine. Section 9.2 describes error trapping in greater detail.
9.1 Default Error Handling

If the programmer calls a library routine with \texttt{NULL} for the error handling parameter, then any errors which occur during the processing of the routine call will be handled by printing a message and aborting. The programmer is thus relieved of the task of checking for an error indicator and attempting to compose a meaningful message. The library routine will only return if it was able to perform its normal function.

In the case of \texttt{i-open}, for example, passing \texttt{NULL} for \texttt{err} causes an error message to be printed and the program to be aborted if, for any reason, the named image cannot be opened. This may be because the file does not exist or because there is a syntax error in the name or for a number of other reasons. The error message will inform the user of the precise nature of the problem.

Passing \texttt{NULL} for \texttt{err} is the normal method of writing applications program. For example, figure 9.1 is the complete source for a simple yet powerful image copying program.

Notice that throughout the program in figure 9.1, \texttt{NULL} is used consistently for the error handling parameter. This means that if any routine fails, the error will be reported and the program aborted.

9.2 Error Trapping

If the programmer calls a library routine and passes to it the address of an integer variable as the error handling parameter then errors will be trapped. This means that, when an error is detected, the integer variable will be set to a non-zero status value and the library routines will return control to the user's program. The program may then check the error status variable and take appropriate action if the value is not zero. The library routines do not set the status variable to zero when there is no error; the programmer must initialize the variable to zero before calling the library routines and must reset it to zero after an error has been trapped. This is important; the library routines do not operate correctly if they are passed a non-zero error handling parameter.

As an example of the user of error trapping, consider the task of writing an interactive routine which prompts the user for a file name and then attempts to open it. If the open is successful, the open image is returned. If the open fails, the user is informed of the failure and asked for another image name. Figure 9.2 is a simple routine to perform this task.

In case the reader is not yet familiar with the \texttt{getstr} routine, it is an interactive procedure which prompts the user with the character string \texttt{prompt} and reads the user's response into \texttt{imagename}. \texttt{Default} is a default image name which will be used if the user does not type anything. For our purposes, \texttt{getstr} is just a convenient way of setting \texttt{imagename} to the name of the image which the user wishes to try to open.

\texttt{Error} is the error status variable. Its address is passed to \texttt{i-open} to enable error trapping. The status is initialized to zero before calling \texttt{i-open} and checked after it returns. If the status remains zero, then no error occurred and \texttt{getimage} returns the open image to its caller. If an error occurs.
# include <gimage.h>

main (argc, argv)
int argc;
char **argv;
{
    /*
    * Declare Generalized Image pointers.
    */
    IMAGE *in, *out;
    /*
    * Initialize the Generalized Image Library.
    */
    i-init ();
    /*
    * Inform user of how to use this program if there
    * are insufficient arguments.
    */
    if (argc < 3) {
        fprintf (stderr, 'Usage: copy in.img out.imgnn');
        exit (1);
    }
    /*
    * Open the input image.
    */
    in = i-open (argv[1], 0, NULL);
    /*
    * Create the output image. By default, the output
    * pixel characteristics and image size are the same
    * as the input image.
    */
    out = i-creat (argv[2], i-pixtype (in), i-pixbits (in),
                   si-imglimits (in, NULL), NULL);
    /*
    * Copy the input image to the output image.
    * The copying is verbose: progress marks will be
    * displayed.
    */
    i-copy (out, in, 1, NULL);
    /*
    * Copy the property list from the input image to the
    * output image.
    */
    i-cpypropolist (out, in, NULL);
# include <string.h>
# include <gimage.h>
IMAGE *getimage (prompt, default)
char *prompt, *default;
{
    IMAGE *image;
    char imgname[1000];
    int error;
    while (1)
    {
        /*
        * Ask user for image name.
        */
        getstr (prompt, default, imgname);
        /*
        * Clear error flag and try to open the image.
        */
        error = 0;
        image = i-open (imgname, 0, &error);
        /*
        * Check for error status returned.
        */
        if (error == 0)
            return (image); /* Successful */
        /*
        * Error detected. Inform the user.
        */
        fprintf (stderr, 'Could not open "%s", imgname);}

Figure 9.2: Interactive routine to open an image.
getimage informs the user and again asks him to enter an image name.

One problem with this implementation of getimage is that the error message is not very informative. Getimage does not know very much about the problem which caused i-open to fail. Although the error status value indicates the type of error which caused the failure, there is not enough information in error to compose a truly informative error message. Even worse, if getimage did compose error messages on the basis of the error status then there would be more program code involved in error handling than in the rest of the routine.

The solution to this problem is reasonably simple. The library routines always compose an appropriate error message and store it away for the application routine to print at its leisure. A pair of routines are provided for dealing with the error message. They are inerror and i perror. inerror may be used to prepend the routine name, or any useful string, before the error message. i perror prints the error message on stderr. We can use these two routines to change getimage to give more meaningful error messages. In the modified version, inerror is used to identify the error with the routine getimage and i perror is then called to print the error message. See figure 9.3.
# include <string.h>
# include <gimage.h>

IMAGE *getimage (prompt, default)
char *prompt, *default;

    IMAGE *image;
    char imgname[1000];
    int error;
    while (1)
    {
        /* Ask user for image name. */
        * /
        getstr (prompt, default, imgname);
        /*
        * Clear error flag and try to open the image.
        */
        error = 0;
        image = i_open (imgname, 0, terror);
        /*
        * Check for error status returned.
        */
        if (error == 0)
            return (image); /* Successful */
        /*
        * Error detected. Identify the error with
        * this routine and print the message.
        */
        i_nerror ('getimage: ');
        i_perror (0);
    }

Figure 9.3: Improved interactive image open routine.
Chapter 10

Examples

This section contains a number of examples which illustrate some of the useful features of the Generalized Image Library.

The routine in figure 10.1 creates a ucmat matrix and copies the pixels from an image into it using i-copy. There are other methods that can also be used to achieve the same effect, but they require a more detailed knowledge of the implementation of ucmat matrices.

The example in figure 10.2 is taken from a program which is used to halftone color images for the Shinko color printer. The program creates three matrices to receive the red, green and blue halftone images. The matrices are converted to generalized images and the halftoning virtual image operation is applied to them. The three halftoned virtual images are then combined into a single color image. Copying the input color image into this structure of generalized images results in the generation of halftoned red, green and blue image data in the underlying matrices. The matrices are then complemented and sent to the printer (program code not shown).

Figures 10.3 and 10.4 are two programs which perform very similar functions. Both programs are designed for digitizing sequences of color images. In both cases, the digitization is performed interactively and the output images are generated automatically. However, figure 10.3 uses the routine i-digitize and is therefore only useful for digitizing sequences of images. In contrast, figure 10.4 uses an image sequence for input as well as output. This second program is capable of copying image sequences in addition to being useful for digitizing them. It may be used, for example, to interactively display an image sequence. It is equivalent to the functionality provided by the iingcp program with the -S option.
# include <gimage.h>
# include <matrix.h>

ucmat get-ucmat (inimg)
IMAGE *inimg;
{
    IMAGE *matimg;
    ucmat mat;
    /*
     * Create a matrix with the same bounds as the generalized image.
     */
    mat = newucmat (i_restart(inimg), i_rend(inimg), i_cstart(inimg),
                    i_cend(inimg), NULL);
    /*
     * Obtain a generalized image for the matrix.
     */
    matimg = i-ucmat (mat, NULL);
    /*
     * Copy the input data into the matrix image.
     */
    i-copy (matimg, inimg, 1, NULL);
    /*
     * Close the matrix image. The underlying matrix remains.
     */
    i-close (matimg, NULL);
    /*
     * Return the matrix as the function value.
     */
    return (mat);
}

Figure 10.1: Copying an Image into a Matrix.
# include <gimage.h>
# include <matrix.h>

main (argc, argv)
int argc;
char **argv;
{
    /*
    * Declare Generalized Image pointers.
    */
    /*
    * Declare matrix structures.
    */
    ucmat redmat, grnmat, blumat;
    /*
    * Initialize the generalized image library.
    */
    i_init (); 
    /*
    * Open input color image.
    */
inrgb = i-mopen (argv[1], IM-COLOR, IM-READ, NULL);
    /*
    * Create three matrices: one each for red, green, 
    * and blue.
    */
    redmat = newucmat (i_rstart(inrgb), i_rend(inrgb),
                        i_cstart(inrgb), i_cend(inrgb), NULL);
grnmat = newucmat (i_rstart(inrgb), i_rend(inrgb),
                        i_cstart(inrgb), i_cend(inrgb), NULL);
blumat = newucmat (i_rstart(inrgb), i_rend(inrgb),
                        i_cstart(inrgb), i_cend(inrgb), NULL);
    /*
    * Convert matrices into generalized images.
    */
    rmat = i_ucmat (redmat, NULL);
gmat = i_ucmat (grnmat, NULL);
bmat = i_ucmat (blumat, NULL);
    /*
    * Apply halftoning to matrix images. The halftoned 
    * image pointers are stored in a list.
    */
    list[0] = i_halftone (rmat, 0, 0, 0, NULL);
    list[1] = i_halftone (gmat, 0, 0, 0, NULL);
    list[2] = i_halftone (bmat, 0, 0, 0, NULL);
# include <gimage.h>

main (argc, argv)
int argc;
char **argv;
{
    IMAGESEQ *outseq;
    IMAGE *in, *out;
    /*
    * Initialize the generalized image library.
    */
    i_init ();
    /*
    * Check usage.
    */
    if (argc < 3)
    {
        fprintf (stderr, "Usage: digitize input-device output-sequence
"
        exit (1);
    }
    /*
    * Open input device as a color image.
    */
    in = i-mopen (argv[1], IM-COLOR, IM-MODIFY, NULL);
    /*
    * Open output image sequence.
    */
    outseq = i_seqmcreat (argv[2], IM-COLOR, i_pixtype (in),
             i_pixbits (in), i-imglimits (in, NULL), NULL);
    /*
    * Loop and digitize interactively.
    */
    while (i_digitize (in, 0, 1, NULL))
    {
        /*
        * Obtain output sequence image.
        */
        out = i_seqnext (outseq, 0, 'Output sequence', NULL);
        /*
        * Copy digitized image to output sequence.
        */
        i-copy (out, in, 1, NULL);
        /*
        * Close output sequence image.
        */
# include <gimage.h>

main (argc, argv)
int argc;
char **argv;
{
    IMAGESEQ *inseq, *outseq;
    IMAGE *in, *out;
    /*
    * Initialize the generalized image library.
    */
    i_init(0);
    /*
    * Check usage.
    */
    if (argc < 3)
    {
        fprintf(stderr,
            'Usage: seqcp input-sequence output-sequencenn');
        exit (1);
    }
    /*
    * Open input device or input image sequence.
    */
    inseq = i_seqmopen(argv[1], IM-COLOR, IM-MODIFY, NULL);
    /*
    * Open output image sequence.
    */
    outseq = i_seqmcreat(argv[2], IM-COLOR, i_pixtype(in),
        i_pixbits(in), si_imglimits(in, NULL), NULL);
    /*
    * Process all input images. If digitizing, interactive
    * control will allow user to select moment to digitize and
    * to quit.
    */
    while ((in = i_seqnext(inseq, 1, 'Input sequence', NULL)) !=
        NULL)
    {
        /*
        * Obtain output sequence image
        */
        out = i_seqnext(outseq, 0, 'Output sequence', NULL);
        /*
        * Copy digitized image to output sequence.
        */
        i_copy(out, in, 1, NULL);
Chapter 11

Bugs

The following are some possible problems which are present in the library.

1. **Efficiency** is **always a potential problem**. The time/space/coding effort trade-off is not very easy to deal with. If you suspect an efficiency problem, the IMDEBUG option may help you in tracking it down.

2. There are a lot of rough edges in the error handling. In particular, when *i-open* returns an error status, there may be one or more images left open as a result of the failed attempt to construct the desired image. There is **currently no** mechanism for closing these images, so error recovery in interactive programs can be incomplete. The same problem applies to other routines which create images or return new images for old. A similar problem also applied to the use of allocated memory which may not be freed when an error is trapped. The library needs a facility to enable cleanup of allocated memory when an error is trapped. A future release may provide this.
Chapter 12

Lisp Interface

The Lisp interface to the Generalized Image Library allows Lisp users access to the Generalized Image Library from within Sun Lucid Common Lisp. The majority of Generalized Image Library functions are supported by Lisp counter-parts. For the most part the Lisp routines have similar names and parameters to their C cousins.

The most notable exception to this rule is that the Lisp interface provides one type-independent routine for fetching or storing image data instead of the five different routines provided to the C programmer. Thus, where the C programmer would invoke `ifputpixel` to store a floating pixel value and `i-putpixel` to store an integer value, the Lisp programmer may simply invoke `i-putpixel` and the Generalized Image Library interface will invoke the appropriate C routine dependent on the data type of the Lisp value.

This document describes release 1.12 of the Generalized Image Library, and later releases. The Lisp interface in earlier releases is no longer supported. NOTE: the 'latest' release (as of 15-Aug-88) has some cursor manipulation routines, which are untested by this author.

12.1 Loading the Lisp Interface

To load the Generalized Image Library Lisp interface, commands similar to the following should be used. The first command loads the Lisp portion of the interface. The second command loads the C code and makes the interface available to the user. This separation allow the library to be preloaded into a saved Lisp image.

```
(load "/usr/vision/experimental/lib/libgimage.lisp")
(gil:use-gil-lisp-interface)
```
12.2 Image Access Operations

The Lisp interface to the Generalized Image Library provides type-generic image access operations. Unlike the C interface which has five putpixel operations corresponding to the five supported C program data types, the Lisp interface has one call which is capable of handling a variety of Lisp number formats. The interface examines the number and automatically determines the most appropriate C interface to use.

Some of the image access operations use data stored in Lisp arrays. In all cases where a Lisp array is passed into the GIL, it must be an array of one of the following Lisp types: single-float, (signed-byte 32), (signed-byte 16) or (unsigned-byte 8). These correspond to the type keywords :float, :int, :short-int, :unsigned-char which may be passed to imake-array (see section 12.3).

12.2.1 Single-Band Image Put Operations

Figure 12.1 summarizes the operations for storing data into single-band images. This section describes their calling conventions from Lisp. A more complete description of the action performed by each routine is in section 2.

i-putpixel accepts a non-complex Lisp number and stores it at the specified co-ordinates in the image. The pixel value is stored as an integer if it satisfies the Lisp predicate integerp. Otherwise it is stored as a floating pixel. The image co-ordinates are converted to integers by rounding, and may be any non-complex Lisp numbers.
i_mputpixel image row column pixel-data
i_mputrow image row cstart cend row-data
i_mputcol image rstart rend column column-data
i_mputbox image rstart rend cstart cend box-data
i_mfillrow image row cstart cend pixel-data
i_mfillcol image rstart rend column pixel-data
i_mfillbox image rstart rend cstart cend pixel-data

Figure 12.2: Routines to store multi-band image data.

i_fillrow, i_fillcol and i_fillbox also accept any non-complex Lisp number as a pixel value. The value is used to fill the specified portion of the image. Again, the image co-ordinates may be any non-complex Lisp numbers and are rounded automatically.

I-putrow and i_putcol accept either a one-dimensional array of data or a list and store it into the image. The dimension of the array must be at least as large as the specified area of the image to be modified. If a list is used, it must have exactly the correct number of values for the image area to be modified. The GIL Lisp interface converts the list into an array of floating values which is then passed to the C interface.

I-putbox accepts a two-dimensional array of data or a list of lists and stores it into the image. If an array is used, its dimensions must be at least as large as the image area to be modified. Element (aref array i j) corresponds to pixel rstart+i, column cstart+j of the image. In a lists-of-lists representation, the innermost lists correspond to rows of the image. This is one such list for each row to be modified. The same data types are supported as for i_putrow.

12.2.2 Multi-Band Image Put Operations

Figure 12.2 summarizes the operations for storing data into multi-band images.

i_mputpixel accepts a one-dimensional array or a list of values and stores it into a multi-band image. The dimension of the array must be at least as large as the number of image bands; if a list is used, it is required to have exactly one element for each image band.

i_fillrow, i_fillcol and i_fillbox fill a selected region of the image with a constant multi-band pixel value. The pixel data may be passed as a one-dimensional array or as a list.

i_mputrow and i_mputcol store either two-dimensional arrays of data or lists of lists into the image. Element (aref array i j) corresponds to pixel j of image band i. When a list-of-lists is used, each inner-most list corresponds to the pixel values for a single band of the image and there is one such list for each image band. For example, figure 12.3 stores the four colors white, red,
(setq img (i_mopen 'color:display' :color))
(i_mputrow img 100 100 103
  '((255 255 0 0)
   (255 0 255 0)
   (255 0 0 255)))

Figure 12.3: Example of multi-band putrow operation.

i_getpixel image row column &optional (type-keyword :float)
i_getrow image row cstart cend &optional (type-or-data :list)
i_getcol image rstart rend column &optional (type-or-data :list)
i_getbox image rstart rend cstart cend &optional (type-or-data :list)

Figure 12.4: Single-band image data retrieval operations.

green and blue into four successive pixels of a color image.

i_mputbox stores either a three-dimensional array of data or a list of lists of lists into a multi-band image. Array element (aref array i j k) is stored into color band i of the pixel at row rstart+j, column cstart+k in the image. When a list of lists of lists is passed into i_mputbox, the nested lists are converted into a three-dimensional array of floating numbers and are then passed into the C interface. The inner-most lists correspond to individual pixels across a row of one band of the image.

12.2.3 Single-Band Image Get Operations

The operations for retrieving data from images can (with the exception of i_getpixel which always returns a scalar value) operate either by storing the data into existing arrays or by creating a new data structure and returning it. Whether a new data structure or an existing structure is used, the structure itself is returned as the function value. Figure 12.4 summarizes the Lisp procedures for retrieving image data.

i_getpixel fetches the pixel value of an image at specified row and column co-ordinates. The image co-ordinates may be any Lisp numbers and are rounded into integers before invoking the C interface. The optional parameter type-keyword may be any one of the following keywords: :float, :int, :short-int or :unsigned-char. It is used to specify the type of pixel fetch operation to be
i_mgetpixel image row column &optional (type-or-data :list)
i_mgetrow image row cstart cend &optional (type-or-data :list)
i_mgetcol image rstart rend column &optional (type-or-data :list)
i_mgetbox image rstart rend cstart cend &optional (type-or-data :list)

Figure 12.5: Multi-band image data retrieval operations.

performed. By default, pixels are fetched as floating values.

I_getrow fetches the pixel values of a selected portion of a row of an image. The optional parameter type-or-data may either be an array into which the data is fetched, or a type keyword. If an array is specified, it must be one of the acceptable types of arrays as listed previously. The type keywords :float, :int, :short-int and :unsigned-char are handled by creating an array of the corresponding type and fetching the data into it. The array is then returned as the function value.

The default keyword :list is handled specially. The GIL Lisp interface creates a floating array and fetches the data into it. It then converts the array into a list which is returned. This is most useful in an interactive contest as a way of conveniently displaying image data.

I_getcol is similar to I_getrow but fetches a portion of a column of the image. I_getbox fetches a two-dimensional portion of the image, either into a two-dimensional array or as a list of lists. The first dimension of the array corresponds to rows of the image. The second dimension of the array corresponds to the columns of the image. In a list-of-lists representation, the inner-most lists correspond to individual rows of the image.

12.2.4 Multi-Band Image Get Operations

The operations for retrieving data from multi-band images can either store the data in existing arrays or create a new data structure and return the data in it. Whether a new structure is created or an existing structure is used, the structure itself is returned as the function value. Figure 12.5 summarizes the GIL Lisp procedures for retrieving multi-band image data.

I_mgetpixel fetches the multi-band pixel value of an image at specified row and column co-ordinates. The image co-ordinates may be any Lisp numbers and are rounded to integers automatically. The optional parameter type-or-data may either be an array of one of the allowable types, or it may be a type keyword as listed previously. By default, I_mgetpixel fetches the pixel data as floating values and then converts them into a list which is returned.

I_mgetrow and I_mgetcol fetch a single row or column of a multi-band image. The parameter type-or-data may either be a two-dimensional array or a type keyword. If a type keyword is used.
the routines create an array of the appropriate type and return the data in it. When an array is used, the element \((\text{aref} \text{ array } i \ j)\) corresponds to pixel \(j\) in band \(i\) of the image. If \text{type-or-data} is unspecified, a list-of-lists structure is created and returned.

\text{Imgetbox} retrieves a rectangular portion of a multi-band image. \text{Type-or-data} may be either a three-dimensional array or a type keyword. As before, keywords are handled by creating an array and returning the data in it. The default is to create a list of lists of lists. When an array is returned, element \((\text{aref} \text{ array } i \ j \ k)\) corresponds to band \(i\) of the pixel at row \(r\text{start}+j\), column \(c\text{start}+k\) in the image.

12.3 Creating Arrays for Pixel Data

The GIL Lisp interface allows data to be fetched into several specific types of Lisp arrays. These types are restricted by the capabilities of the Lucid Lisp interface to C. Arrays of the required type may be created by invoking \text{make-array} or by invoking the GIL equivalent routine \text{i.make-array}. The syntax of \text{i.make-array} as is follows.

\text{i.make-array type dimensions}

\text{Type} is one of the following data type keywords: \text{:float}, \text{:int}, \text{:short-int} or \text{:unsigned-char}. These keywords correspond to the C data types \text{float}, \text{int}, \text{short int} and \text{unsigned char}. They also correspond to the Lisp data types \text{single-float}, \text{(signed-byte 32)}, \text{(signed-byte 16)} and \text{(unsigned-byte 8)}.

\text{Dimensions} is a list of integer which specify the desired array dimensions.

\text{i.make.array} is provided purely as a convenience to the user. The following two Lisp statements are equivalent in every respect.

\[(\text{i.make.array} :\text{int} \ '(4))\]

\[(\text{make-array} \ '(4) :\text{element-type} \ '(\text{signed-byte 32}))\]

12.4 Copying Images

The image copying routines described in figure 12.6 can be used to efficiently copy data from one image to another. A detailed functional description of these routines can be found in section 2.13.

\text{I-copy} copies \text{fromimg} to \text{toimg}. \text{Verbose} is the verbosity parameter; if it is not \text{nil} (the default) then progress marks will be displayed during copying. \text{I-copy} allows each of the parameters \text{toimg} and \text{fromimg} to be either a Lisp string or an image pointer. If the parameter \text{fromimg} is a Lisp
Figure 12.6: Image copying routines.

```
i_copy toimg fromimg %optional (verbose nil) (err nil)
i_copyw toimg fromimg bounds &optional (verbose nil) (err nil)
i_scopuw toimg fromimg bounds row-offset column-offset
&optional (verbose nil) (err nil)
```

string, then an image of that name will be opened and the data copied into the output image. After copying, the image will be closed. If fromimg is an image pointer, then the data will simply be copied from the image and the image will not be closed. Similarly, if toimg is a Lisp string, an image of that name will be created and the data will be copied into it. If toimg is an image pointer, then the data will simply be copied into it. This feature of the Lisp interface is especially useful interactively. For example, the following Lisp command displays a currently active image resulting in the top-left corner of the display device.

```
(i_copy 'q:1:display' resultimg)
```

I-copyw and iscopyw do not currently allow their parameters fromimg and toimg to be strings: they must be currently active image pointers.

### 12.5 Opening and Closing Images

The GIL Lisp interface for opening and closing images is very similar to the C interface. The differences are that some parameters are optional and the Lisp interface uses Lisp keywords for pixel type and image bands specifications. Figure 12.7 summarizes the Lisp interface for opening and closing images.

I-open opens a single-band image. Image is a Lisp string which specifies the image name. Mode is an optional access-mode keyword: either :read, :write or :modify. The default is :read. Err is the Lisp error handling parameter which defaults to nil.

I_mopen is similar to i-open but opens multi-band images. Bands is a bands specification for the image. It may either be a Lisp string, or nil, or one of the keywords :color, :stereo, :color-stereo, :single or :bw. The last two are interpreted as requests to open a single-band image, which is equivalent to passing an empty string. Nil indicates that the bands are unspecified and any multi-band image is allowed. It is equivalent to NULL in the C interface.
i-open image &optional (mode :read) (err nil)
imopen image bands &optional (mode :read) (err nil)
i-creat image pixtype pixbits bounds &optional (err nil)
imcreat image bands pixtype pixbits bounds &optional (err nil)
i-close image &optional (err nil)
iscrap image &optional (err nil)
isetscrap image &optional (err nil)

Figure 12.7: Routines to open and close images.

(setq img
  (i-creat 'new.gif' :unsigned 8 '(0 479 0 511)))

Figure 12.8: Example of i-creat.

I-creat creates a single-band image. Image is the name of the image to be created. Pixtype is the pixel type, specified as one of the following keywords: :float, :signed or :unsigned. Pixbits specifies the bits per pixel and is a Lisp number. Bounds specifies the image bounds. It may be a list of four numbers or an array of type (signed-byte 32). The elements of the list or array correspond to the row start, row end, column start and column end co-ordinates in that order. For example, figure 12.8 shows the creation of an image with 480 rows by 512 columns of unsigned 8-bit pixels.

I-mcreat creates a multi-band image. It is similar to i-creat but also has a bands parameter which is interpreted exactly as in imopen.

I-close, iscrap and i-setscrap have exactly the same semantics as their C equivalents. In all cases, the Lisp error handling parameter is optional and defaults to nil.

12.6 Changing Images

Figure 12.9 summarizes the operations which are provided in the GIL Lisp interface for virtual modification of images. Detailed functional descriptions of these operations are in section 5.

I-dup duplicates an image descriptor. The error handling parameter is optional and defaults to nil.
i_dup image &optional (err nil)
i_crop image bounds &optional (err nil)
i_shift image &optional (restart 0) (cstart 0) (err nil)
i_ltrans image &optional (multiply 1.0) (add 0.0) (err nil)

Figure 12.9: Virtual Image Modifications.

I-crop crops an image. The parameter bounds specifies the desired cropped image bounds. It may be either a list of four non-complex numbers, or an array of four integers of type (signed-byte 32). When a list representation is used, the values are automatically rounded to integers before passing them to the C interface. The elements of the list or array correspond to the starting row, the ending row, the starting column and the ending column.

I-shift shifts the origin of an image. The optional parameters restart and cstart specify the desired origin: if omitted an origin of (0,0) is assumed.

I_ltrans computes a linear transformation of an image. When pixels are fetched from the transformed image, they are multiplied by multiply and then odd is added to them. Multiply and add may be any non-complex numbers: they are converted to floats and passed to the C interface.

12.7 Image Descriptive Information

The GIL Lisp interface allows images to be queried for their descriptive information using the routines described in figure 12.10.

i_rstart, i_rend, i_cstart and i_cend return the image bounds as Lisp integers. i_rows and i_cols return the size of the image as Lisp integers. i_bounds returns the image bounds as a list of four integers corresponding to the row start, row end, column start and column end co-ordinates.

I-pixbits returns the pixel bits of the image as a Lisp integer. I-pixtype returns one of the keywords :float, :signed or :unsigned indicating the pixel type of the image.

I_bands returns the number of bands in the image as a Lisp integer. I_bands returns the multi-band specification of the image as a Lisp string.

I_max_bands (which has no parameters) returns the maximum number of bands that is permitted in a generalized image. Currently, this number is 128.
The simple graphics operations described in section 7 are also provided in the GIL Lisp interface. However, only one interface routine is provided for each graphic operation; the Lisp interface invokes the appropriate C routine depending on the type of the pixel data. Further, the same routine can be used to interface to both single-band and multi-band images, and may be passed a single number, a list of numbers or a one-dimensional array of any of the allowed types. Figure 12.13 summarizes the GIL Lisp graphics operations.

**Iline** draws a line in the image. The line is drawn from pixel co-ordinate \( r1 \ c1 \) to pixel co-ordinate \( r2 \ c2 \). The co-ordinates may be any non-complex Lisp numbers. The line is clipped automatically at the boundary of the image.
(setq disp (i_mopen 'display' :color :modify))
(dotimes (i 100)
  (i_circle disp 240 256 (* i 3) '(255 255 0)))

Figure 12.12: Drawing circles on the display.

i_text img row col text val
i_vtext img row col text val
i_mtext img row col text val
i_mvtext img row col text val

Figure 12.13: Text Routines

**I_rectangle** draws a rectangle on the image. The rectangle is defined as having \( r_1 c_1 \) as one of its corners and \( r_2 c_2 \) as the diagonally opposite corner. If the rectangle lies partially off the image, that portion of the rectangle will not be displayed.

**I_circle** draws a circle on the image. The circle is centred at pixel co-ordinates \( r_\text{centre} c_\text{centre} \) and has radius \( \text{radius} \). The co-ordinates and radius may be any non-complex Lisp numbers. They are passed into the C interface as floating quantities. The circle is clipped to the image bounds. Figure 12.12 contains an example of Lisp code which draws 100 concentric yellow circles on the display image.

**I_set-width** adjusts the line thickness used for drawing. The line width is expressed in pixels and defaults to one.

### 12.9 Text Routines

The text routines described in section 7 are provided in GIL Lisp interface. But as before there is only one routine for each operation rather than four routines for each operation to distinguish the types. The routines may be passed a single number, a list of numbers, or a one dimensional array of any of the allowed types as pixel data. There are single band and multi-band routines as well as routines for producing vertical text. Figure 12.13 summarizes the text routines.

**I-text** and **i-vtext** put \( t_{text} \) on a single band image starting at position \( r_{\text{row}} c_{\text{col}} \) where \( t_{text} \) is a string. **i_text** produces the text horizontally and **i_vtext** vertically. (hence the 'v' in the name)
i_get_text_size ini &optional (err nil)
i_set_text_size iuw &optional (err nil)

Figure 12.14: Getting and Setting Test Size

Imtext and imvtext are the same as i_text and i_vtext except that they work for multi-band images.

There are also two functions i_get_text_size and i_set_text_width that set the size of the test. i_get_text_size returns a cons cell (h,w) where h is the height of the test in pixels and w is its width. i_set_text_size takes a cons cell and sets the test size to be the sizes in the cons cell. Figure 12.14 describes these functions.

12.10 Error Handling

In the GIL C language interface, the error handling parameter is either an integer pointer or NULL. In the GIL Lisp interface, the error handling parameter is either an array of one element of type (signed-byte 32), or nil which corresponds to NULL in the C interface. In addition, the Lisp interface adopts the convention that the error handling parameter is always optional: when it is omitted, nil is assumed. It is thus possible to interactively invoke Generalized Image Library calls without any concern for the error handling parameter.

In the event that an error is detected in the C portion of the library, the error is handled as described in section 9: if the error handling parameter is indeed a one-element array of (signed-byte 32), then the error code is stored into the array and control is returned to Lisp. It is the responsibility of the Lisp user to initialize the array to zero, and to reinitialize it after an error has been trapped. If the error handling parameter is nil then the Generalized Image Library displays an error message and generates a segmentation violation interrupt which returns control to the Lisp debugger. At this point the user should simply type :A to return to the Lisp top-level evaluator. This method of error recovery is also employed when an error is detected inside the image access routines which have no error handling parameter.

Warning: The recovery of the GIL after interrupt-handled errors is not always complete. The recovery is even worse if the user interrupts GIL processing by typing a keyboard interrupt.

In order to trap errors using the error handling parameter it is necessary to create a one-element array of (signed-byte 32). This can be achieved with make-array or with the following call to imake-array.

(setq gil-error-trap...
(defun getimage (prompt default)
  (let (given-name image
        (error-trap (i-make-array :int '1)))
    (loop
      ;; Prompt the user and get his response.
      (format t "A [A] : prompt default)
      (setq given-name (read-line))
      ;; If empty response, assume the default.
      (if (equalp given-name ')
          (setq given-name default)
          ;; Clear the error trap status and attempt to
          ;; open the image.
          (setf (aref error-trap 0) 0)
          (setq image (i-open given-name :read error-trap))
          (if (= 0 (aref error-trap 0))
              (return image))
          ;; Report the error and try again.
          (i-nmerror 'getimage: ')
          (i-perror)
          )))

Figure 12.15: An interactive Lisp procedure for opening an image.

As an example, figure 12.15 contains a Lisp procedure getimage for interactively opening an image. This routines traps any error which occurs during opening of the image. reports the error to the user and asks the user for another image name to open. It is the Lisp equivalent of the C routine getimage in figure 9.3.
12.11 A Simple Example

A simple example of a program using the Lisp interface can be seen in figure 12.16. This program when called will create a checker board pattern in the file checkers.gif. **Note:** We do not call i.init because the GIL Lisp interface performs all necessary initialization.
(defun checkers ()
  ; Create the image file.
  (let ((img (i_creat 'checkers.gif'
                   :unsigned 32 '(-100 100 -100 100) nil)))

    ; Store the pattern in it.
    (make-pattern img)

    ; Close the image.
    (i_close img nil)
))

(defun make-pattern (img)
  (let ((r0 (i_rstart img))
        (c0 (i_cstart img)))
    ; Process all rows and columns.
    (dotimes (i (i_rows img))
      (dotimes (j (i_cols img))
        ; Put the value at each pixel.
        (putval img (+ i r0) (+ j c0))))
))

(defun putval (img x y)
  ; Defines a checkerboard pattern.
  (if (evenp (+ (round (/ x 10)) (round (/ y 10))))
      (i_putpixel img x y 0)
      (i_putpixel img x y 255))
))

Figure 12.16: Creating a pattern with the GIL Lisp interface.