Appendix C

Programming Techniques

This appendix covers some aspects of C programming that are important for image processing and outlines the design of the image processing routines that were used to produce some of the examples in this book.

C.1 Image Descriptors

Images can be represented in C using image descriptors, which are structures that contain all of the information needed by subroutines to process an image. Image descriptors are an efficient representation and allow an image to be passed to a subroutine through a single subroutine parameter.

An image descriptor contains the following fields:

`address`	Pointer to the beginning of the image array
`height`	Number of rows in the image
`width`	Number of columns in the image
`span`	Allocated length of each row in the image array

The image descriptor fields should be accessed with macros defined in a header file so that programs are independent of changes in the image representation. A portion of the header file is listed in Program C.1.

Program C.1 The definition of pixels and image descriptors

```
typedef int pixel;

typedef struct _id
{
    pixel *id_address;
    int id_height,
        id_width,
        id_span;
} image_descriptor;

#define address(id_ptr)     ((id_ptr) -> id_address)
#define span(id_ptr)        ((id_ptr) -> id_span)
#define height(id_ptr)      ((id_ptr) -> id_height)
#define width(id_ptr)       ((id_ptr) -> id_width)
```

For convenience, pixels are defined to be 32-bit signed numbers. By redefining the pixel data type and recompiling the programs, the software can be changed to use more common pixel data types such as unsigned bytes. However, image processing algorithms may yield intermediate results that do not conform to the limited range of small integers. More common numerical representations, such as single-precision floating point numbers, may be more suitable for many image processing operations.

The C routine **array** takes a pointer to an image descriptor and returns an array of pointers to the first pixel in each row of the image. A pointer to the array of row pointers is stored in the image descriptor. The routine **array** uses the array stored in the descriptor if it exists; otherwise, it allocates and returns the array of row pointers after storing it in the image descriptor. The storage used by the array of row pointers is reclaimed when the storage used by the image descriptor itself is freed by a call to **free_image**. The array of pointers to the image rows should not be freed explicitly.

The array of image row pointers allows C style array access. As an example of the use of image descriptors, consider the routine listed in Program C.2 for summing the pixels in an image.

Program C.2 A routine for summing the pixels in an image using an array of pointers to the image rows

```
int sum_image(image)
image_descriptor *image;
{
  pixel **image_array = array(image);
  int row, column, sum;

  for (sum = 0, row = 0; row < height(image); row ++)
    for (column = 0; column < width(image); column ++)
      sum += image_array[row][column];
  return (sum);
}
```

Using an array of pointers to the array rows is the easiest method for accessing multidimensional arrays in C and is probably the style that is used most often, but there are situations where it is necessary to write C code that manipulates the pointers directly. Program C.3 sums the pixels in an image without using an array of pointers to the image rows. Pointer arithmetic is coded directly in the program.

Program C.3 A routine for summing the pixels in an image by advancing pointers down the rows and across the columns of the image

```
int sum_image(image)
image_descriptor *image;
{
  pixel *rowptr, *colptr;
  int rowcnt, colcnt, sum;

  for (sum = 0, rowcnt = height(image),
       rowptr = address(image);
       rowcnt > 0;
       rowptr += span(image), rowcnt--)
    for (colcnt = width(image), colptr = rowptr;
         colcnt > 0;
         colcnt--)
```

```
      sum += *(colptr++);
  return (sum);
}
```

The third alternative for accessing the pixels in an image is to code the subscript calculations for accessing each pixel. The formula for accessing the pixel at a specified row and column in an image is

```
address(image)[row * span(image) + column]
```

A program for copying one image to another is listed in Program C.4.

Program C.4 Program to copy one image to another that will work correctly even if the images are not the same size or have different row dimensions

```
copy_image (input, output)
image_descriptor *input, *output;
{
  pixel *input_row_ptr, *output_row_ptr, *inptr, *outptr;
  int rowcnt, colcnt;

  for (rowcnt = min(height(input), height(output)),
       input_row_ptr = address(input),
       output_row_ptr = address(output);
       rowcnt > 0;
       rowcnt--,
       input_row_ptr += span(input),
       output_row_ptr += span(output))
    {
      for (colcnt = min(width(input), width(output)),
           inptr = input_row_ptr,
           outptr = output_row_ptr;
           colcnt > 0;
           colcnt--)
        *(outptr++) = *(inptr++);
    }
}
```

Program C.4 is carefully written to work correctly even if the images are of different sizes or the image spans are different. Defensive coding makes the programs more robust.

The reason for including the span (also called the stride or row dimension) in the image descriptor is that the image may be stored in an array that is wider than the image. For example, Program C.5 returns a subimage of an image. The beauty of image descriptors is that the programs that handle images will work with subimages since the operations using image descriptors are identical.

Program C.5 A routine to return a subimage of an image

```
image_descriptor *make_subimage
                (image, subimage_row, subimage_column,
                 subimage_height, subimage_width)
image_descriptor *image;
int subimage_row, subimage_column,
                              subimage_height, subimage_width;
{
  image_descriptor *subimage;

  subimage = (image_descriptor *)
                             malloc(sizeof(image_descriptor));

  init_image_descriptor (subimage,
                         pixel_address(image, subimage_row,
                                                 subimage_column),
                         span(image),
                         subimage_height,
                         subimage_width);
  return (subimage);
}
```

Routines are provided for allocating images and freeing images: `make_image` and `free_image`, respectively.

C.2 Mapping Operators

Some patterns of code occur repeatedly in image processing programs. For example, the code for incrementing pixel pointers and counts to apply a local operator to successive windows of an image will occur without change in every routine that applies local operators to images. These program idioms can be coded in a set of mapping routines that apply an arbitrary operator to an image. The operators are not entirely arbitrary since the number of images must be coded into the argument list. As an example, consider Program C.6, which applies an operator of arbitrary size across successive supports of an input image and places the output into successive pixels of an output image.

Program C.6 Routine to map an operator with arbitrary support across an image and store the results in an output image. The image operator takes a pointer to a window as input and returns a pixel.

```
map_image_support (input, output, function,
                   support_height, support_width)
image_descriptor *input, *output;
int support_height, support_width;
pixel (* function)();
{
  int output_height = height(input) - support_height + 1,
      output_width  = width(input)  - support_width  + 1;
  image_descriptor support_descriptor;
  image_descriptor *support = &support_descriptor;
  int rowcnt, colcnt;
  pixel *input_row_ptr, *output_row_ptr,
        *input_column_ptr, *output_column_ptr;

  init_image_descriptor (support, NULL, span(input),
                         support_height, support_width);

  for (rowcnt = min(output_height, height(output)),
       input_row_ptr = address(input),
       output_row_ptr = address(output);
       rowcnt > 0;
       rowcnt--, input_row_ptr += span(input),
```

```
        output_row_ptr += span(output))
    for (colcnt = min(output_width, width(output)),
         input_column_ptr = input_row_ptr,
         output_column_ptr = output_row_ptr;
         colcnt > 0; colcnt--)
      {
        address(support) = input_column_ptr++;
        *(output_column_ptr++) = (* function)(support);
      }
}
```

As an example of how a mapping function can be used, the following statement uses **map_image_support** listed in Program C.6 and **sum_image** listed in Program C.3 to smooth an image with a 3×3 averaging mask:

```
map_image_support (input, output, sum_image, 3, 3);
```

The statement works because **sum_image** is passed an image descriptor for the 3 by 3 window. Note that the mapping functions may not update the array of row pointers computed by **array**, since this would be inefficient. To use mapping functions, the pointer calculations are coded directly into the routines that implement the local operations.

Mapping functions can be very useful since they allow arbitrarily complicated operators to be applied to images. This is generalized convolution and is used frequently in image processing.

C.3 Image File Formats

There are many image file formats in use, but the formats supported by the *pbmplus* package are portable across different machine architectures. The package handles binary images, gray-level images, and multichannel (color) images and is available on the Internet. The pbmplus package includes C routines for reading and writing image files and utility programs for converting between the pbmplus file formats and other image file formats.

Images stored in pbmplus files are not compressed, but the files can be compressed by programs such as *compress*, which is available on Unix systems, or *gzip* which is available on the Internet.

Further Reading

There are many excellent books on programming in C. The original book on C was written by Kernighan and Ritchie [142]. Harbison and Steele wrote a reference manual on C that is useful for experienced C programmers [104].

The book *Numerical Recipes* [197] describes methods for programming matrix and vector operations, and *Graphics Gems* [89] includes some discussion of programming techniques for image processing.

The Tcl/Tk toolkit is useful for developing programs with interactive interfaces and also provides an extensible command line interpreter [191]. Image processing operations can be implemented as commands that are invoked interactively or through the command line interface. The command language allows machine vision algorithms to be written as scripts, so that repetitive operations can be performed easily. Machine vision applications can be modified by changing the scripts, so it is not necessary to change the image processing programs.