CHAPTER

1

Practical Aspects of a Vision System—Image Display, Input/Output, and Library Calls

When experimenting with vision- and image-analysis systems or implementing one for a practical purpose, a basic software infrastructure is essential. Images consist of pixels, and in a typical image from a digital camera there will be 4–6 million pixels, each representing the color at a point in the image. This large amount of data is stored as a file in a format (such as GIF or JPEG) suitable for manipulation by commercial software packages, such as Photoshop and Paint. Developing new image-analysis software means first being able to read these files into an internal form that allows access to the pixel values. There is nothing exciting about code that does this, and it does not involve any actual image processing, but it is an essential first step. Similarly, image-analysis software will need to display images on the screen and save them in standard formats. It's probably useful to have a facility for image capture available, too. None of these operations modify an image but simply move it about in useful ways.

These bookkeeping tasks can require most of the code involved in an imaging program. The procedure for changing all red pixels to yellow, for example, can contain as few as 10 lines of code; yet, the program needed to read the image, display it, and output of the result may require an additional 2,000 lines of code, or even more.

Of course, this infrastructure code (which can be thought of as an *application programming interface*, or *API*) can be used for all applications; so, once it is developed, the API can be used without change until updates are required. Changes in the operating system, in underlying libraries, or in additional functionalities can require new versions of the API. If properly done, these

new versions will require little or no modification to the vision programs that depend on it. Such an API is the *OpenCV* system.

1.1 OpenCV

OpenCV was originally developed by Intel. At the time of this writing, version 2.0 is current and can be downloaded from `http://sourceforge.net/projects/opencvlibrary/`.

However, Version 2.0 is relatively new, yet it does not install and compile with all of the major systems and compilers. All the examples in this book use Version 1.1 from `http://sourceforge.net/projects/opencvlibrary/files/opencv-win/1.1pre1/OpenCV_1.1pre1a.exe/download`, and compile with the Microsoft Visual C++ 2008 Express Edition, which can be downloaded from `www.microsoft.com/express/Downloads/#2008-Visual-CPP`.

The *Algorithms for Image Processing and Computer Vision* website (`www.wiley.com/go/jrparker`) will maintain current links to new versions of these tools. The website shows how to install both the compiler and OpenCV. The advantage of using this combination of tools is that they are still pretty current, they work, and they are free.

1.2 The Basic OpenCV Code

OpenCV is a library of C functions that implement both infrastructure operations and image-processing and vision functions. Developers can, of course, add their own functions into the mix. Thus, any of the code described here can be invoked from a program that uses the OpenCV paradigm, meaning that the methods of this book are available in addition to those of OpenCV. One simply needs to know how to call the library, and what the basic data structures of open CV are.

OpenCV is a large and complex library. To assist everyone in starting to use it, the following is a basic program that can be modified to do almost anything that anyone would want:

```
// basic.c : A `wrapper´ for basic vision programs.
#include "stdafx.h"
#include "cv.h"
#include "highgui.h"
int main (int argc, char* argv[])
{
    IplImage *image = 0;
```

```
    image = cvLoadImage(˜C:\AIPCV\image1.jpg˝, 1 );
    if( image )
    {
        cvNamedWindow( ˜Input Image˝, 1 );
        cvShowImage( ˜Input Image˝, image );
        printf( ˜Press a key to exit\n˝);
        cvWaitKey(0);
        cvDestroyWindow(˜String˝);
    }
    else
        fprintf( stderr, ˜Error reading image\n˝ );
    return 0;
}
```

This is similar to many example programs on the Internet. It reads in an image (`C:\AIPCV\image1.jpg` is a string giving the path name of the image) and displays it in a window on the screen. When the user presses a key, the program terminates after destroying the display window.

Before anyone can modify this code in a knowledgeable way, the data structures and functions need to be explained.

1.2.1 The IplImage Data Structure

The `IplImage` structure is the in-memory data organization for an image. Images in `IplImage` form can be converted into arrays of pixels, but `IplImage` also contains a lot of structural information about the image data, which can have many forms. For example, an image read from a GIF file could be 256 grey levels with an 8-bit pixel size, or a JPEG file could be read into a 24-bit per pixel color image. Both files can be represented as an `IplImage`.

An `IplImage` is much like other internal image representations in its basic organization. The essential fields are as follows:

`width`	An integer holding the width of the image in pixels
`height`	An integer holding the height of the image in pixels
`imageData`	A pointer to an array of characters, each one an actual pixel or color value

If each pixel is one byte, this is really all we need. However, there are many data types for an image within OpenCV; they can be bytes, ints, floats, or doubles in type, for instance. They can be greys (1 byte) or 3-byte color (RGB), 4 bytes, and so on. Finally, some image formats may have the origin at the upper left (most do, in fact) and some use the lower left (only Microsoft).

Other useful fields to know about include the following:

`nChannels`	An integer specifying the number of colors per pixel (1–4).
`depth`	An integer specifying the number of bits per pixel.
`origin`	The origin of the coordinate system. An integer: 0=upper left, 1=lower left.
`widthStep`	An integer specifying, in bytes, the size of one row of the image.
`imageSize`	An integer specifying, in bytes, the size of the image (= widthStep * height).
`imageDataOrigin`	A pointer to the origin (root, base) of the image.
`roi`	A pointer to a structure that defines a region of interest within this image that is being processed.

When an image is created or read in from a file, an instance of an `IplImage` is created for it, and the appropriate fields are given values. Consider the following definition:

```
IplImage* img = 0;
```

As will be described later in more detail, an image can be read from a file by the following code:

```
img = cvLoadImage(filename);
```

where the variable `filename` is a string holding the name of the image file. If this succeeds, then

```
img->imageData
```

points to the block of memory where the pixels can be found. Figure 1.1 shows a JPEG image named `marchA062.jpg` that can be used as an example.

Reading this image creates a specific type of internal representation common to basic RGB images and will be the most likely variant of the `IplImage` structure to be encountered in real situations. This representation has each pixel represented as three bytes: one for red, one for green, and one for blue. They appear in the order b, g, r, starting at the first row of the image and stepping through columns, and then rows. Thus, the data pointed to by `img->imageData` is stored in the following order:

$b_{0,0}$ $g_{0,0}$ $r_{0,0}$ $b_{0,1}$ $g_{0,1}$ $r_{0,1}$ $b_{0,2}$ $g_{0,2}$ $r_{0,2}$ . . .

This means that the RGB values of the pixels in the first row (row 0) appear in reverse order (b, g, r) for all pixels in that row. Then comes the next row, starting over at column 0, and so on, until the final row.

Figure 1.1: Sample digital image for use in this chapter. It is an image of a tree in Chico, CA, and was acquired using an HP Photosmart M637 camera. This is typical of a modern, medium-quality camera.

How can an individual pixel be accessed? The field `widthStep` is the size of a row, so the start of image row `i` would be found at

```
img->imageData + i*img->widthStep
```

Column `j` is `j` pixels along from this location; if pixels are bytes, then that's

```
img->imageData + i*img->widthStep + j
```

If pixels are RGB values, as in the JPEG image read in above, then each pixel is 3 bytes long and pixel `j` starts at location

```
img->imageData + i*img->widthStep + j*3
```

The value of the field `nChannels` is essentially the number of bytes per pixel, so the pixel location can be generalized as:

```
img->imageData + i*img->widthStep))[j*img->nChannels]
```

Finally, the color components are in the order blue, green, and red. Thus, the blue value for pixel [i,j] is found at

```
(img->imageData + i*img->widthStep)[j*img->nChannels + 0]
```

and green and red at the following, respectively:

```
(img->imageData + i*img->widthStep)[j*img->nChannels + 1]
(img->imageData + i*img->widthStep)[j*img->nChannels + 2]
```

The data type for a pixel will be unsigned character (or `uchar`).

There is a generic way to access pixels in an image that automatically uses what is known about the image and its format and returns or modifies a specified pixel. This is quite handy, because pixels can be bytes, RGB, float, or

double in type. The function `cvGet2D` does this; getting the pixel value at `i,j` for the image above is simply

```
p = cvGet2D (img, i, j);
```

The variable `p` is of type `CvScalar`, which is

```
struct CvScalar
{
   double val[4];
}
```

If the pixel has only a single value (i.e., grey), then p.val[0] is that value. If it is RGB, then the color components of the pixel are as follows:

- Blue is `p.val[0]`
- Green is `p.val[1]`
- Red is `p.val[2]`

Modifying the pixel value is done as follows:

```
p.val[0] = 0;           // Blue
p.val[1] = 255;         // Green
p.val[2] = 255;         // Red
cvSet2D(img,i,j,p);     // Set the (i,j) pixel to yellow
```

This is referred to as *indirect access* in OpenCV documentation and is slower than other means of accessing pixels. It is, on the other hand, clean and clear.

1.2.2 Reading and Writing Images

The basic function for image input has already been seen; `cvLoadImage` reads an image from a file, given a path name to that file. It can read images in JPEG, BMP, PNM, PNG, and TIF formats, and does so automatically, without the need to specify the file type. This is determined from the data on the file itself. Once read, a pointer to an `IplImage` structure is returned that will by default be forced into a 3-channel RGB form, such as has been described previously. So, the call

```
img = cvLoadImage (filename);
```

returns an `IplImage*` value that is an RGB image, unless the file name indicated by the string variable `filename` can't be read, in which case the function returns `0` (null). A second parameter can be used to change the default return image. The call

```
img = cvLoadImage (filename, f);
```

returns a 1 channel (1 byte per pixel) grey-level image if f=0, and returns the actual image type that is found in the file if f<0.

Writing an image to a file can be simple or complex, depending on what the user wants to accomplish. Writing grey-level or RGB color images is simple, using the code:

```
k = cvSaveImage( filename, img );
```

The `filename` is, as usual, a string indicating the name of the file to be saved, and the `img` variable is the image to be written to that file. The file type will correspond to the suffix on the file, so if the `filename` is `file.jpg`, then the file format will be JPEG. If the file cannot be written, then the function returns `0`.

1.2.3 Image Display

If the basic C/C++ compiler is used alone, then displaying an image is quite involved. One of the big advantages in using OpenCV is that it provides easy ways to call functions that open a window and display images within it. This does not require the use of other systems, such as Tcl/Tk or Java, and asks the programmer to have only a basic knowledge of the underlying system for managing windows on their computer.

The user interface functions of OpenCV are collected into a library named `highgui`, and are documented on the Internet and in books. The basics are as follows: a window is created using the `cvNamedWindow` function, which specifies a name for the window. All windows are referred to by their name and not through pointers. When created, the window can be given the `autosize` property or not. Following this, the function `cvShowImage` can be used to display an image (as specified by an `IplImage` pointer) in an existing window. For windows with the `autosize` property, the window will change size to fit the image; otherwise, the image will be scaled to fit the window.

Whenever `cvShowimage` is called, the image passed as a parameter is displayed in the given window. In this way, consecutive parts of the processing of an image can be displayed, and simple animations can be created and displayed. After a window has been created, it can be moved to any position on the screen using `cvMoveWindow (name, x, y)`. It can also be moved using the mouse, just like any other window.

1.2.4 An Example

It is now possible to write a simple OpenCV program that will read, process, and display an image. The input image will be that of Figure 1.1, and the goal will be to threshold it.

First, add the needed `include` files, declare an image, and read it from a file.

```
// Threshold a color image.

#include "stdafx.h"
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <cv.h>
#include <highgui.h>


int main (int argc, char* argv[])
{
    IplImage *image = 0;
     int i,j,k;
     int mean=0, count=0;
     char c;

     image = cvLoadImage("C:/AIPCV/marchA062.jpg");
```

At this point, there should be image data pointed to by `image`. If so (if the image is not null), display it in a window, as before.

```
     if( image )
     {
        printf ("Height %d X with %d\n", image->height, image->width);
        cvNamedWindow( "mainWin", CV_WINDOW_AUTOSIZE);
        cvShowImage( "mainWin", image );
        printf ("Display of image is done.\n");
        cvWaitKey(0);          // wait for a key
```

Now perform the thresholding operation. But this is a color image, so convert it to grey first using the average of the three color components.

```
     for (i=0; i<image->height; i++)
       for (j=0; j<image->width; j++)
       {
         k=( (image->imageData+i*image->widthStep)[j*image->nChannels+0]
            +(image->imageData+i*image->widthStep)[j*image->nChannels+1]
            +(image->imageData+i*image->widthStep)[j*image->nChannels+2])/3;
         (image->imageData+i*image->widthStep)[j*image->nChannels+0]
              = (UCHAR) k;
         (image->imageData+i*image->widthStep)[j*image->nChannels+1]
              = (UCHAR) k;
         (image->imageData+i*image->widthStep)[j*image->nChannels+2]
              = (UCHAR) k;
```

At this point in the loop, count and sum the pixel values so that the mean can be determined later.

```
            mean += k;
            count++;
          }
```

Make a new window and display the grey image in it.

```
cvNamedWindow( "grey", CV_WINDOW_AUTOSIZE);
cvShowImage( "grey", image );
cvWaitKey(0);            // wait for a key
```

Finally, compute the mean level for use as a threshold and pass through the image again, setting pixels less than the mean to 0 and those greater to 255;

```
mean = mean/count;
for (i=0; i<image->height; i++)
  for (j=0; j<image->width; j++)
  {
    k=(image->imageData+i*image->widthStep)
                        [j * image->nChannels + 0];
    if (k < mean) k = 0;
     else k = 255;

    (image->imageData+i*image->widthStep)[j*image->nChannels+0]
      = (UCHAR) k;
    (image->imageData+i*image->widthStep)[j*image->nChannels+1]
      = (UCHAR) k;
    (image->imageData+i*image->widthStep)[j*image->nChannels+2]
      = (UCHAR) k;
  }
```

One final window is created, and the final thresholded image is displayed and saved.

```
 cvNamedWindow( "thresh");
 cvShowImage( "thresh", image );
 cvSaveImage( "thresholded.jpg", image );
```

Wait for the user to type a key before destroying all the windows and exiting.

```
 cvWaitKey(0);          // wait for a key

 cvDestroyWindow("mainWin");
 cvDestroyWindow("grey");
 cvDestroyWindow("thresh");
}
```

```
        else
          fprintf( stderr, "Error reading image\n" );
        return 0;
    }
```

Figure 1.2 shows a screen shot of this program.

Figure 1.2: The three image windows created by the thresholding program.

1.3 Image Capture

The processing of still photos or scientific images can be done quite effectively using scanned image or data from digital cameras. The availability of digital image data has increased many-fold over the past decade, and it is no longer unusual to find a digital camera, a scanner, and a video camera in a typical household or small college laboratory. Other kinds of data and other devices can be quite valuable sources of images for a vision system, key among these the *webcam*. These are digital cameras, almost always USB powered, having image sizes of 640x480 or larger. They acquire color images at video rates, making such cameras ideal for certain vision applications: surveillance,

robotics, games, biometrics, and places where computers are easily available and very high quality is not essential.

There are a great many types of webcam, and the details of how they work are not relevant to this discussion. If a webcam is properly installed, then OpenCV should be able to detect it, and the capture functions should be able to acquire images from it. The scheme used by OpenCV is to first declare and initialize a camera, using a handle created by the system. Assuming that this is successful, images can be captured through the handle.

Initializing a camera uses the `cvCaptureFromCAM` function:

```
CvCapture  *camera = 0;
camera = cvCaptureFromCAM( CV_CAP_ANY );
if( !camera )           error ...
```

The type `CvCapture` is internal, and represents the handle used to capture images. The function `cvCaptureFromCam` initializes capturing a video from a camera, which is specified using the single parameter. `CV_CAP_ANY` will allow any connected camera to be used, but the system will choose which one. If 0 is returned, then no camera was seen, and image capture is not possible; otherwise, the camera's handle is returned and is needed to grab images.

A frame (image) can be captured using the `cvQueryFrame` function:

```
IplImage   *frame = 0;
frame = cvQueryFrame( camera );
```

The image returned is an `IplImage` pointer, which can be used immediately.

When the program is complete, it is always a good idea to free any resources allocated. In this case, that means releasing the camera, as follows:

```
cvReleaseCapture( &camera );
```

It is now possible to write a program that drives the webcam. Let's have the images displayed in a window so that the live video can be seen. When a key is pressed, the program will save the current image in a JPEG file named `VideoFrame`*xx*`.jpg`, where *xx* is a number that increases each time.

```
// Capture.c - image capture from a webcam
#include "stdafx.h"
#include "stdio.h"
#include "string.h"
#include "cv.h"
#include "highgui.h"

int main(int argc, char ** argv)
{
  CvCapture  *camera = 0;
```

```
IplImage   *frame = 0;
int         i, n=0;
char        filename[256];
char c;
```

Initialize the camera and check to make sure that it is working.

```
camera = cvCaptureFromCAM( CV_CAP_ANY );
if( !camera )                          // Get a camera?
{
  fprintf(stderr, "Can't initialize camera\n");
  return -1;
}
```

Open a window for image display.

```
cvNamedWindow("video", CV_WINDOW_AUTOSIZE);
cvMoveWindow ("video", 150, 200);
```

This program will capture 600 frames. At video rates of 30 FPS, this would be 20 seconds, although cameras do vary on this.

```
for(i=0; i<600; i++)
{
  frame = cvQueryFrame( camera );     // Get one frame.
  if( !frame )
  {
    fprintf(stderr, "Capture failed.\n");
  }
```

The following creates a short pause between frames. Without it, the images come in too fast, and in many cases nothing is displayed. `cvWaitKey` waits for a key press or for the time specified — in this case, 100 milliseconds.

```
  c = cvWaitKey(100);
```

Display the image we just captured in the window.

```
// Display the current frame.
   cvShowImage("video", frame);
```

If `cvWaitKey` actually caught a key press, this means that the image is to be saved. If so, the character returned will be >0. Save it as a file in the `AIPCV` directory.

```
   if (c>0)
   {
      sprintf(filename, "C:/AIPCV/VideoFrame%d.jpg", n++);
      if( !cvSaveImage(filename, frame) )
```

```
        {
          fprintf(stderr, "Failed to save frame as '%s'\n", filename);
        } else
          fprintf (stderr, "Saved frame as 'VideoFrame%d.jpg'\n", n-1);
    }
}
```

Free the camera to avoid possible problems later.

```
 cvReleaseCapture( &camera );

// Wait for terminating keypress.
 cvWaitKey(0);
 return 0;
}
```

The data from the camera will be displayed at a rate of 10 frames/second, because the delay between frames (as specified by `cvWaitKey` is 100 milliseconds, or 100/1000 = 0.1 seconds. This means that the frame rate can be altered by changing this parameter, without exceeding the camera's natural maximum. Increasing this parameter decreases the frame rate. An example of how this program appears on the screen while running is given as Figure 1.3.

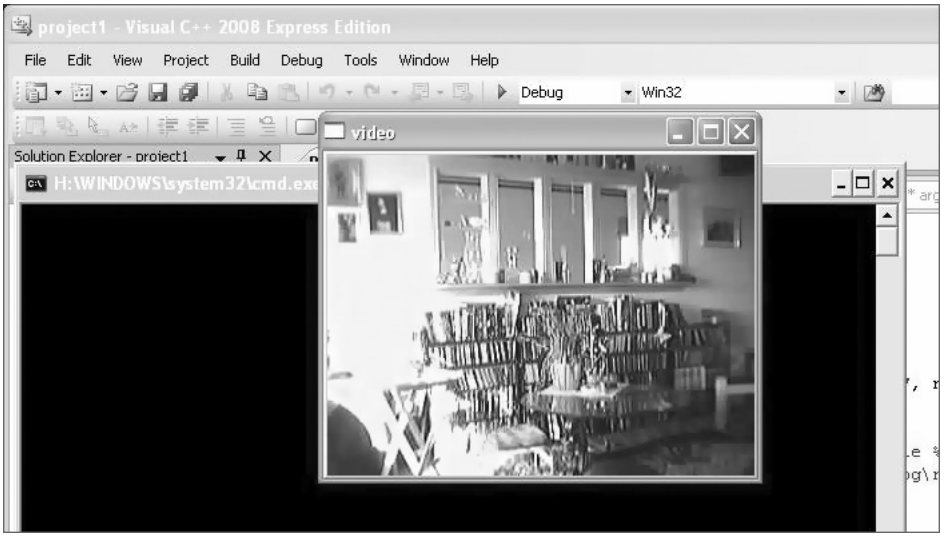

Figure 1.3: How the camera capture program looks on the screen. The image seems static, but it is really live video.

1.4 Interfacing with the AIPCV Library

This book discusses many algorithms, almost all of which are provided in source code form at the book's corresponding website. To access the examples and images on a PC, copy the directory `AIPCV` to the `C:` directory. Within that directory are many C source files that implement the methods discussed here. These programs are intended to be explanatory rather than efficient, and represent another way, a very precise way, to explain an algorithm. These programs comprise a library that uses a specific internal form for storing image data that was intended for use with grey-level images. It is not directly compatible with OpenCV, and so a conversion tool is needed.

OpenCV is not only exceptionally valuable for providing infrastructure to a vision system, but it also provides a variety of image-processing and computer vision functions. Many of these will be discussed in upcoming chapters (Canny and Sobel edge detection, for example), but many of the algorithms described here and provided in code form in the AIPCV library do not come with OpenCV. How can the two systems be used together?

The key detail when using OpenCV is knowledge of how the image structure is implemented. Thus, connecting OpenCV with the AIPCV library is largely a matter of providing a way to convert between the image structures of the two systems. This turns out to be quite simple for grey-level, one-channel images, and more complex for color images.

The basic image structure in the AIPCV library consists of two structures: a header and an image. The image structure, named simply `image`, consists of two pointers: one to a header and one to an array of pixel data:

```
struct image
{
  struct header *info;     // Pointer to header
  unsigned char **data;    // Pointer tp pixels
};
```

The pixel data is stored in the same way as for single-channel byte images in OpenCV: as a block of bytes addressed in row major order. It is set up to be indexed as a 2D array, however, so `data` is an array of pointers to rows. The variable `data[0]` is a pointer to the beginning of the entire array, and so is equivalent to `IplImage.imageData`.

The header is quite simple:

```
struct header
{
      int nr, nc;
      int oi, oj;
};
```

The field `nr` is the number of rows in the image, and `nc` is the number of columns. These are equivalent to `IplImage.height` and `IplImage.width`, respectively. The `oi` and `oj` fields specify the origin of the image, and are used only for a very few cases (e.g., restoration). There are no corresponding fields in OpenCV.

The way to convert an AIPCV image into an OpenCV image is now clear, and is needed so that images can be displayed in windows and saved in JPEG and other formats.

```
IplImage *toOpenCV (IMAGE x)
{
    IplImage *img;
    int i=0, j=0;
    CvScalar s;

    img=cvCreateImage(cvSize(x->info->nc,x->info->nr),8, 1);
    for (i=0; i<x->info->nr; i++)
    {
        for (j=0; j<x->info->nc; j++)
        {
            s.val[0] = x->data[i][j];
            cvSet2D (img, i,j,s);
        }
    }
    return img;
}
```

This function copies the pixel values into a new `IplImage`. It is also possible to use the original data array in the `IplImage` directly. There is some danger in this, in that OpenCV may decide to free the storage, for instance, making both versions inaccessible.

Converting from `IplImage` to AIPCV is more complicated, because OpenCV images might be in color. If so, how is it converted into grey? We'll not dwell on this except to say that one color image can be converted into three monochrome images (one each for red, green, and blue), or a color map could be constructed using a one-byte index that could be used as the pixel value. The solution presented here is to convert a 3-channel color image into grey by averaging the RGB values, leaving the other solutions for future consideration.

```
IMAGE fromOpenCV (IplImage *x)
{
    IMAGE img;
    int color=0, i=0;
    int k=0, j=0;
    CvScalar s;

    if ((x->depth==IPL_DEPTH_8U) &&(x->nChannels==1)) // Grey image
```

```
    img = newimage (x->height, x->width);
    else if ((x->depth==8) && (x->nChannels==3)) //Color
    {
        color = 1;
        img = newimage (x->height, x->width);
    }
    else return 0;

    for (i=0; i<x->height; i++)
    {
        for (j=0; j<x->width; j++)
        {
            s = cvGet2D (x, i, j);
            if (color)
               k= (unsigned char)  ((s.val[0]+s.val[1]+s.val[2])/3);
            else k = (unsigned char)(s.val[0]);
               img->data[i][j] = k;
         }
     }
     return img;
}
```

The two functions `toOpenCV` and `fromOpenCV` do the job of allowing the image-processing routines developed here to be used with OpenCV. As a demonstration, here is the main routine only for a program that thresholds an image using the method of grey-level histograms devised by Otsu and presented in Chapter 4. It is very much like the program for thresholding written earlier in Section 1.2.4, but instead uses the AIPCV library function `thr_glh` to find the threshold and apply it.

```
int main(int argc, char *argv[])
{
  IplImage* img=0;
  IplImage* img2=0;
  IMAGE x;
  int height,width,step,channels;
  uchar *data;
  int mean=0,count=0;

  if(argc<1){
    printf("Usage: main <image-file-name>\n\7");
    exit(0);
  }

  // load an image
  img=cvLoadImage("H:/AIPCV/marchA062.jpg");
```

```
  if(!img)
  {
    printf("Could not load image file: %s\n",argv[1]);
    exit(0);
  }

  // get the image data
  height    = img->height;
  width     = img->width;
  step      = img->widthStep;
  channels  = img->nChannels;
  data      = (uchar *)img->imageData;
  printf("Processing a %dx%d image with %dchannels\n",
         height,width,channels);

  // create a window
  cvNamedWindow("win1", CV_WINDOW_AUTOSIZE);
  cvMoveWindow("win1", 100, 100);

  // show the image
  cvShowImage("win1", img );

  // Convert to AIPCV IMAGE type
  x = fromOpenCV (img);
  if (x)
  {
    thr_glh (x);
    img2 = toOpenCV (x); // Convert to OpenCV to display
    cvNamedWindow( "thresh");
    cvShowImage( "thresh", img2 );
    cvSaveImage( "thresholded.jpg", img2 );
  }


  // wait for a key
  cvWaitKey(0);

  // release the image
  cvReleaseImage(&img);
  return 0;
}
```

In the remainder of this book, we will assume that OpenCV can be used for image display and I/O and that the native processing functions of OpenCV can be added to what has already been presented.

For convenience, the AIPCV library contains the following X functions for IO and display of its images directly to OpenCV:

`display_image (IMAGE x)`	Displays the specified image on the screen
`save_image (IMAGE x, char *name)`	Saves the image in a file with the given name
`IMAGE get_image (char *name)`	Reads the image in the named file and return a pointer to it
`IMAGE grab_image ()`	Captures an image from an attached webcam and return a pointer to it

1.5 Website Files

The website associated with this book contains code and data associated with each chapter, in addition to new information, errata, and other comments. Readers should create a directory for this information on their PC called `C:\AIPCV`. Within that, directories for each chapter can be named `CH1`, `CH2`, and so on.

The following material created for this chapter will appear in `C:\AIPCV\CH1`:

`capture.c`	Gets an image from a webcam
`lib0.c`	A collection of OpenCV input/output/display functions
`thr_glh.c`	Thresholds an image

1.6 References

Agam, Gady. ''Introduction to Programming With OpenCV,'' `www.edu/~agam/cs512/lect-notes/opencv-intro/opencv-intro.html` (accessed January 27, 2006).

Bradsky, Gary and Kaehler, Adrian. *Learning OpenCV: Computer Vision with the OpenCV Library*. Sebastopol: O'Reilly Media Inc, 2008.

''CV Reference Manual,'' `http://cognotics.com/opencv/docs/1.0/ref/opencvref_cv.htm` (accessed March 16, 2010).

''cvCam Reference Manual,'' `www.cognotics.com/opencv/docs/1.0/cvcam.pdf` (accessed March 16, 2010).

''CXCORE Reference Manual,'' `http://cognotics.com/opencv/docs/1.0/ref/opencvref_cxcore.htm` (accessed March 16, 2010).

''Experimental and Obsolete Functionality Reference,'' `http://cognotics.com/opencv/docs/1.0/ref/opencvref_cvaux.htm` (accessed March 16, 2010).

''HighGUI Reference Manual,'' `cognotics.com/opencv/docs/1.0/ref/opencvref_highgui.htm` (accessed March 16, 2010).

''OpenCV Wiki-Pages,'' `http://opencv.willowgarage.com/wiki`.

Otsu, N, ''A Threshold Selection Method from Grey-Level Histograms,'' *SMC* 9, no. 1 (1979): 62–66.

Parker, J. R. *Practical Computer Vision Using C*. New York: John Wiley & Sons, Inc., 1994.