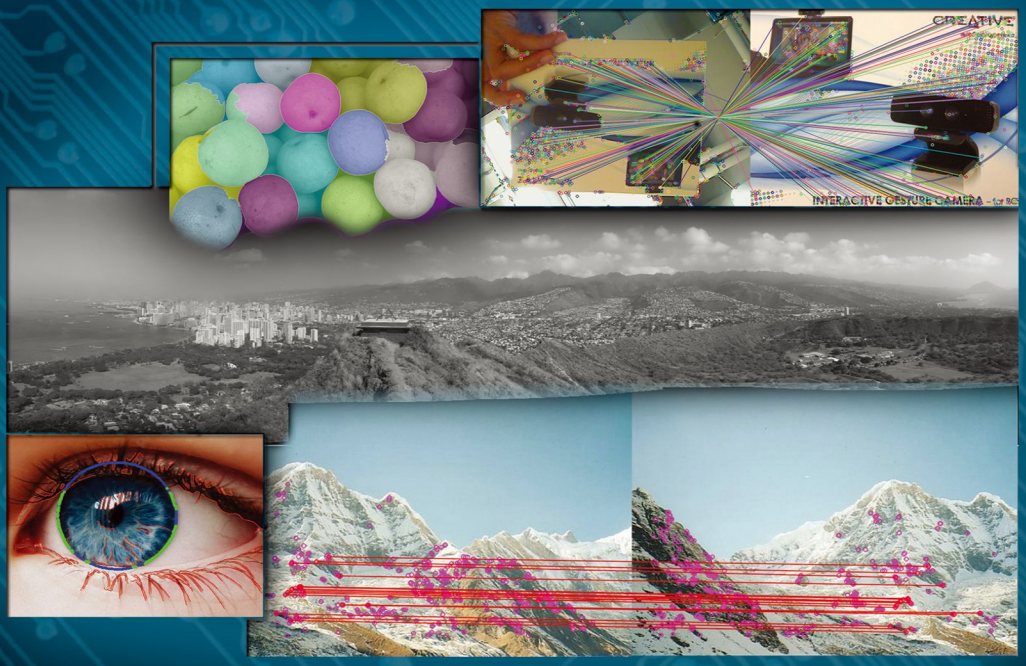
TECHNOLOGY IN ACTION™

Practical OpenCV

HANDS ON PROJECTS FOR COMPUTER VISION ON THE WINDOWS, LINUX AND RASPBERRY PI PLATFORMS



Samarth Brahmbhatt

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

About the Author xiii

About the Technical Reviewer xv

Acknowledgments xvii

■ Part 1: Getting Comfortable 1

■ Chapter 1: Introduction to Computer Vision and OpenCV 3

■ Chapter 2: Setting up OpenCV on Your Computer 7

■ Chapter 3: CV Bling—OpenCV Inbuilt Demos 13

■ Chapter 4: Basic Operations on Images and GUI Windows 23

■ Part 2: Advanced Computer Vision Problems and Coding Them in OpenCV 39

■ Chapter 5: Image Filtering 41

■ Chapter 6: Shapes in Images 67

■ Chapter 7: Image Segmentation and Histograms 95

■ Chapter 8: Basic Machine Learning and Object Detection Based on Keypoints 119

■ Chapter 9: Affine and Perspective Transformations and Their Applications to Image Panoramas 155

■ Chapter 10: 3D Geometry and Stereo Vision 173

■ Chapter 11: Embedded Computer Vision: Running OpenCV Programs on the Raspberry Pi 201

Index 219

PART 1

Getting Comfortable

CHAPTER 1

Introduction to Computer Vision and OpenCV

A significant share of the information that we get from the world while we are awake is through sight. Our eyes do a wonderful job of swiveling about incessantly and changing focus as needed to see things. Our brain does an even more wonderful job of processing the information stream from both eyes and creating a 3D map of the world around us and making us aware of our position and orientation in this map. Wouldn't it be cool if robots (and computers in general) could see, and understand what they see, as we do?

For robots, seeing in itself is less of a problem—cameras of all sorts are available and quite easy to use. However, to a computer with a camera connected to it, the camera feed is technically just a time-varying set of numbers.

Enter computer vision.

Computer vision is all about making robots intelligent enough to take decisions based on what they see.

Why Was This Book Written?

In my opinion, robots today are like personal computers 35 years ago—a budding technology that has the potential to revolutionize the way we live our daily lives. If someone takes you 35 years ahead in time, don't be surprised to see robots roaming the streets and working inside buildings, helping and collaborating safely with humans on a lot of daily tasks. Don't be surprised also if you see robots in industries and hospitals, performing the most complex and precision-demanding tasks with ease. And you guessed it right, to do all this they will need highly efficient, intelligent, and robust vision systems.

Computer vision is perhaps the hottest area of research in robotics today. There are a lot of smart people all around the world trying to design algorithms and implement them to give robots the ability to interpret what they see intelligently and correctly. If you too want to contribute to this field of research, this book is your first step.

In this book I aim to teach you the basic concepts, and some slightly more advanced ones, in some of the most important areas of computer vision research through a series of projects of increasing complexity. Starting from something as simple as making the computer recognize colors, I will lead you through a journey that will even teach you how to make a robot estimate its speed and direction from how the objects in its camera feed are moving.

We shall implement all our projects with the help of a programming library (roughly, a set of prewritten functions that can execute relevant higher-level tasks) called OpenCV.

This book will familiarize you with the algorithm implementations that OpenCV provides via its built-in functions, theoretical details of the algorithms, and the C++ programming philosophies that are generally employed while using OpenCV. Toward the end of the book, we will also discuss a couple of projects in which we employ OpenCV's framework for algorithms of our own design. A moderate level of comfort with C++ programming will be assumed.

OpenCV

OpenCV (Open-source Computer Vision, opencv.org) is the Swiss Army knife of computer vision. It has a wide range of modules that can help you with a lot of computer vision problems. But perhaps the most useful part of OpenCV is its architecture and memory management. It provides you with a framework in which you can work with images and video in any way you want, using OpenCV's algorithms or your own, without worrying about allocating and deallocating memory for your images.

History of OpenCV

It is interesting to delve a bit into why and how OpenCV was created. OpenCV was officially launched as a research project within Intel Research to advance technologies in CPU-intensive applications. A lot of the main contributors to the project included members of Intel Research Russia and Intel's Performance Library Team. The objectives of this project were listed as:

- Advance vision research by providing not only open but also optimized code for basic vision infrastructure. (No more reinventing the wheel!)
- Disseminate vision knowledge by providing a common infrastructure that developers could build on, so that code would be more readily readable and transferable.
- Advance vision-based commercial applications by making portable, performance-optimized code available for free—with a license that did not require the applications to be open or free themselves.

The first alpha version of OpenCV was released to the public at the IEEE Conference on Computer Vision and Pattern Recognition in 2000. Currently, OpenCV is owned by a nonprofit foundation called `OpenCV.org`.

Built-in Modules

OpenCV's built-in modules are powerful and versatile enough to solve most of your computer vision problems for which well-established solutions are available. You can crop images, enhance them by modifying brightness, sharpness and contrast, detect shapes in them, segment images into intuitively obvious regions, detect moving objects in video, recognize known objects, estimate a robot's motion from its camera feed, and use stereo cameras to get a 3D view of the world—to name just a few applications. If, however, you are a researcher and want to develop a computer vision algorithm of your own for which these modules themselves are not entirely sufficient, OpenCV will still help you a lot by its architecture, memory-management environment, and GPU support. You will find that your own algorithms working in tandem with OpenCV's highly optimized modules make a potent combination indeed.

One aspect of the OpenCV modules that needs to be emphasized is that they are highly optimized. They are intended for real-time applications and designed to execute very fast across a variety of computing platforms from MacBooks to small embedded fitPCs running stripped down flavors of Linux.

OpenCV provides you with a set of modules that can execute roughly the functionalities listed in Table 1-1.

Table 1-1. *Built-in modules offered by OpenCV*

Module	Functionality
Core	Core data structures, data types, and memory management
Imgproc	Image filtering, geometric image transformations, structure, and shape analysis
Highgui	GUI, reading and writing images and video
Video	Motion analysis and object tracking in video
Calib3d	Camera calibration and 3D reconstruction from multiple views
Features2d	Feature extraction, description, and matching
Objdetect	Object detection using cascade and histogram-of-gradient classifiers
ML	Statistical models and classification algorithms for use in computer vision applications
Flann	Fast Library for Approximate Nearest Neighbors—fast searches in high-dimensional (feature) spaces
GPU	Parallelization of selected algorithms for fast execution on GPUs
Stitching	Warping, blending, and bundle adjustment for image stitching
Nonfree	Implementations of algorithms that are patented in some countries

In this book, I shall cover projects that make use of most of these modules.

Summary

I hope this introductory chapter has given you a rough idea of what this book is all about! The readership I have in mind includes students interested in using their knowledge of C++ to program fast computer vision applications and in learning the basic theory behind many of the most famous algorithms. If you already know the theory, and are interested in learning OpenCV syntax and programming methodologies, this book with its numerous code examples will prove useful to you also.

The next chapter deals with installing and setting up OpenCV on your computer so that you can quickly get started with some exciting projects!

CHAPTER 2

Setting up OpenCV on Your Computer

Now that you know how important computer vision is for your robot and how OpenCV can help you implement a lot of it, this chapter will guide you through the process of installing OpenCV on your computer and setting up a development workstation. This will also allow you to try out and play with all the projects described in the subsequent chapters of the book. The official OpenCV installation wiki is available at http://opencv.willowgarage.com/wiki/InstallGuide, and this chapter will build mostly upon that.

Operating Systems

OpenCV is a platform independent library in that it will install on almost all operating systems and hardware configurations that meet certain requirements. However, if you have the freedom to choose your operating system I would advise a Linux flavor, preferably Ubuntu (the latest LTS version is 12.04). This is because it is free, works as well as (and sometimes better than) Windows and Mac OS X, you can integrate a lot of other cool libraries with your OpenCV project, and if you plan to work on an embedded system such as the Beagleboard or the Raspberry Pi, it will be your only option.

In this chapter I will provide setup instructions for Ubuntu, Windows, and Mac OSX but will mainly focus on Ubuntu. The projects themselves in the later chapters are platform-independent.

Ubuntu

Download the OpenCV tarball from http://sourceforge.net/projects/opencvlibrary/ and extract it to a preferred location (for subsequent steps I will refer to it as `OPENCV_DIR`). You can extract by using the Archive Manager or by issuing the tar –xvf command if you are comfortable with it.

Simple Install

This means you will install the current stable OpenCV version, with the default compilation flags, and support for only the standard libraries.

1. If you don't have the standard build tools, get them by

   ```
   sudo apt-get install build-essential checkinstall cmake
   ```

2. Make a build directory in OPENCV_DIR and navigate to it by

   ```
   mkdir build
   cd build
   ```

3. Configure the OpenCV installation by

   ```
   cmake ..
   ```

4. Compile the source code by

   ```
   make
   ```

5. Finally, put the library files and header files in standard paths by

   ```
   sudo make install
   ```

Customized Install (32-bit)

This means that you will install a number of supporting libraries and configure the OpenCV installation to take them into consideration. The extra libraries that we will install are:

- FFmpeg, gstreamer, x264 and v4l to enable video viewing, recording, streaming, and so on
- Qt for a better GUI to view images

1. If you don't have the standard build tools, get them by

   ```
   sudo apt-get install build-essential checkinstall cmake
   ```

2. Install gstreamer

   ```
   sudo apt-get install libgstreamer0.10-0 libgstreamer0.10-dev gstreamer0.10-tools
   gstreamer0.10-plugins-base libgstreamer-plugins-base0.10-dev gstreamer0.10-plugins-good
   gstreamer0.10-plugins-ugly gstreamer0.10-plugins-bad gstreamer0.10-ffmpeg
   ```

3. Remove any installed versions of ffmpeg and x264

   ```
   sudo apt-get remove ffmpeg x264 libx264-dev
   ```

4. Install dependencies for ffmpeg and x264

   ```
   sudo apt-get update
   sudo apt-get install git libfaac-dev libjack-jackd2-dev libmp3lame-dev
   libopencore-amrnb-dev libopencore-amrwb-dev libsdl1.2-dev libtheora-dev
   libva-dev libvdpau-dev libvorbis-dev libx11-dev libxfixes-dev libxvidcore-dev
   texi2html yasm zlib1g-dev libjpeg8 libjpeg8-dev
   ```

5. Get a recent stable snapshot of x264 from ftp://ftp.videolan.org/pub/videolan/x264/snapshots/, extract it to a folder on your computer and navigate into it. Then configure, build, and install by

   ```
   ./configure --enable-static
   make
   sudo make install
   ```

6. Get a recent stable snapshot of ffmpeg from http://ffmpeg.org/download.html, extract it to a folder on your computer and navigate into it. Then configure, build, and install by

   ```
   ./configure --enable-gpl --enable-libfaac --enable-libmp3lame
   --enable-libopencore-amrnb --enable-libopencore-amrwb --enable-libtheora
   --enable-libvorbis --enable-libx264 --enable-libxvid --enable-nonfree
   --enable-postproc --enable-version3 --enable-x11grab
   make
   sudo make install
   ```

7. Get a recent stable snapshot of v4l from http://www.linuxtv.org/downloads/v4l-utils/, extract it to a folder on your computer and navigate into it. Then build and install by

```
make
sudo make install
```

8. Install cmake-curses-gui, a semi-graphical interface to CMake that will allow you to see and edit installation flags easily

```
sudo apt-get install cmake-curses-gui
```

9. Make a build directory in OPENCV_DIR by

```
mkdir build
cd build
```

10. Configure the OpenCV installation by

```
ccmake ..
```

11. Press 'c' to start configuring. CMake-GUI should do its thing, discovering all the libraries you installed above, and present you with a screen showing the installation flags (Figure 2-1).

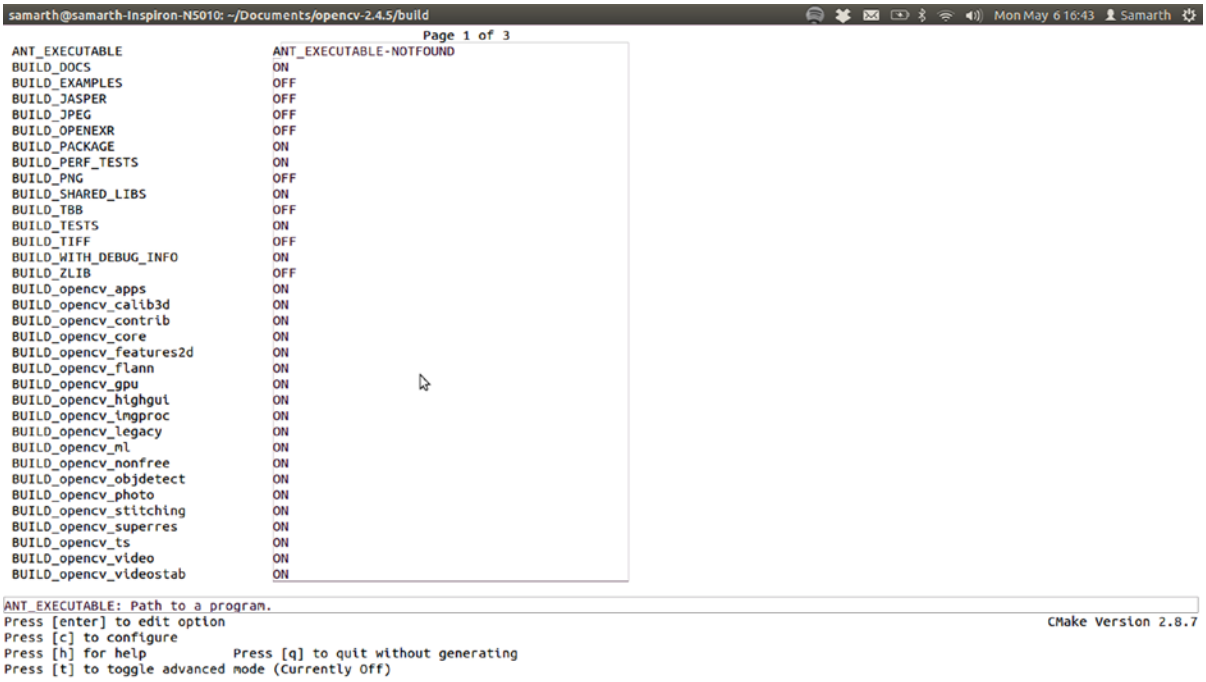


Figure 2-1. *Configuration flags when you start installing OpenCV*

12. You can navigate among the flags by the up and down arrows, and change the value of a flag by pressing the Return key. Change the following flags to the values shown in Table 2-1.

Table 2-1. *Configuration flags for installing OpenCV with support for other common libraries*

FLAG	VALUE
BUILD_DOCS	ON
BUILD_EXAMPLES	ON
INSTALL_C_EXAMPLES	ON
WITH_GSTREAMER	ON
WITH_JPEG	ON
WITH_PNG	ON
WITH_QT	ON
WITH_FFMPEG	ON
WITH_V4L	ON

13. Press 'c' to configure and 'g' to generate, and then build and install by

    ```
    make
    sudo make install
    ```

14. Tell Ubuntu where to find the OpenCV shared libraries by editing the file `opencv.conf` (first time users might not have that file—in that case, create it)

    ```
    sudo gedit /etc/ld.so.conf.d/opencv.conf
    ```

15. Add the line '`/usr/local/lib`' (without quotes) to this file, save and close. Bring these changes into effect by

    ```
    sudo ldconfig /etc/ld.so.conf
    ```

16. Similarly, edit /etc/bash.bashrc and add the following lines to the bottom of the file, save, and close:

    ```
    PKG_CONFIG_PATH=$PKG_CONFIG_PATH:/usr/local/lib/pkgconfig
    export PKG_CONFIG_PATH
    ```

Reboot your computer.

Customized Install (64-bit)

If you have the 64-bit version of Ubuntu, the process remains largely the same, except for the following changes.

1. During the step 5 to configure x264, use this command instead:

   ```
   ./configure --enable-shared –-enable-pic
   ```

2. During the step 6 to configure ffmpeg, use this command instead:

   ```
   ./configure --enable-gpl --enable-libfaac --enable-libmp3lame
   –-enable-libopencore-amrnb –-enable-libopencore-amrwb --enable-libtheora
   --enable-libvorbis --enable-libx264 --enable-libxvid --enable-nonfree
   --enable-postproc --enable-version3 --enable-x11grab –-enable-shared –-enable-pic
   ```

Checking the Installation

You can check the installation by putting the following code in a file called hello_opencv.cpp. It displays an image, and closes the window when you press "q":

```
#include <iostream>
#include <opencv2/highgui/highgui.hpp>
using namespace std;
using namespace cv;

int main(int argc, char **argv)
{
Mat im = imread("image.jpg", CV_LOAD_IMAGE_COLOR);
namedWindow("Hello");
imshow("Hello", im);

       cout << "Press 'q' to quit..." << endl;
       while(1)
          {
                  if(char(waitKey(1)) == 'q') break;
          }
          destroyAllWindows();
          return 0;
}
```

1. Open up that directory in a Terminal and give the following command to compile the code:

```
g++ 'pkg-config opencv --cflags' hello_opencv.cpp -o hello_opencv 'pkg-config opencv --libs'
```

2. Run the compiled code by

```
./hello_opencv
```

Note that you need to have an image called "image.jpg" in the same directory for this program to run.

Installing Without Superuser Privileges

Many times you do not have superuser access privileges to the computer you are working on. You can still install and use OpenCV, if you tell Ubuntu where to look for the library and header files. In fact, this method of using OpenCV is recommended over the previous method, as it does not "pollute" the system directories with conflicting versions of OpenCV files according to the official OpenCV installation Wiki page. Note that installing extra libraries such as Qt, Ffmpeg, and so on will still require superuser privileges. But OpenCV will still work without these add-ons. The steps involved are:

1. Download the OpenCV tarball and extract it to a directory where you have read/write rights. We shall call this directory `OPENCV_DIR`. Make the following directories in `OPENCV_DIR`

```
mkdir build
cd build
mkdir install-files
```

2. Configure your install as mentioned previously. Change the values of flags depending on which extra libraries you have installed in the system. Also, set the value of `CMAKE_INSTALL_PREFIX` to `OPENCV_DIR/build/install-files`.

3. Continue the same making process as the normal install, up to step 12. Then, run `make install` instead of `sudo make install`. This will put all the necessary OpenCV files in `OPENCV_DIR/build/install-files`.
4. Now, edit the file `~/.bashrc` (your local bashrc file over which you should have read/write access) and add the following lines to the end of the file, then save and close

```
export INCLUDE_PATH=<path-to-OPENCV_DIR>/build/install-files/include:$INCLUDE_PATH
export LD_LIBRARY_PATH=<path-to-OPENCV_DIR>/build/install-files/lib:$LD_LIBRARY_PATH
export PKG_CONFIG_PATH=<path-to-OPENCV_DIR>/build/install-files/lib/pkgconfig:$PKG_CONFIG_PATH
```

where `<path-to-OPENCV_DIR>` can for example be `/home/user/libraries/opencv/.`

1. Reboot your computer.
2. You can now compile and use OpenCV code as mentioned previously, like a normal install.

Using an Integrated Development Environment

If you prefer to work in an IDE rather than a terminal, you will have to configure the IDE project to find your OpenCV library files and header files. For the widely used Code::Blocks IDE, very good instructions are available at http://opencv.willowgarage.com/wiki/CodeBlocks, and the steps should be pretty much the same for any other IDE.

Windows

Installation instructions for Windows users are available at http://opencv.willowgarage.com/wiki/InstallGuide and they work quite well. Instructions for integration with MS Visual C++ are available at http://opencv.willowgarage.com/wiki/VisualC++.

Mac OSX

Mac OSX users can install OpenCV on their computers by following instructions at http://opencv.willowgarage.com/wiki/Mac_OS_X_OpenCV_Port.

Summary

So you see how much more fun installing software in Linux than it is in Windows and Mac OS X! Jokes apart, going through this whole process will give valuable insight to beginners about the internal workings of Linux and the use of Terminal. If, even after following the instructions to the dot, you have problems installing OpenCV, Google your error. Chances are very high that someone else has had that problem, too, and they have asked a forum about it. There are also a number of websites and detailed videos on YouTube explaining the installation process for Linux, Windows, and Mac OS X.

CHAPTER 3

CV Bling—OpenCV Inbuilt Demos

Now that you (hopefully) have OpenCV installed on your computer, it is time to check out some cool demos of what OpenCV can do for you. Running these demos will also serve to confirm a proper install of OpenCV.

OpenCV ships with a bunch of demos. These are in the form of C, C++, and Python code files in the `samples` folder inside `OPENCV_DIR` (the directory in which you extracted the OpenCV archive while installing; see Chapter 2 for specifics). If you specified the flag `BUILD_EXAMPLES` to be `ON` while configuring your installation, the compiled executable files should be present ready for use in `OPENCV_DIR/build/bin`. If you did not do that, you can run your configuration and installation as described in Chapter 2 again with the flag turned on.

Let us take a look at some of the demos OpenCV has to offer. Note that you can run these demos by

```
./<demo_name> [options]
```

where `options` is a set of command line arguments that the program expects, which is usually the file name. The demos shown below have been run on images that ship with OpenCV, which can be found in `OPENCV_DIR/samples/cpp`.

Note that all the commands mentioned below are executed after navigating to `OPENCV_DIR/build/bin`.

Camshift

Camshift is a simple object tracking algorithm. It uses the intensity and color histogram of a specified object to find an instance of the object in another image. The OpenCV demo first requires you to draw a box around the desired object in the camera feed. It makes the required histogram from the contents of this box and then proceeds to use the camshift algorithm to track the object in the camera feed. Run the demo by navigating to `OPENCV_DIR/build/bin` and doing

```
./cpp-example-camshiftdemo
```

However, camshift *always* tries to find an instance of the object. If the object is not present, it shows the nearest match as a detection (see Figure 3-4).

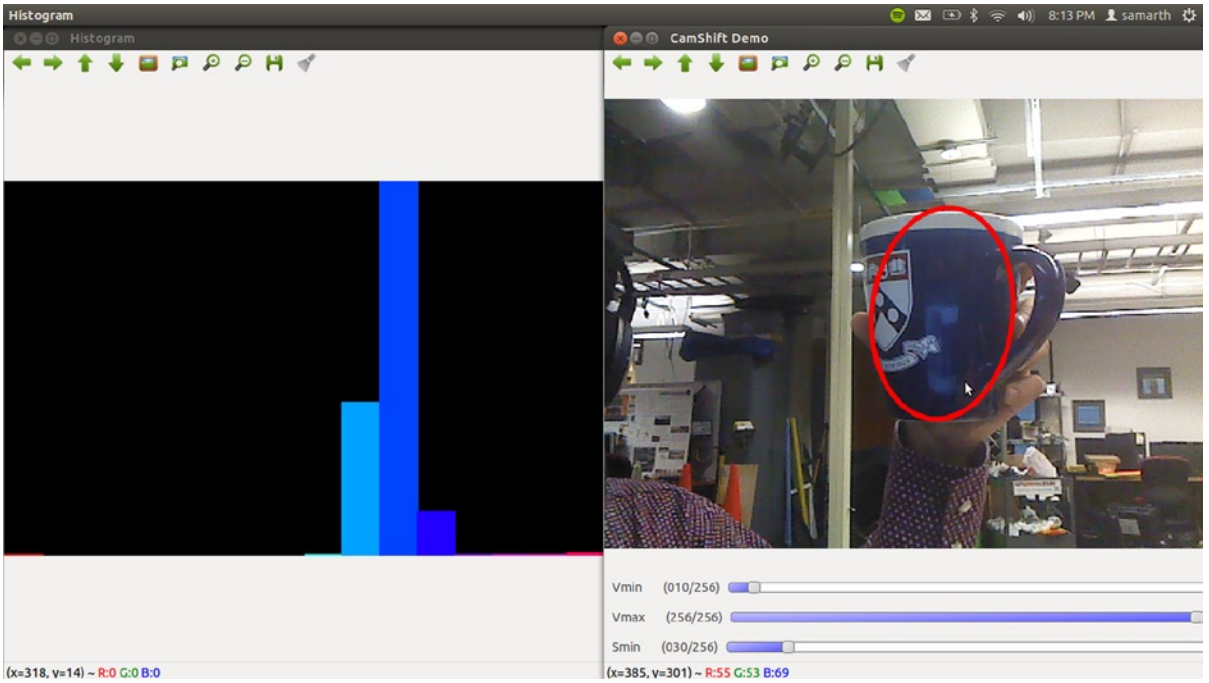


Figure 3-1. *Camshift object tracking—specifying the object to be tracked*

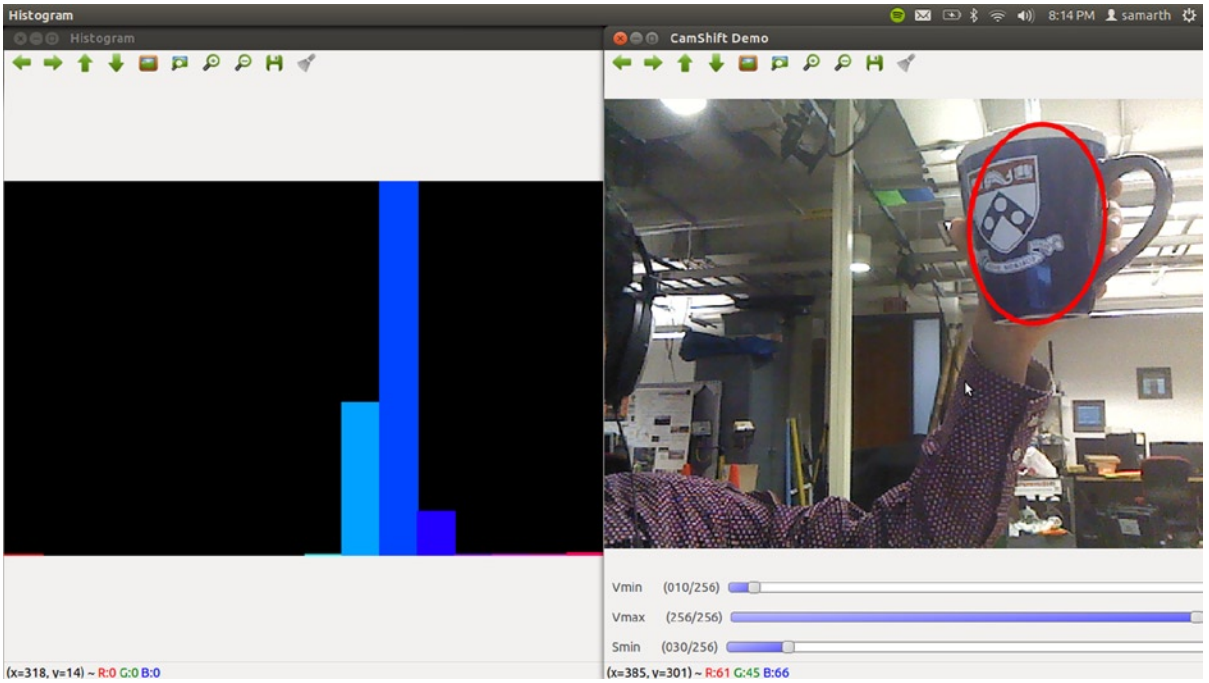


Figure 3-2. *Camshift object tracking*

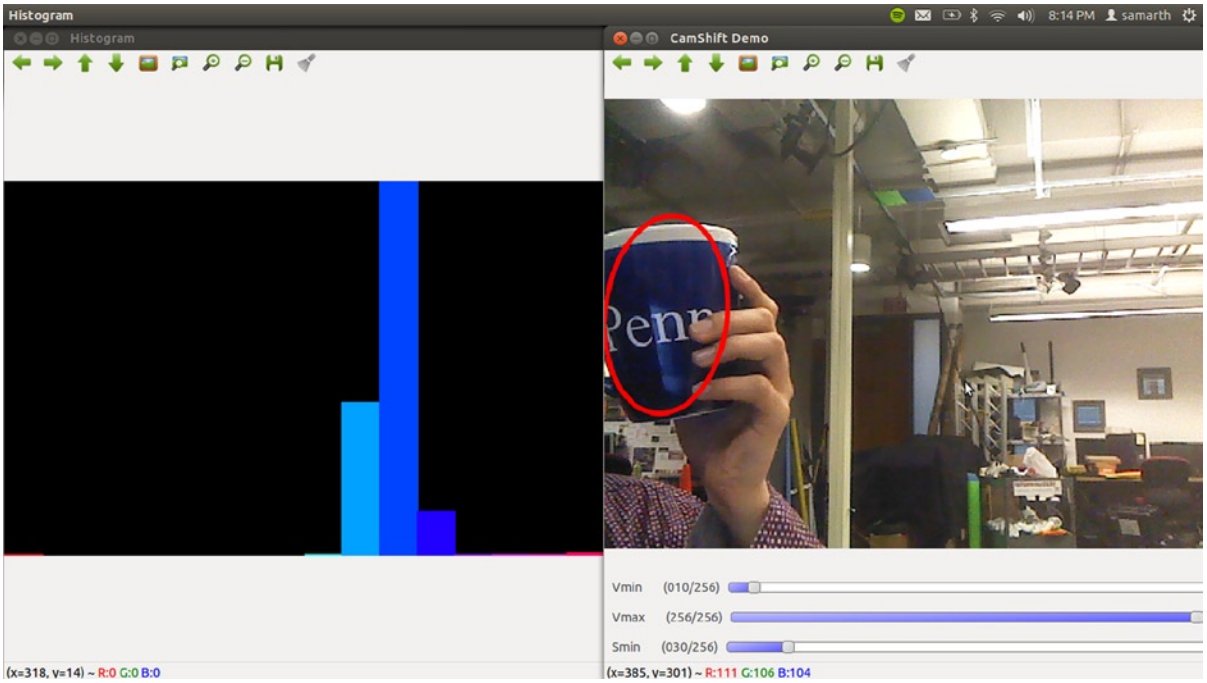


Figure 3-3. *Camshift object tracking*

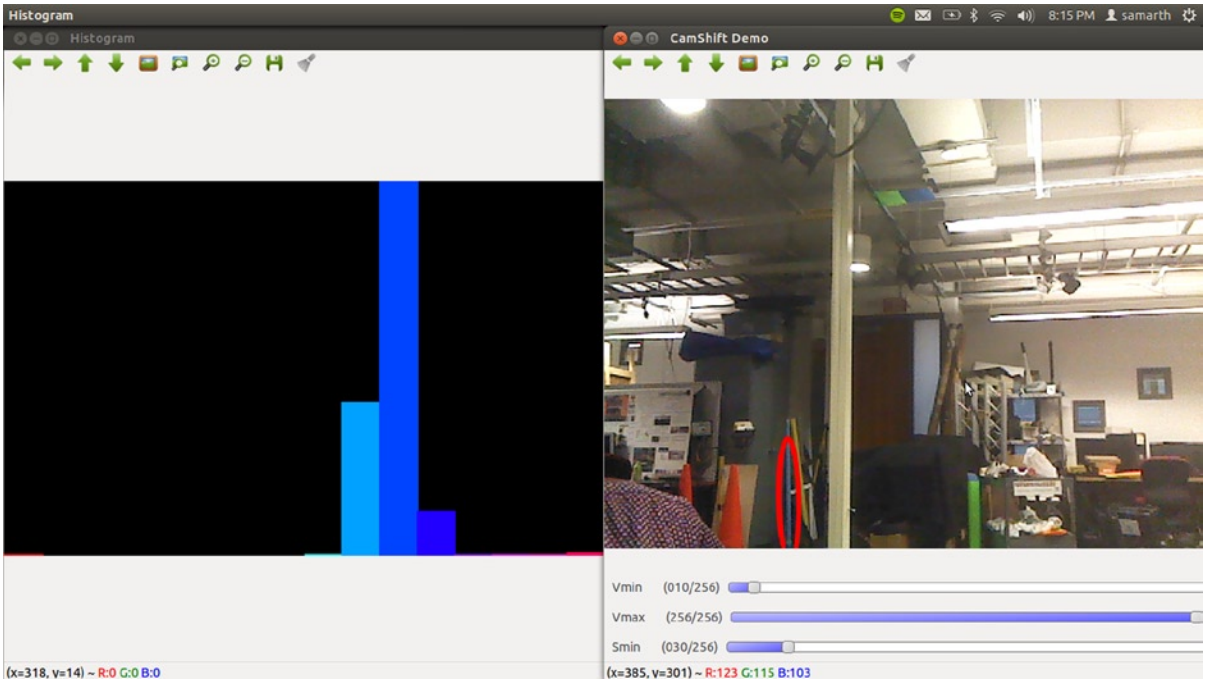


Figure 3-4. *Camshift giving a false positive*

Stereo Matching

The `stereo_matching` demo showcases the stereo block matching and disparity calculation abilities of OpenCV. It takes two images (taken with a left and right stereo camera) as input and produces an image in which the disparity is grey color-coded. I will devote an entire chapter to stereo vision later on in the book, Meanwhile, a short explanation of disparity: when you see an object using two cameras (left and right), it will appear to be at slightly different horizontal positions in the two images, The difference of the position of the object in the right frame with respect to the left frame is called disparity. Disparity can give an idea about the depth of the object, that is, its distance from the cameras, because disparity is inversely proportional to distance. In the output image, pixels with higher disparity are lighter. (Recall that higher disparity means lesser distance from the camera.) You can run the demo on the famous Tsukuba images by

```
./cpp-example-stereo_match OPENCV_DIR/samples/cpp/tsukuba_l.png OPENCV_DIR/samples/cpp/tsukuba_r.png
```

where `OPENCV_DIR` is the path to `OPENCV_DIR`

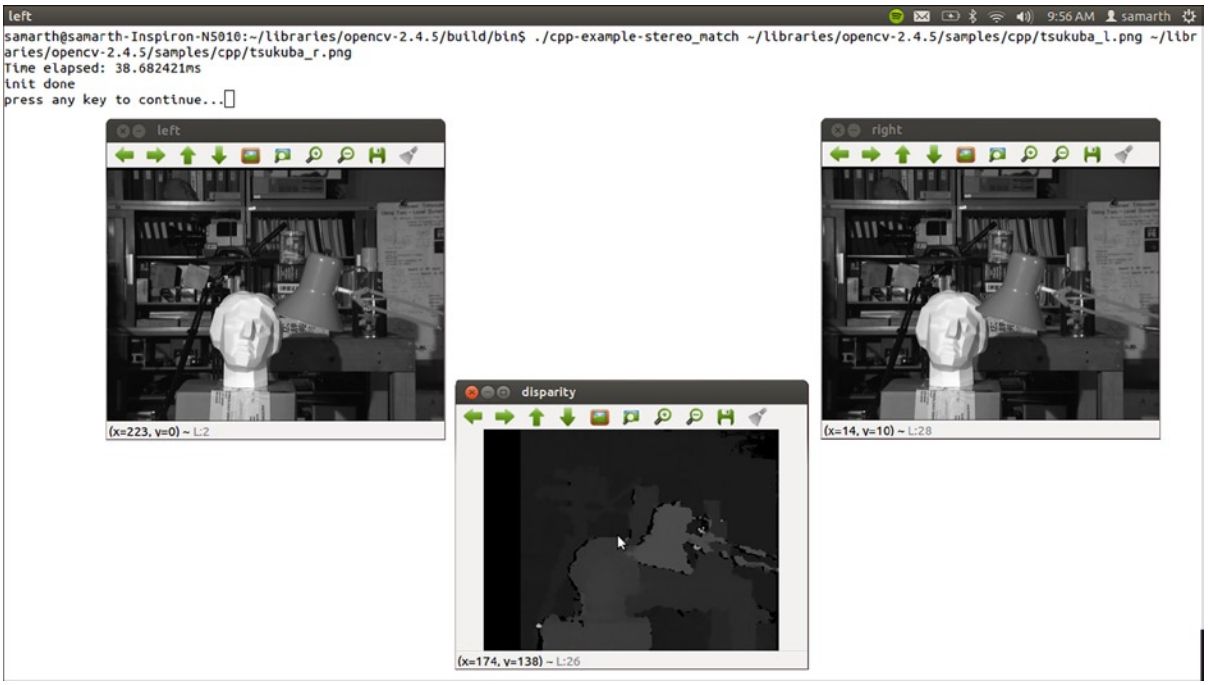


Figure 3-5. *OpenCV stereo matching*

Homography Estimation in Video

The `video_homography` demo uses the FAST corner detector to detect interest points in the image and matches BRIEF descriptors evaluated at the keypoints. It does so for a "reference" frame and any other frame to estimate the homography transform between the two images. A homography is simply a matrix that transforms points from one plane to another. In this demo, you can choose your reference frame from the camera feed. The demo draws lines in the direction of the homography transform between the reference frame and the current frame. You can run it by

```
./cpp-example-video_homography 0
```

where 0 is the device ID of the camera. 0 usually means the laptop's integrated webcam.

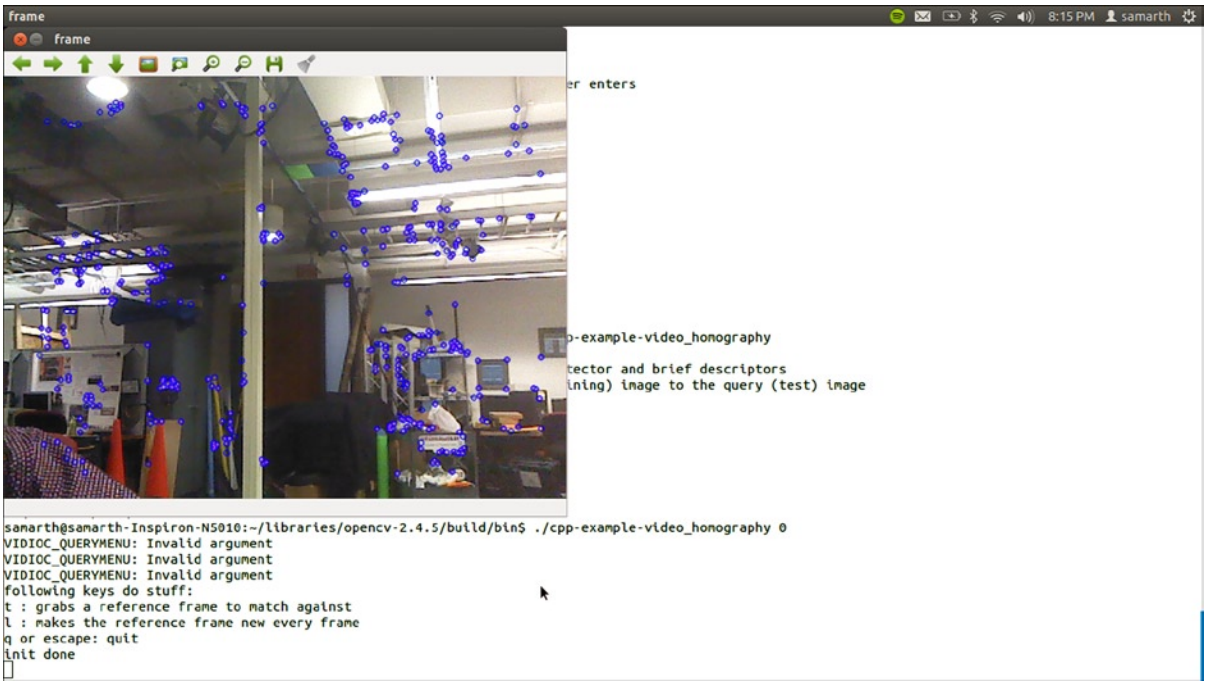


Figure 3-6. *The reference frame for homography estimation, also showing FAST corners*

Figure 3-7. *Estimated homography shown by lines*

Circle and Line Detection

The houghcircles and houghlines demos in OpenCV detect circles and lines respectively in a given image using the Hough transform. I shall have more to say on Hough transforms in Chapter 6. For now, just know that the Hough transform is a very useful tool that allows you to detect regular shapes in images. You can run the demos by

```
./cpp-example-houghcircles OPENCV_DIR/samples/cpp/board.jpg
```

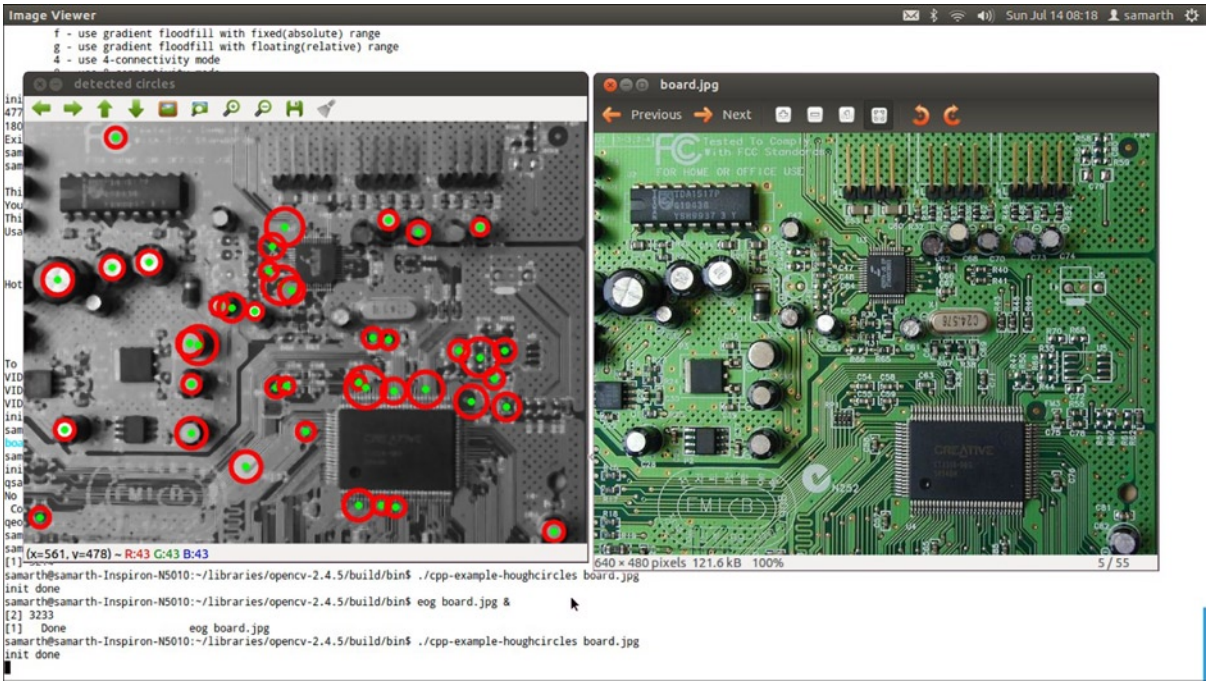


Figure 3-8. *Circle detection using Hough transform*

and

```
./cpp-example-houghlines OPENCV_DIR/samples/cpp/pic1.png
```

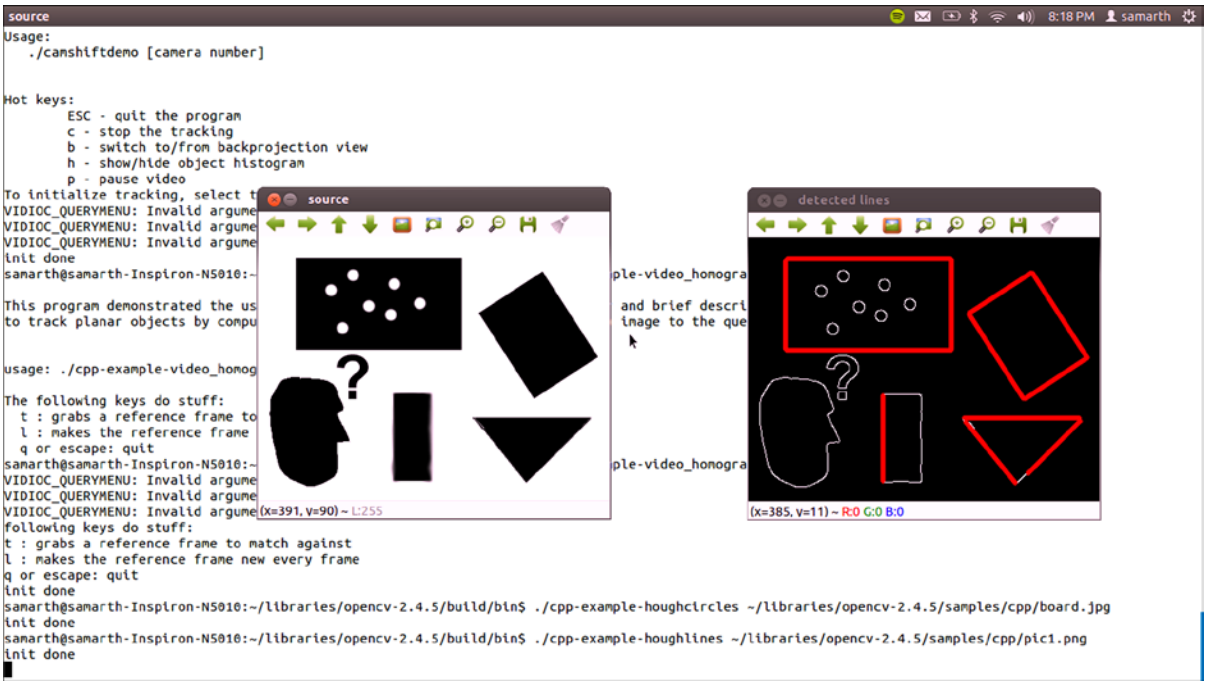


Figure 3-9. *Line detection using Hough transform*

Image Segmentation

The `meanshift_segmentation` demo implements the meanshift algorithm for image segmentation (distinguishing different "parts" of the image). It also allows you to set various thresholds associated with the algorithm. Run it by

```
./cpp-example-meanshift_segmentation OPENCV_DIR/samples/cpp/tsukuba_l.png
```

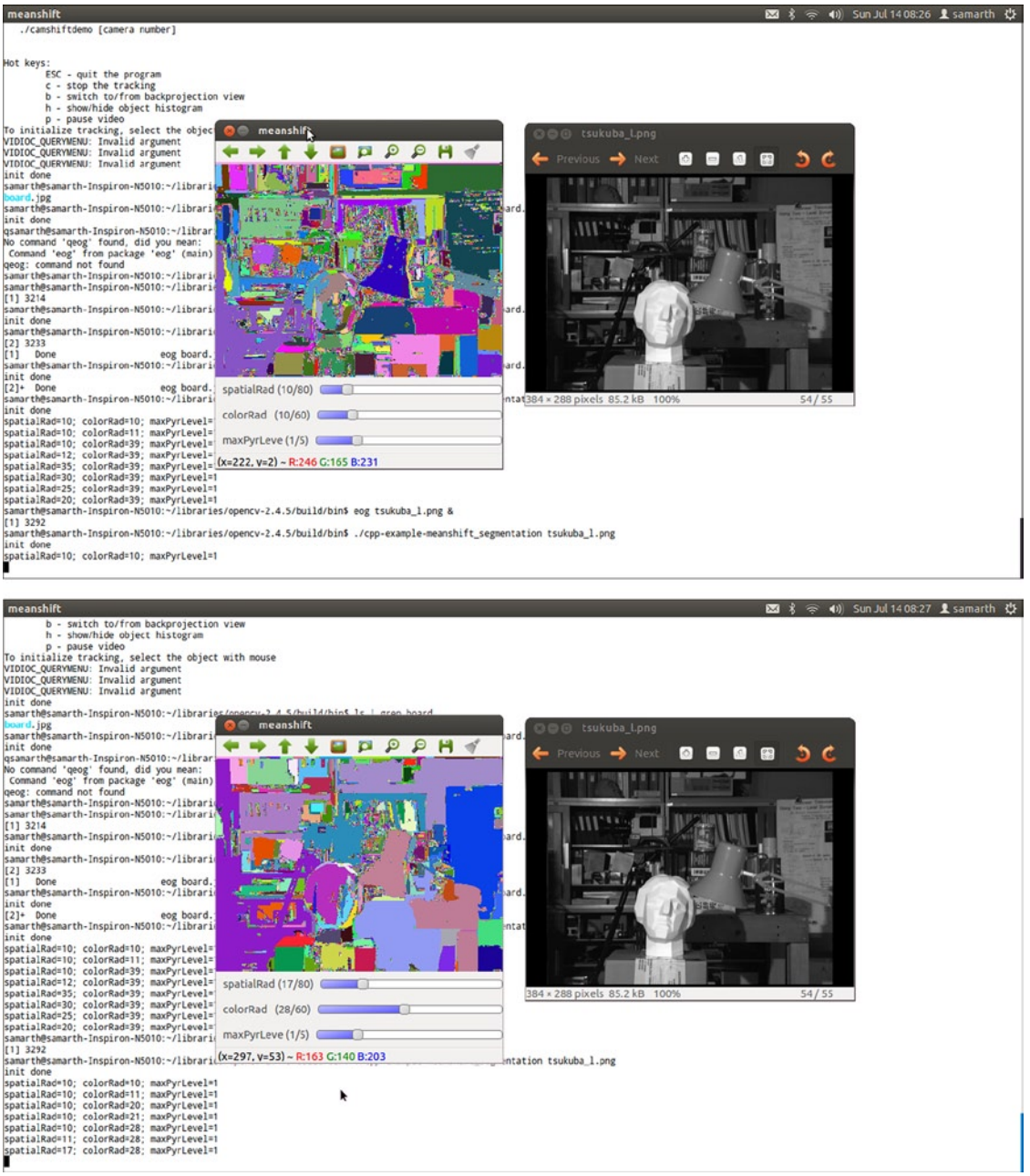


Figure 3-10. *Image segmentation using the meanshift algorithm*

As you can see, various regions in the image are colored differently.

Bounding Box and Circle

The `minarea` demo finds the smallest rectangle and circle enclosing a set of points. In the demo, the points are selected from within the image area randomly.

```
./cpp-example-minarea
```

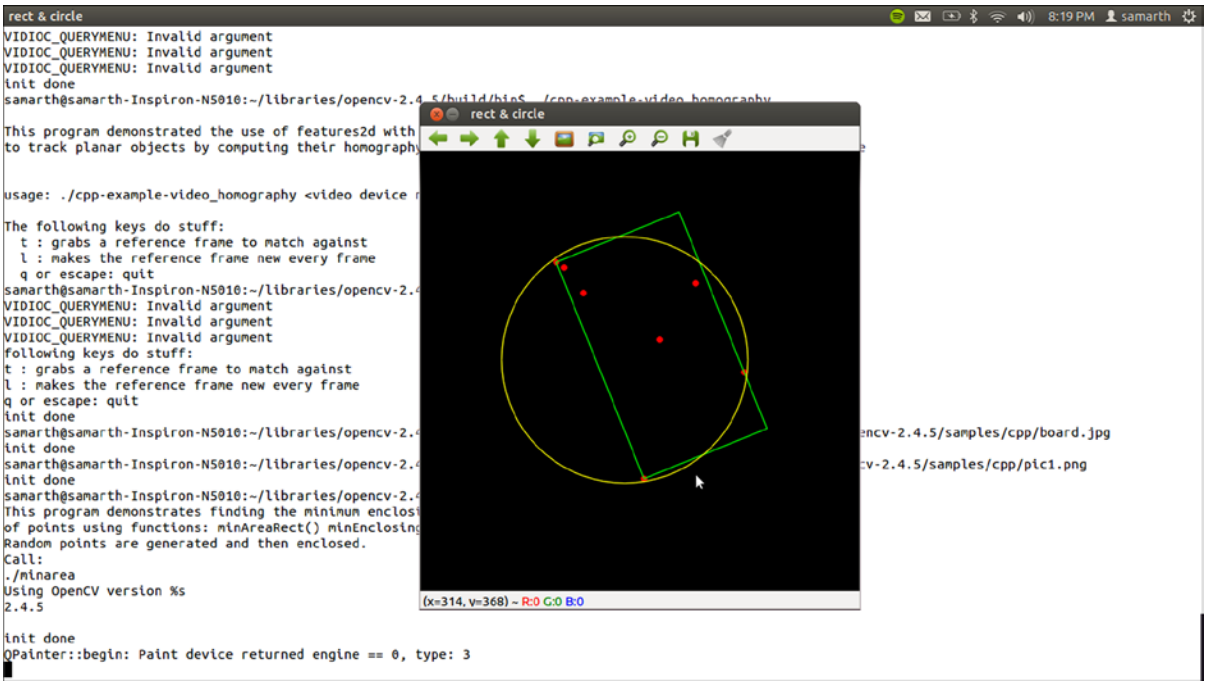


Figure 3-11. *Bounding box and circle*

Image Inpainting

Image inpainting is replacing certain pixels in the image with surrounding pixels. It is mainly used to repair damages to images such as accidental brush-strokes. The OpenCV `inpaint` demo allows you to vandalize an image by making white marks on it and then runs the inpainting algorithm to repair the damages.

```
./cpp-example-inpaint OPENCV_DIR/samples/cpp/fruits.jpg
```

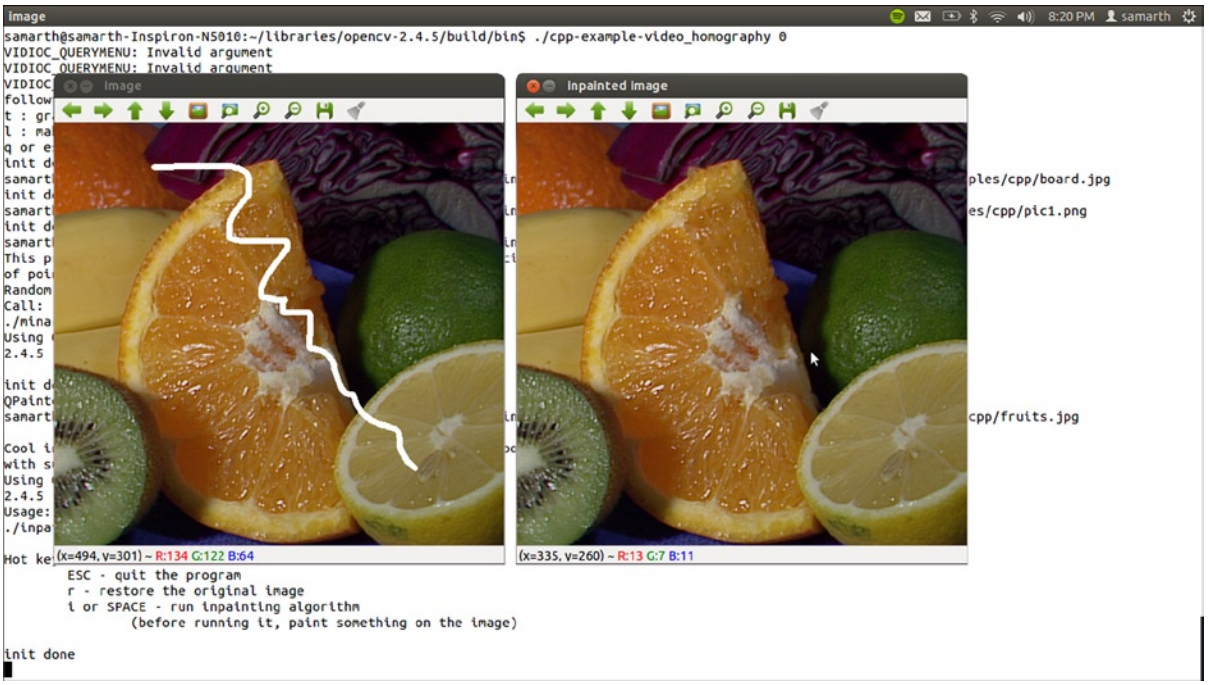


Figure 3-12. *Image inpainting*

Summary

The purpose of this chapter was to give you a glimpse of OpenCV's varied abilities. There are lots of other demos; feel free to try them out to get an even better idea. A particularly famous OpenCV demo is face detection using Haar cascades. Proactive readers could also go though the source code for these samples, which can be found in `OPENCV_DIR/samples/cpp`. Many of the future projects in this book will make use of code snippets and ideas from these samples.

CHAPTER 4

Basic Operations on Images and GUI Windows

In this chapter you will finally start getting your hands dirty with OpenCV code that you write yourself. We will start out with some easy tasks. This chapter will teach you how to:

- Display an image in a window
- Convert your image to/from color to grayscale
- Create GUI track-bars and write callback functions
- Crop parts from an image
- Access individual pixels of an image
- Read, display and write videos

Let's get started! From this chapter onward, I will assume that you know how to compile and run your code, that you are comfortable with directory/path management, and that you will put all the files that the program requires (e.g., input images) in the same directory as the executable file.

I also suggest that you use the OpenCV documentation at `http://docs.opencv.org/` extensively. It is not possible to discuss all OpenCV functions in all their forms and use-cases in this book. But the documentation page is where information about all OpenCV functions as well as their usage syntaxes and argument types is organized in a very accessible manner. So every time you see a new function introduced in this book, make it a habit to look it up in the docs. You will become familiar with the various ways of using that function and probably come across a couple of related functions, too, which will add to your repertoire.

Displaying Images from Disk in a Window

It is very simple to display disk images in OpenCV. The `highgui` module's `imread()`, `namedWindow()` and `imshow()` functions do all the work for you. Take a look at Listing 4-1, which shows an image in a window and exits when you press Esc or 'q' or 'Q' (it is exactly the same code we used to check the OpenCV install in Chapter 2):

Listing 4-1. Displaying an image in a window

```
#include <iostream>
#include <opencv2/highgui/highgui.hpp>
using namespace std;
using namespace cv;
```

```
int main(int argc, char **argv)
{
Mat im = imread("image.jpg", CV_LOAD_IMAGE_COLOR);
namedWindow("Hello");
imshow("Hello", im);

        cout << "Press 'q' to quit..." << endl;

        while(char(waitKey(1)) != 'q') {}
        return 0;
}
```

I'll now break the code down into chunks and explain it.

```
Mat im = imread("image.jpg", CV_LOAD_IMAGE_COLOR);
```

This creates a variable `im` of type `cv::Mat` (we write just `Mat` instead of `cv::Mat` because we have used namespace `cv;` above, this is standard practice). It also reads the image called `image.jpg` from the disk, and puts it into `im` through the function `imread()`. `CV_LOAD_IMAGE_COLOR` is a flag (a constant defined in the `highgui.hpp` header file) that tells `imread()` to load the image as a color image. A color image has 3 channels – red, green and blue as opposed to a grayscale image, which has just one channel—intensity. You can use the flag `CV_LOAD_IMAGE_GRAYSCALE` to load the image as grayscale. The type of `im` here is `CV_8UC3`, in which 8 indicates the number of bits each pixel in each channel occupies, `U` indicates unsigned character (each pixel's each channel is an 8-bit unsigned character) and `C3` indicates 3 channels.

```
namedWindow("Hello");
imshow("Hello", im);
```

First creates a window called Hello (Hello is also displayed in the title bar of the window) and then shows the image stored in `im` in the window. That's it! The rest of the code is just to prevent OpenCV from exiting and destroying the window before the user presses 'q' or 'Q'.

A noteworthy function here is `waitKey()`. This waits for a key event infinitely (when `n <= 0`) or for `n` milliseconds, when it is positive. It returns the ASCII code of the pressed key or `-1` if no key was pressed before the specified time elapses. Note that `waitKey()` works only if an OpenCV GUI window is open and in focus.

The cv::Mat Structure

The cv::Mat structure is the primary data structure used in OpenCV for storing data (image and otherwise). It is worthwhile to take a slight detour and learn a bit about how awesome cv::Mat is.

The cv::Mat is organized as a header and the actual data. Because the layout of the data is similar to or compatible with data structures used in other libraries and SDKs, this organization allows for very good interoperability. It is possible to make a cv::Mat header for user-allocated data and process it in place using OpenCV functions.

Tables 4-1, 4-2, and 4-3 describe some common operations with the cv::Mat structure. Don't worry about remembering it all right now; rather, read through them once to know about things you can do, and then use the tables as a reference.

Creating a cv::Mat

Table 4-1. *Creating a cv::Mat*

Syntax	Description
double m[2][2] = {{1.0, 2.0}, {3.0, 4.0}}; Mat M(2, 2, CV_32F, m);	Creates a 2 x 2 matrix from multidimensional array data
Mat M(100, 100, CV_32FC2, Scalar(1, 3));	Creates a 100 x 100 2 channel matrix, 1st channel filled with 1 and 2nd channel filled with 3
M.create(300, 300, CV_8UC(15));	Creates a 300 x 300 15 channel matrix, previously allocated data will be deallocated
int sizes[3] = {7, 8, 9}; Mat M(3, sizes, CV_8U, Scalar::all(0));	Creates a 3 dimensional array, where the size of each dimension is 7, 8 and 9 respectively. The array is filled with 0
Mat M = Mat::eye(7, 7, CV_32F);	Creates a 7 x 7 identity matrix, each element a 32 bit float
Mat M = Mat::zeros(7, 7, CV_64F);	Creates a 7 x 7 matrix filled with 64 bit float zeros. Similarly Mat::ones() creates matrices filled with ones

Accessing elements of a cv::Mat

Table 4-2. *Accessing elements from a cv::Mat*

Syntax	Description
M.at<double>(i, j)	Accesses the element at the ith row and jth column of M, which is a matrix of doubles. Note that row and column numbers start from 0
M.row(1)	Accesses the 1st row of M. Note that the number of rows starts from 0
M.col(3)	Accesses the 3rd column of M. Again, number of columns starts from 0
M.rowRange(1, 4)	Accesses the 1st to 4th rows of M
M.colRange(1, 4)	Accesses the 1st to 4th columns of M
M.rowRange(2, 5).colRange(1, 3)	Accesses the 2nd to 5th rows and 1st to 3rd columns of M
M.diag()	Accesses the diagonal elements of a square matrix. Can also be used to create a square matrix from a set of diagonal values

Expressions with cv::Mat

Table 4-3. *Expressions with a cv::Mat*

Syntax	Description
Mat M2 = M1.clone();	Makes M2 a copy of M1
Mat M2; M1.copyTo(M2);	Makes M2 a copy of M1
Mat M1 = Mat::zeros(9, 3, CV_32FC3); Mat M2 = M1.reshape(0, 3);	Makes M2 a matrix with same number of channels as M1 (indicated by the 0) and with 3 rows (and hence 9 columns)

(*continued*)

Table 4-3. (*continued*)

Syntax	Description
Mat M2 = M1.t();	Makes M2 a transpose of M1
Mat M2 = M1.inv();	Makes M2 the inverse of M1
Mat M3 = M1 * M2;	Makes M3 the matrix product of M1 and M2
Mat M2 = M1 + s;	Adds a scalar s to matrix M1 and stores the result in M2

Many more operations with cv::Mats can be found on the OpenCV documentation page at http://docs.opencv.org/modules/core/doc/basic_structures.html#mat.

Converting Between Color-spaces

A color-space is a way of describing colors in an image. The simplest color space is the RGB color space, which just represents the color of every pixel as a Red, a Green, and a Blue value, as red, green, and blue are primary colors and you can create all other colors by combining the three in various proportions. Usually, each "channel" is an 8-bit unsigned integer (with values ranging from 0 to 255); hence, you will find that most color images in OpenCV have the type CV_8UC3. Some common RGB triplets are described in Table 4-4.

Table 4-4. *Common RGB triplets*

Triplet	Color
(255, 0, 0)	Red
(0, 255, 0)	Green
(0, 0, 255)	Blue
(0, 0, 0)	Black
(255, 255, 255)	White

Another color space is grayscale, which is technically not a color space at all, because it discards the color information. All it stores is the intensity at every pixel, often as an 8-bit unsigned integer. There are a lot of other color spaces, and notable among them are YUV, CMYK, and LAB. (You can read about them on Wikipedia.)

As you saw previously, you can load images in either the RGB or grayscale color space by using the CV_LOAD_IMAGE_COLOR and CV_LOAD_IMAGE_GRAYSCALE flags, respectively, with imread(). However, if you already have an image loaded up, OpenCV has functions to convert its color space. You might want to convert between color spaces for various reasons. One common reason is that the U and V channels in the YUV color space encode all color information and yet are invariant to illumination or brightness. So if you want to do some processing on your images that needs to be illumination invariant, you should shift to the YUV color space and use the U and V channels (the Y channel exclusively stores the intensity information). Note that none of the R, G or B channels are illumination invariant.

The function cvtColor() does color space conversion. For example, to convert an RGB image in img1 to a grayscale image you would do:

```
cvtColor(img1, img2, CV_RGB2GRAY);
```

where CV_RGB2GRAY is a predefined code that tells OpenCV which conversion to perform. This function can convert to and from a lot of color spaces, and you can read up more about it at its OpenCV documentation page (http://docs.opencv.org/modules/imgproc/doc/miscellaneous_transformations.html?highlight=cvtcolor#cv.CvtColor).

GUI Track-Bars and Callback Functions

This section will introduce you to a very helpful feature of the OpenCV `highgui` module—track-bars or sliders, and the callback functions necessary to operate them. We will use the slider to convert an image from RGB color to grayscale and vice versa, so hopefully that will also reinforce your concepts of color-space conversions.

Callback Functions

Callback functions are functions that are called automatically when an event occurs. They can be associated with a variety of GUI events in OpenCV, like clicking the left or right mouse button, shifting the slider, and so on. For our color-space conversion application, we will associate a callback function with the shifting of a slider. This function gets called automatically whenever the user shifts the slider. In a nutshell, the function examines the value of the slider and display the image after converting its color-space accordingly. Although this might sound complicated, OpenCV makes it really simple. Let us look at the code in Listing 4-2.

Listing 4-2. Color-space conversion

```
// Function to change between color and grayscale representations of an image using a GUI trackbar
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania

#include <iostream>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

using namespace std;
using namespace cv;

// Global variables
const int slider_max = 1;
int slider;
Mat img;

// Callback function for trackbar event
void on_trackbar(int pos, void *)
{
        Mat img_converted;
        if(pos > 0) cvtColor(img, img_converted, CV_RGB2GRAY);
        else img_converted = img;

        imshow("Trackbar app", img_converted);
}
```

```
int main()
{
        img = imread("image.jpg");

        namedWindow("Trackbar app");
        imshow("Trackbar app", img);

        slider = 0;

        createTrackbar("RGB <-> Grayscale", "Trackbar app", &slider, slider_max, on_trackbar);
        while(char(waitKey(1)) != 'q') {}

        return 0;
}
```

As usual, I'll break up the code into chunks and explain.

The lines

```
const int slider_max = 1;
int slider;
Mat img;
```

declare global variables for holding the original image, slider position and maximum possible slider position. Since we want just two options for our slider—color and grayscale (0 and 1), and the minimum possible slider position is always 0, we set the maximum slider position to 1. The global variables are necessary so that both functions can access them.

The lines

```
img = imread("image.jpg");

namedWindow("Trackbar app");
imshow("Trackbar app", img);
```

in the main function simply read a color image called `image.jpg`, create a window called "Trackbar app" (a window is necessary to create a track-bar) and show the image in the window.

```
createTrackbar("RGB <-> Grayscale", "Trackbar app", &slider, slider_max, on_trackbar);
```

creates a track-bar with the name 'RGB <-> Grayscale' in the window called "Trackbar app" that we created earlier (you should look up this function in the OpenCV docs). We also pass a pointer to the variable that holds the starting value of the track-bar by using `&slider`, the maximum possible value of the track-bar and the associate the callback function called `on_trackbar` to track-bar events.

Now let us look at the callback function `on_trackbar()`, which (for a track-bar callback) must always be of the type `void foo(int. void *)`. The variable `pos` here holds the value of the track-bar and every time the user slides the track-bar, this function will be called with an updated value for `pos`. The lines

```
if(pos > 0) cvtColor(img, img_converted, CV_RGB2GRAY);
else img_converted = img;

imshow("Trackbar app", img_converted);
```

simply check the value of `pos` and display the proper image in the previously created window.

Compile and run your color-space converter application and if everything goes well, you should see it in action as shown in Figure 4-1.

Figure 4-1. *The color-space conversion app in action*

ROIs: Cropping a Rectangular Portion out of an Image

In this section you will learn about ROIs—Regions of Interest. You will then use this knowledge to make an app that allows you to select a rectangular part in an image crop it out.

Region of Interest in an Image

A region of interest is exactly what it sounds like. It is a region of the image in which we are particularly interested and would like to concentrate our processing on. It is used mainly in cases in which the image is too large and all parts of it are not relevant to our use; or the processing operation is so heavy that applying it to the whole image is computationally prohibitive. Usually a ROI is specified as a rectangle. In OpenCV a rectangular ROI is specified using a `rect` structure (again, look for `rect` in the OpenCV docs). We need the top-left corner position, width and height to define a `rect`.

Let us take a look at the code for our application (Listing 4-3) and then analyze it a bit at a time.

Listing 4-3. Cropping a part out of an image

```
// Program to crop images using GUI mouse callbacks
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania

#include <iostream>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

using namespace std;
using namespace cv;

// Global variables
// Flags updated according to left mouse button activity
bool ldown = false, lup = false;
// Original image
Mat img;
// Starting and ending points of the user's selection
Point corner1, corner2;
// ROI
Rect box;

// Callback function for mouse events
static void mouse_callback(int event, int x, int y, int, void *)
{
        // When the left mouse button is pressed, record its position and save it in corner1
        if(event == EVENT_LBUTTONDOWN)
        {
                ldown = true;
                corner1.x = x;
                corner1.y = y;
                cout << "Corner 1 recorded at " << corner1 << endl;
        }
```

```
        // When the left mouse button is released, record its position and save it in corner2
        if(event == EVENT_LBUTTONUP)
        {
                // Also check if user selection is bigger than 20 pixels (jut for fun!)
                if(abs(x - corner1.x) > 20 && abs(y - corner1.y) > 20)
                {
                        lup = true;
                        corner2.x = x;
                        corner2.y = y;
                        cout << "Corner 2 recorded at " << corner2 << endl << endl;
                }
                else
                {
                        cout << "Please select a bigger region" << endl;
                        ldown = false;
                }
        }

        // Update the box showing the selected region as the user drags the mouse
        if(ldown == true && lup == false)
        {
                Point pt;
                pt.x = x;
                pt.y = y;
                Mat local_img = img.clone();
                rectangle(local_img, corner1, pt, Scalar(0, 0, 255));
                imshow("Cropping app", local_img);
        }

        // Define ROI and crop it out when both corners have been selected
        if(ldown == true && lup == true)
        {
                box.width = abs(corner1.x - corner2.x);
                box.height = abs(corner1.y - corner2.y);
                box.x = min(corner1.x, corner2.x);
                box.y = min(corner1.y, corner2.y);

                // Make an image out of just the selected ROI and display it in a new window
                Mat crop(img, box);
                namedWindow("Crop");
                imshow("Crop", crop);

                ldown = false;
                lup = false;
        }
}
```

```
int main()
{
        img = imread("image.jpg");

        namedWindow("Cropping app");
        imshow("Cropping app", img);

        // Set the mouse event callback function
        setMouseCallback("Cropping app", mouse_callback);

        // Exit by pressing 'q'
        while(char(waitKey(1)) != 'q') {}

        return 0;
}
```

The code might seem big at the moment but, as you may have realized, most of it is just logical handling of mouse events. In the line

```
setMouseCallback("Cropping app", mouse_callback);
```

of the main function, we set the mouse callback to be the function called `mouse_callback`. The function `mouse_callback` basically does the following:

- Records the (x,y) position of the mouse when the left button is pressed.
- Records the (x,y) position of the mouse when the left button is released.
- Defines a ROI in the image when both of these have been done and shows another image made of just the ROI in another window (you could add in a feature that saves the ROI—use imwrite() for that).
- Draws the user selection and keeps updating it as the user drags the mouse with left button pressed.

Implementation is quite simple and self explanatory. I want to concentrate on three new programming features introduced in this program: the `Point`, the `rect`, and creating an image from another image by specifying a `rect` ROI.

The `Point` structure is used to store information about a point, in our case the corners of the user's selection. The structure has two data members, both `int`, called `x` and `y`. Other point structures such as `Point3d`, `Point2d`, and `Point3f` also exist in OpenCV and you should check them out in the OpenCV docs.

The `rect` structure is used to store information about a rectangle, using its `x`, `y`, `width`, and `height`. `x` and `y` here are the coordinates of the top left corner of the rectangle in the image.

If a `rect` called `r` holds information about a ROI in an image `M1`, you can extract the ROI in a new image `M2` using

```
Mat M2(M1, r);
```

The cropping app looks like Figure 4-2 in action.

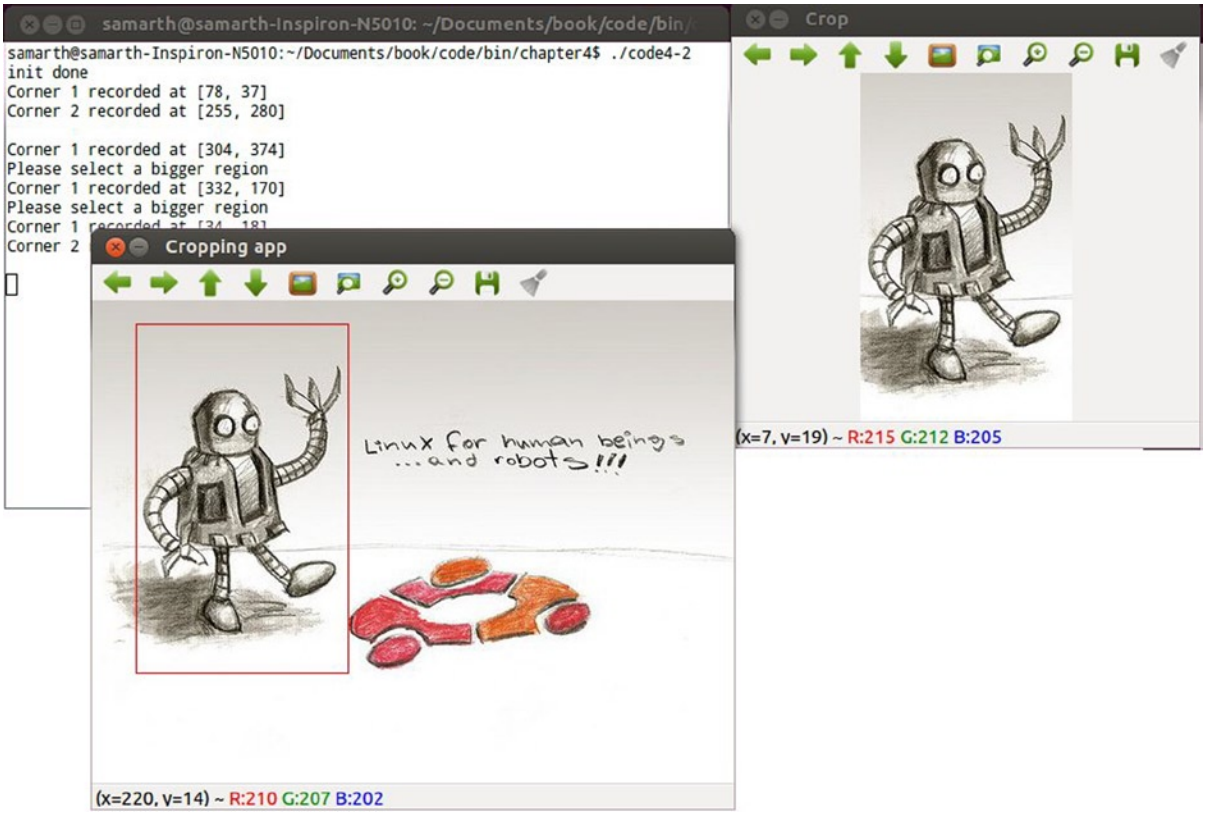


Figure 4-2. *The cropping app in action*

Accessing Individual Pixels of an Image

Sometimes it becomes necessary to access values of individual pixels in an image or its ROI. OpenCV has efficient ways of doing this. To access the pixel at position `(i, j)` in a `cv::Mat` image, you can use the `cv::Mat`'s `at()` attribute as follows:

For a grayscale image `M` in which every pixel is an 8-bit unsigned char, use `M.at<uchar>(i, j)`.

For a 3-channel (RGB) image `M` in which every pixel is a vector of 3 8-bit unsigned chars, use `M.at<Vec3b>[c]`, where `c` is the channel number, from 0 to 2.

Exercise

Can you make a very simple color image segmentation app based on the concepts you have learned so far?

Segmentation means identifying different parts of an image. Parts here are defined in the sense of color. We want to identify red areas in an image: given a color image, you should produce a black-and-white image output, the pixels of which are 255 (ON) at the red regions in the original image, and 0 (OFF) at the non-red regions.

You go through the pixels in the color image and check if their red value is in a certain range. If it is, turn ON the corresponding pixel of the output image. You can of course do it the simple way, by iterating through all pixels in the image. But see if you can go through the OpenCV docs and find a function that does exactly the same task for you. Maybe you can even make a track bar to adjust this range dynamically!

Videos

Videos in OpenCV are handled through FFMPEG support. Make sure that you have installed OpenCV with FFMPEG support before proceeding with the code in this section.

Displaying the Feed from Your Webcam or USB Camera/File

Let's examine a very short piece of code (Listing 4-4) that will display the video feed from your computer's default camera device. For most laptop computers, that is the integrated webcam.

Listing 4-4. Displaying the video feed from default camera device

```
// Program to display a video from attached default camera device
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania

#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;

int main()
{
        // 0 is the ID of the built-in laptop camera, change if you want to use other camera
        VideoCapture cap(0);

        //check if the file was opened properly
        if(!cap.isOpened())
        {
                cout << "Capture could not be opened successfully" << endl;
                return -1;
        }

        namedWindow("Video");

        // Play the video in a loop till it ends
        while(char(waitKey(1)) != 'q' && cap.isOpened())
        {
                Mat frame;
                cap >> frame;
                // Check if the video is over
                if(frame.empty())
                {
                        cout << "Video over" << endl;
                        break;
                }
                imshow("Video", frame);
        }

        return 0;
}
```

The code itself is self-explanatory, and I would like to touch on just a couple of lines.

```
VideoCapture cap(0);
```

This creates a VideoCapture object that is linked to device number 0 (default device) on your computer. And

```
cap >> frame;
```

extracts a frame from the device to which the VideoCapture object cap is linked. There are some other ways to extract frames from camera devices, especially when you have multiple cameras and you want to synchronize them (extract frames from all of them at the same time). I shall introduce such methods in Chapter 10.

You can also give a file name to the VideoCapture constructor, and OpenCV will play the video in that file for you exactly in the same manner (see Listing 4-5).

Listing 4-5. Program to display video from a file

```
// Program to display a video from a file
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania
// Video from: http://ftp.nluug.nl/ftp/graphics/blender/apricot/trailer/sintel_trailer-480p.mp4

#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;

int main()
{
        // Create a VideoCapture object to read from video file
        VideoCapture cap("video.mp4");

        //check if the file was opened properly
        if(!cap.isOpened())
        {
                cout << "Capture could not be opened succesfully" << endl;
                return -1;
        }

        namedWindow("Video");

        // Play the video in a loop till it ends
        while(char(waitKey(1)) != 'q' && cap.isOpened())
        {
                Mat frame;
                cap >> frame;
                // Check if the video is over
                if(frame.empty())
                {
                        cout << "Video over" << endl;
                        break;
                }
                imshow("Video", frame);
        }

        return 0;
}
```

Writing Videos to Disk

A `VideoWriter` object is used to write videos to disk. The constructor of this class requires the following as input:

- Output file name
- Codec of the output file. In the code that follows, we use the MPEG codec, which is very common. You can specify the codec using the `CV_FOURCC` macro. Four character codes of various codecs can be found at www.fourcc.org/codecs.php. Note that to use a codec, you must have that codec installed on your computer
- Frames per second
- Size of frames

You can get various properties of a video (like the frame size, frame rate, brightness, contrast, exposure, etc.) from a `VideoCapture` object using the `get()` function. In Listing 4-6, which writes video to disk from the default camera device, we use the `get()` function to get the frame size. You can also use it to get the frame rate if your camera supports it.

Listing 4-6. Code to write video to disk from the default camera device feed

```
// Program to write video from default camera device to file
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania

#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;

int main()
{
        // 0 is the ID of the built-in laptop camera, change if you want to use other camera
        VideoCapture cap(0);
        //check if the file was opened properly
        if(!cap.isOpened())
        {
                cout << "Capture could not be opened succesfully" << endl;
                return -1;
        }

        // Get size of frames
        Size S = Size((int) cap.get(CV_CAP_PROP_FRAME_WIDTH), (int)
cap.get(CV_CAP_PROP_FRAME_HEIGHT));

        // Make a video writer object and initialize it at 30 FPS
        VideoWriter put("output.mpg", CV_FOURCC('M','P','E','G'), 30, S);
        if(!put.isOpened())
        {
                cout << "File could not be created for writing. Check permissions" << endl;
                return -1;
        }
        namedWindow("Video");
```

```
        // Play the video in a loop till it ends
        while(char(waitKey(1)) != 'q' && cap.isOpened())
        {
                Mat frame;
                cap >> frame;
                // Check if the video is over
                if(frame.empty())
                {
                        cout << "Video over" << endl;
                        break;
                }
                imshow("Video", frame);
                put << frame;
        }

        return 0;
}
```

Summary

In this chapter, you got your hands dirty with lots of OpenCV code, and you saw how easy it is to program complex tasks such as showing videos in OpenCV. This chapter does not have a lot of computer vision. It was designed instead to introduce you to the ins and outs of OpenCV. The next chapter will deal with image filtering and transformations and will make use of the programming concepts you learned here.

PART 2

Advanced Computer Vision Problems and Coding Them in OpenCV

CHAPTER 5

Image Filtering

In this chapter, we will continue our discussion of basic operations on images. In particular, we will talk about some filter theory and different kinds of filters that you can apply to images in order to extract various kinds of information or suppress various kinds of noise.

There is a fine line between image processing and computer vision. Image processing mainly deals with getting different representations of images by transforming them in various ways. Often, but not always, this is done for "viewing" purposes—examples include changing the color space of the image, sharpening or blurring it, changing the contrast, affine transformation, cropping, resizing, and so forth. Computer vision, by contrast, is concerned with extracting information from images so that one can make *decisions*. Often, information has to be extracted from noisy images, so one also has to analyze the noise and think of ways to suppress that noise while not affecting the relevant information content of the image too much.

Take, for example, a problem where you have to make a simple wheeled automatic robot that can move in one direction, which tracks and intercepts a red colored ball.

The computer vision problem here is twofold: see if there is a red ball in the image acquired from the camera on the robot and, if yes, know its position relative to the robot along the direction of movement of the robot. Note that both these are decisive pieces of information, based on which the robot can take a decision whether to move or not and, if yes, which direction to move in.

Filters are the most basic operations that you can execute on images to extract information. (They can be extremely complex. too, but we will start out with simple ones.) To give you an overview of the content of this chapter, we will first start out with some image filter theory and then look into some simple filters. Applying these filters can serve as a useful pre- or postprocessing step in many computer vision pipelines. These operations include:

- Blurring
- Resizing images—up and down
- Eroding and dilating
- Detecting edges and corners

We will then discuss how to efficiently check for bounds on values of pixels in images. Using this newfound knowledge, we will then make our first very simple objector detector app. This will be followed by a discussion on the image morphology operations of opening and closing, which are useful tools in removing noise from images (we will demonstrate this by adding the opening and closing steps to our object detector app to remove noise).

Image Filters

A filter is nothing more than a function that takes the local value(s) of a signal and gives an output that is proportional in some way to the information that is contained in the signal. Usually, one "slides" the filter through the signal. To make these two important statements clear, consider the following one-dimensional time-varying signal, which could be the temperature of a city everyday (or something of that sort).

The information that we want to extract is temperature fluctuation; specifically, we want to see how drastically the temperature changes on a day-to-day basis. So we make a filter function that just gives the absolute value of the difference in today's temperature with yesterday's temperature. The equation that we follow is $y[n] = |x[n] - x[n-1]|$, where y[n] is the output of the filter at day n and x[n] is the signal, that is, the city's temperature at day n.

This filter (of length 2) is "slid" through the signal, and the output looks something like Figure 5-1.

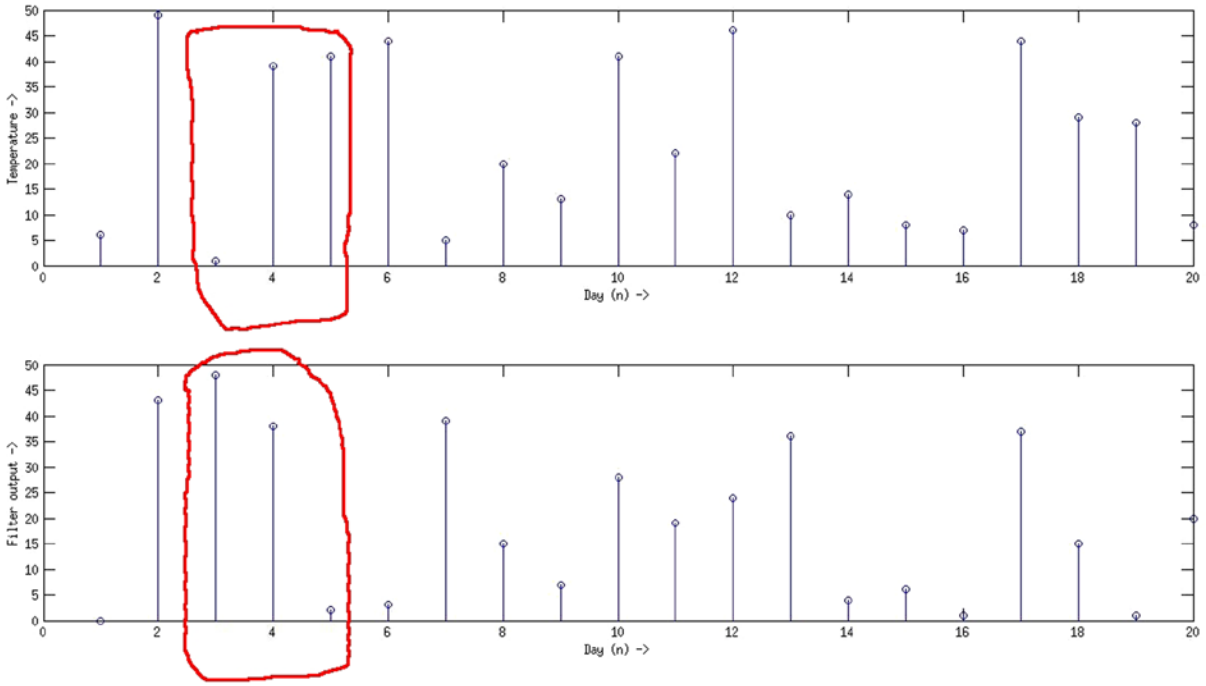


Figure 5-1. *A simple signal (above) and output of a differential filter on that signal (below)*

As you can observe, the filter enhanced the difference in the signal. Stated simply, if today's temperature is a lot different than yesterday, the filter's output for today will be higher. If today's temperature is almost the same as yesterday, the filter output today will be almost zero. Hopefully this very simple example convinces you that filter design is basically figuring out a function that will take in the values of the signal and enhance the chosen information content in it. There are some other conditions and rules you have to take care of, but we can ignore them for our simplistic applications.

Let's now move on to image filters, which are a bit different from the 1-D filter we discussed earlier, because the image signal is 2-D and hence the filter also has to be 2-D (if we want to consider pixel neighbors on all four sides). An example filter that detects vertical edges in the image will help you to understand better. The first step is to determine a filter matrix. A filter matrix is a discretized version of a filter function, and makes applying filters possible on computers. Their lengths and widths are usually odd numbers, so a center element can be unambiguously determined. For our case of detecting vertical edges, the matrix is quite simple:

```
 0    0    0
-1    2   -1
 0    0    0
```

Or if we want to consider two neighboring pixels:

```
 0     0     0     0     0
 0     0     0     0     0
-1    -2     6    -2    -1
 0     0     0     0     0
 0     0     0     0     0
```

Now, let's slide this filter through an image to see if it works! Before that, I must elaborate on what "applying" a filter matrix (or kernel) to an image means. The kernel is placed over the image, usually starting from the top-left corner of the image. The following steps are executed at every iteration:

- Element-wise multiplication is performed between elements of the kernel and pixels of the image covered by the kernel
- A function is used to calculate a single number using the result of all these element-wise multiplications. This function can be sum, average, minimum, maximum, or something very complicated. The value thus calculated is known as the "response" of the image to the filter at that iteration
- The pixel falling below the central element of the kernel assumes the value of the response
- The kernel is shifted to the right and if necessary down

Filtering an image made of just horizontal and vertical edges with this filter matrix (also called a kernel) gives us the filtered image shown in Figure 5-2.

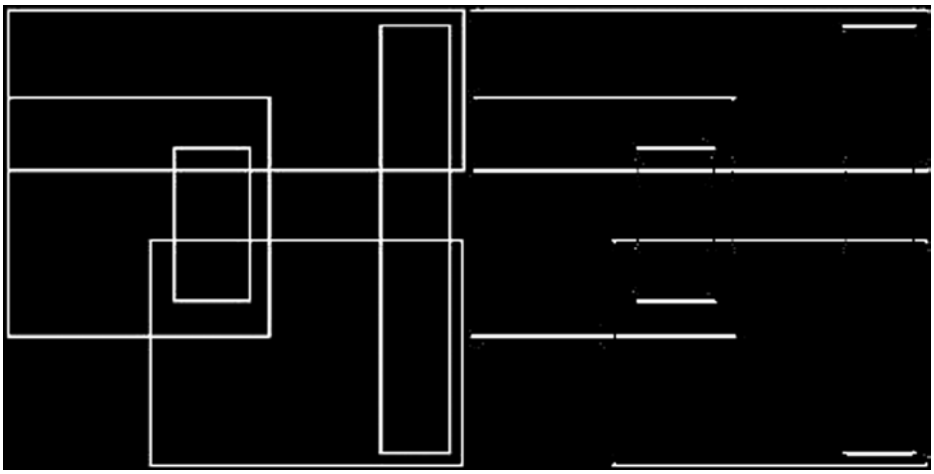


Figure 5-2. *A simple image with horizontal and vertical edges (left) and output of filtering with kernel (right)*

OpenCV has a function called `filter2D()` that we can use for efficient kernel-based filtering. To see how it is used, study the code used for the filtering discussed earlier, and also read up on its documentation. This function is quite powerful, because it allows you to filter an image by any kernel you specify. Listing 5-1 shows this function being used.

Listing 5-1. Program to apply a simple filter matrix to an image to detect horizontal edges

```
// Program to apply a simple filter matrix to an image
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>

using namespace std;
using namespace cv;

int main() {
    Mat img = imread("image.jpg", CV_LOAD_IMAGE_GRAYSCALE), img_filtered;

    // Filter kernel for detecting vertical edges
    float vertical_fk[5][5] = {{0,0,0,0,0}, {0,0,0,0,0}, {-1,-2,6,-2,-1}, {0,0,0,0,0}, {0,0,0,0,0}};
    // Filter kernel for detecting horizontal edges
    float horizontal_fk[5][5] = {{0,0,-1,0,0}, {0,0,-2,0,0}, {0,0,6,0,0}, {0,0,-2,0,0}, {0,0,-1,0,0}};

    Mat filter_kernel = Mat(5, 5, CV_32FC1, horizontal_fk);

    // Apply filter
    filter2D(img, img_filtered, -1, filter_kernel);

    namedWindow("Image");
    namedWindow("Filtered image");
    imshow("Image", img);
    imshow("Filtered image", img_filtered);

    // imwrite("filtered_image.jpg", img_filtered);

    while(char(waitKey(1)) != 'q') {}

    return 0;
}
```

As you might have guessed, the kernel to detect vertical edges is:

```
0       0       -1      0       0
0       0       -2      0       0
0       0        6      0       0
0       0       -2      0       0
0       0       -1      0       0
```

And it nicely detects vertical edges as shown in Figure 5-3.

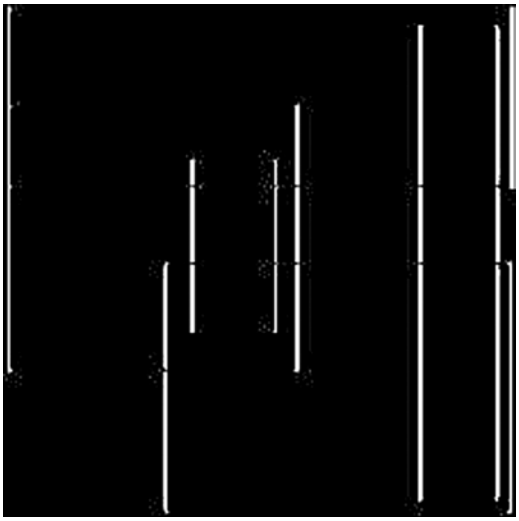


Figure 5-3. *Detecting vertical edges in an image*

It is fun to try making different detector kernels and experimenting with various images!

If you give a multichannel color image as input to `filter2D()`, it applies the same kernel matrix to all channels. Sometimes you might want to detect edges only in a certain channel or use different kernels for different channels and select the strongest edge (or average edge strength). In that case, you should split the image using the `split()` function and apply kernels individually.

Don't forget to check the OpenCV documentation on all the new functions you are learning!

Because edge detection is a very important operation in computer vision, a lot of research has been done to devise methods and intelligent filter matrices that can detect edges at any arbitrary orientation. OpenCV offers implementations of some of these algorithms, and I will have more to say on that later on in the chapter. Meanwhile, let us stick to our chapter plan and discuss the first image preprocessing step, blurring.

Blurring Images

Blurring an image is the first step to reducing the size of images without changing their appearance too much. Blurring can be thought of as a low-pass filtering operation, and is accomplished using a simple intuitive kernel matrix. An image can be thought of as having various "frequency components" along both of its axes' directions. Edges have high frequencies, whereas slowly changing intensity values have low frequencies. More specifically, a vertical edge creates high frequency components along the horizontal axis of the image and vice versa. Finely textured regions also have high frequencies (note that an area is called finely textured if pixel intensity values in it change considerably in short pixel distances). Smaller images cannot handle high frequencies nicely.

Think of it like this: suppose that you have a finely textured 640 x 480 image. You cannot maintain all of those short-interval high-magnitude changes in pixel intensity values in a 320 x 240 image, because it has only a quarter of the number of pixels. So whenever you want to reduce the size of an image, you should remove high-frequency components from it. In other words, blur it. Smooth out those high-magnitude short-interval changes. If you do not blur before you resize, you are likely to observe artifacts in the resized image. The reason for this is simple and depends on a fundamental theorem of signal theory, which states that sampling a signal causes infinite duplication in the frequency domain of that signal. So if the signal has many high-frequency components, the edges of the duplicated frequency domain representations will interfere with each other. Once that happens, the signal cannot be recovered faithfully. Here, the signal is our image and resizing is done by removing rows and columns, that is, down-sampling. This phenomenon is known as aliasing. If you want to know more about it, you should able to find a detailed explanation in any good digital signal processing resource. Because blurring removes high-frequency components from the image, it helps in avoiding aliasing.

Blurring is also an important postprocessing step when you want to increase the size of an image. If you want to double the size of an image, you add a blank row (column) for every row (column) and then blur the resulting image so that the blank rows (columns) assume appearance similar to their neighbors.

Blurring can be accomplished by replacing each pixel in the image with some sort of average of the pixels in a region around it. To do this efficiently, the region is kept rectangular and symmetric around the pixel, and the image is convolved with a "normalized" kernel (normalized because we want the average, not the sum). A very simple kernel is a box kernel:

```
1       1       1       1       1
1       1       1       1       1
1       1       1       1       1
1       1       1       1       1
1       1       1       1       1
```

This kernel deems every pixel equally important. A better kernel would be one that decreases the effect of a pixel as its distance from the central pixel increases. The Gaussian kernel does this, and is the most commonly used blurring kernel:

```
1       4       6       4       1
4       16      24      16      4
6       24      36      24      6
4       16      24      16      4
1       4       6       4       1
```

One "normalizes" the kernel by dividing by the sum of all elements, 25 for the box kernel and 256 for the Gaussian kernel. You can create different sizes of the Gaussian kernel by using the OpenCV function `getGaussianKernel()`. Check the documentation for this function to see the formula OpenCV uses to calculate the kernels. You can go ahead and plug these kernels into Listing 5-1 to blur some images (don't forget to divide the kernel by the sum of its elements). However, OpenCV also gives you the higher level function `GaussianBlur()` which just takes the kernel size and variance of the Gaussian function as input and does all the other work for us. We use this function in the code for Listing 5-2, which blurs an image with a Gaussian kernel of the size indicated by a slider. It should help you understand blurring practically. Figure 5-4 shows the code in action.

Listing 5-2. Program to interactively blur an image using a Gaussian kernel of varying size

```
// Program to interactively blur an image using a Gaussian kernel of varying size
// Author: Samarth Manoj Brahmbhatt, University of Pennyslvania

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

using namespace std;
using namespace cv;

Mat image, image_blurred;
int slider = 5;
float sigma = 0.3 * ((slider - 1) * 0.5 - 1) + 0.8;
```

```
void on_trackbar(int, void *) {
    int k_size = max(1, slider);
    k_size = k_size % 2 == 0 ? k_size + 1 : k_size;
    setTrackbarPos("Kernel Size", "Blurred image", k_size);
    sigma = 0.3 * ((k_size - 1) * 0.5 - 1) + 0.8;
    GaussianBlur(image, image_blurred, Size(k_size, k_size), sigma);
    imshow("Blurred image", image_blurred);
}

int main() {
    image = imread("baboon.jpg");

    namedWindow("Original image");
    namedWindow("Blurred image");

    imshow("Original image", image);
    sigma = 0.3 * ((slider - 1) * 0.5 - 1) + 0.8;
    GaussianBlur(image, image_blurred, Size(slider, slider), sigma);
    imshow("Blurred image", image_blurred);

    createTrackbar("Kernel Size", "Blurred image", &slider, 21, on_trackbar);

    while(char(waitKey(1) != 'q')) {}

    return 0;
}
```

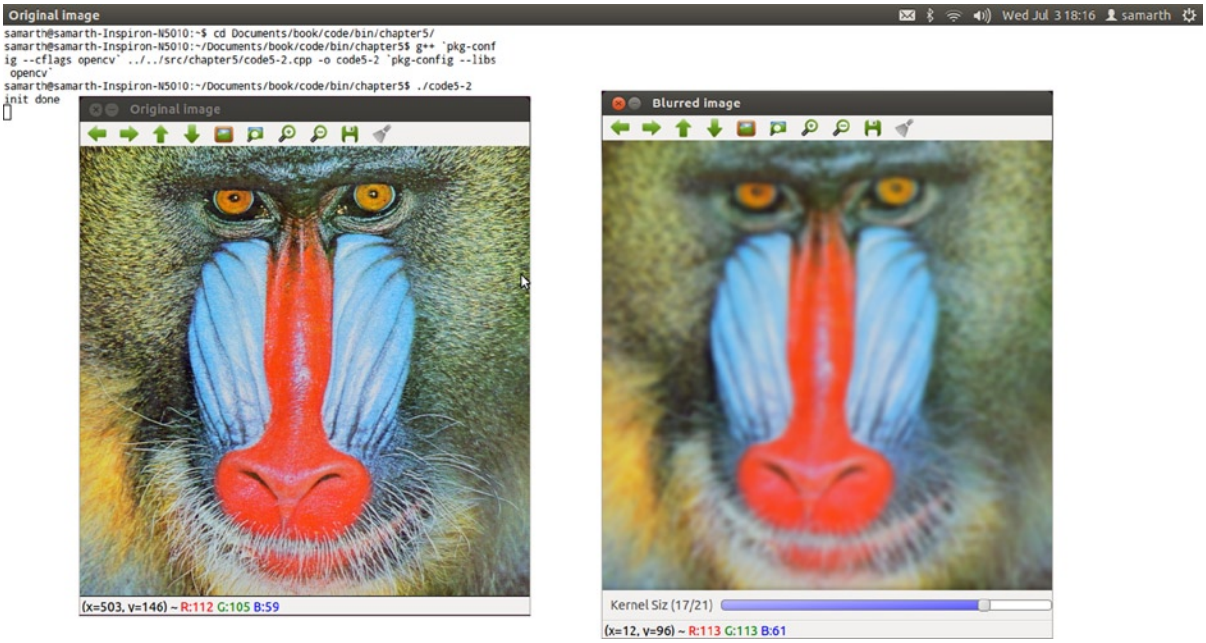


Figure 5-4. *Blurring an image with a Gaussian kernel*

Note the heuristic formula we use for calculating the variance based on size of the kernel and also the toolbar 'locking' mechanism that uses the `setTrackbarPos()` function to force the kernel size to be odd and greater than 0.

Resizing Images—Up and Down

Now that we know the importance of blurring our images while resizing them, we are ready to resize and verify if the theories that I have been expounding are correct.

You can do a naïve geometric resize (simply throwing out rows and columns) by using the `resize()` function as shown in Figure 5-5.

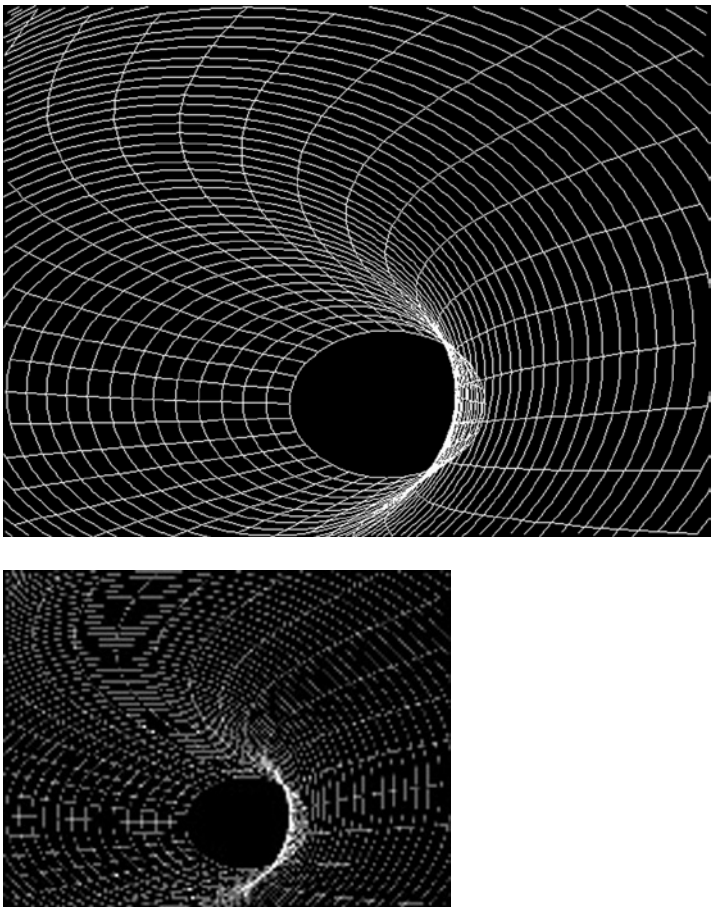


Figure 5-5. *Aliasing artifacts—simple geometric resizing of an image down by a factor of 4*

Observe the artifacts produced in the resized image due to aliasing. The `pyrDown()` function blurs the image by a Gaussian kernel and resizes it down by a factor of 2. The image in Figure 5-6 is a four-times downsized version of the original image, obtained by using `pyrDown()` twice (observe the absence of aliasing artifacts).

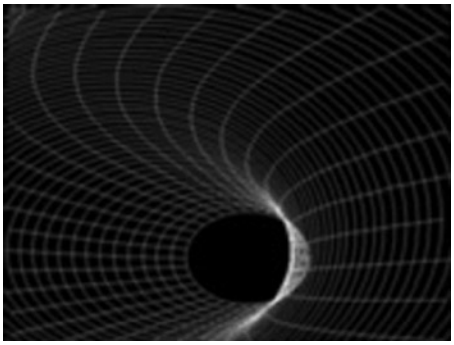


Figure 5-6. *Avoiding aliasing while resizing images by first blurring them with a Gaussian kernel*

The function `resize()` also works if you want to up-size an image and employs a bunch of interpolation techniques that you can choose from. If you want to blur the image after up-sizing, use the `pyrUp()` function an appropriate number of times (because it works by a factor of 2).

Eroding and Dilating Images

Erosion and dilation are two basic morphological operations on images. As the name suggests, morphological operations work on the form and structure of the image.

Erosion is done by sliding a rectangular kernel of all ones (a box kernel) over an image. Response is defined as the maximum of all element-wise multiplications between kernel elements and pixels falling under the kernel. Because all kernel elements are ones, applying this kernel means replacing each pixel value with the minimum value in a rectangular region surrounding the pixel. You can imagine that this will cause the black areas in the image to "encroach" into the white areas (because pixel value for white is higher than that for black).

Dilating the image is the same, the only difference being that the response if defined as the maximum of element-wise multiplications instead of minimum. This will cause the white regions to encroach into black regions.

The size of the kernel decides the amount of erosion or dilation. Listing 5-3 makes an app that switches between erosion and dilation, and allows you to choose the size of the kernel (also known as structuring element in the context of morphological operations).

Listing 5-3. Program to examine erosion and dilation of images

```
// Program to examine erosion and dilation of images
// Author: Samarth Manoj Brahmbhatt, University of Pennyslvania

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

using namespace std;
using namespace cv;

Mat image, image_processed;
int choice_slider = 0, size_slider = 5; // 0 - erode, 1 - dilate
```

```
void process() {
    Mat st_elem = getStructuringElement(MORPH_RECT, Size(size_slider, size_slider));

    if(choice_slider == 0) {
        erode(image, image_processed, st_elem);
    }
    else {
        dilate(image, image_processed, st_elem);
    }
    imshow("Processed image", image_processed);
}

void on_choice_slider(int, void *) {
    process();
}

void on_size_slider(int, void *) {
    int size = max(1, size_slider);
    size = size % 2 == 0 ? size + 1 : size;
    setTrackbarPos("Kernel Size", "Processed image", size);
    process();
}

int main() {
    image = imread("j.png");

    namedWindow("Original image");
    namedWindow("Processed image");

    imshow("Original image", image);
    Mat st_elem = getStructuringElement(MORPH_RECT, Size(size_slider, size_slider));
    erode(image, image_processed, st_elem);
    imshow("Processed image", image_processed);

    createTrackbar("Erode/Dilate", "Processed image", &choice_slider, 1, on_choice_slider);
    createTrackbar("Kernel Size", "Processed image", &size_slider, 21, on_size_slider);

    while(char(waitKey(1) != 'q')) {}

    return 0;
}
```

Figure 5-7 shows an image being eroded and dilated by different amounts specified by the user using sliders.

Figure 5-7. *Eroding and dilating images*

Notable in this code apart from the functions `erode()` and `dilate()` is the function `getStructuralElement()`, which returns the structural element (kernel matrix) of a specified shape and size. Predefined shapes include rectangle, ellipse, and cross. You can even create custom shapes. All of these shapes are returned embedded in a rectangular matrix of zeros (elements belonging to the shape are ones).

Detecting Edges and Corners Efficiently in Images

You saw earlier that vertical and horizontal edges can be detected quite easily using filters. If you build a proper kernel, you can detect edges of any orientation, as long as it is one fixed orientation. However, in practice one has to detect edges of all orientations in the same image. We will talk about some intelligent ways to do this. Corners can also be detected by employing kernels of the proper kind.

Edges

Edges are points in the image where the gradient of the image is quite high. By gradient we mean change in the value if the intensity of pixels. The gradient of the image is calculated by calculating gradient in X and Y directions and then combining them using Pythagoras' theorem. Although it is usually not required, you can calculate the angle of gradient by taking the arctangent of the ratio of gradients in Y and X directions, respectively.

X and Y direction gradients are calculated by convolving the image with the following kernels, respectively:

```
-3      0       3
-10     0       10
-3      0       3       (for X direction)
```

And

```
-3      -10     -3
0       0       0
3       10      3       (for Y direction)
```

Overall gradient, $G = \sqrt{(Gx^2 + Gy^2)}$

Angle of gradient, $\Phi = \arctan(Gy / Gx)$

The two kernels shown above are known as Scharr operators and OpenCV offers a function called Scharr() that applies a Scharr operator of a specified size and specified orientation (X or Y) to an image. So let us make our Scharr edges detector program as shown in Listing 5-4.

Listing 5-4. Program to detect edges in an image using the Scharr operator

```
// Program to detect edges in an image using the Scharr operator
// Author: Samarth Manoj Brahmbhatt, University of Pennyslvania

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

using namespace std;
using namespace cv;

int main() {
    Mat image = imread("lena.jpg"), image_blurred;

    // Blur image with a Gaussian kernel to remove edge noise
    GaussianBlur(image, image_blurred, Size(3, 3), 0, 0);

    // Convert to gray
    Mat image_gray;
    cvtColor(image_blurred, image_gray, CV_RGB2GRAY);

    // Gradients in X and Y directions
    Mat grad_x, grad_y;

    Scharr(image_gray, grad_x, CV_32F, 1, 0);
    Scharr(image_gray, grad_y, CV_32F, 0, 1);

    // Calculate overall gradient
    pow(grad_x, 2, grad_x);
    pow(grad_y, 2, grad_y);

    Mat grad = grad_x + grad_y;
    sqrt(grad, grad);

    // Display
    namedWindow("Original image");
    namedWindow("Scharr edges");
```

```
    // Convert to 8 bit depth for displaying
    Mat edges;
    grad.convertTo(edges, CV_8U);

    imshow("Original image", image);
    imshow("Scharr edges", edges);

    while(char(waitKey(1)) != 'q') {}

    return 0;
}
```

Figure 5-8 shows Scharr edges of the beautiful Lena image.

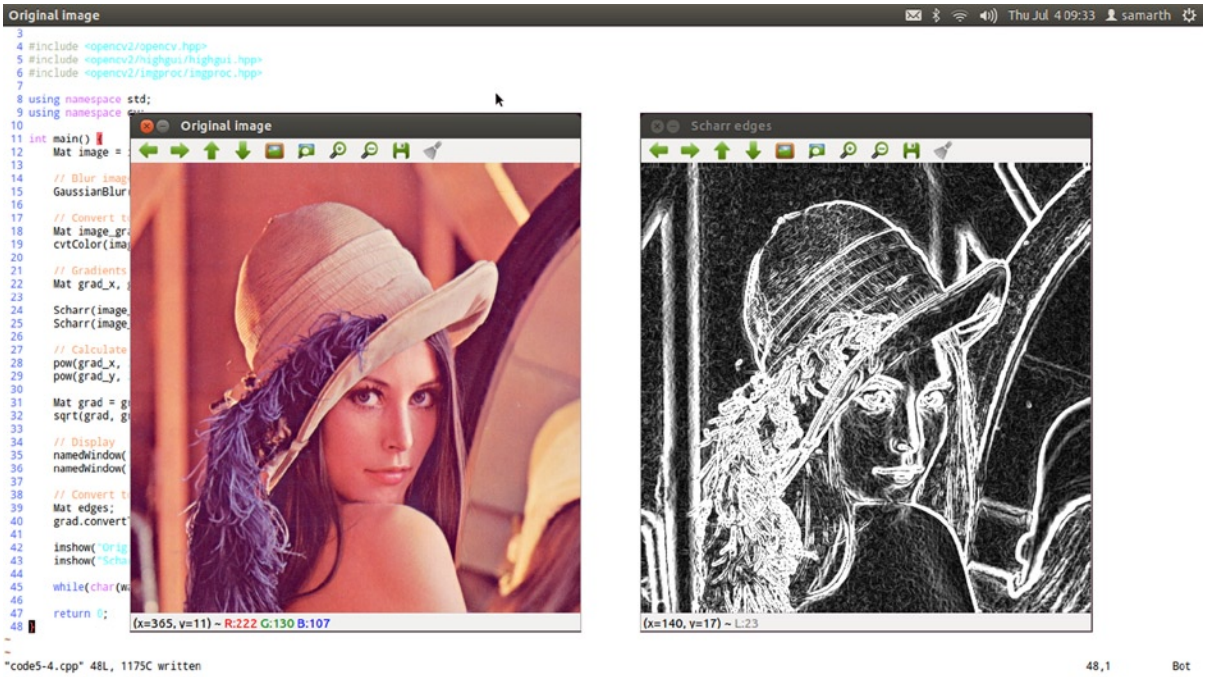


Figure 5-8. *Scharr edge detector*

You can see that the Scharr operator finds gradients as promised. However, there is a lot of noise in the edge image. Because the edge image has 8-bit depth, you can threshold it by a number between 0 and 255 to remove the noise. As always, you can make an app with a slider for the threshold. Code for this app is shown in Listing 5-5 and the thresholded Scharr outputs at different thresholds are shown in Figure 5-9.

Listing 5-5. Program to detect edges in an image using the thresholded Scharr operator

```
// Program to detect edges in an image using the thresholded Scharr operator
// Author: Samarth Manoj Brahmbhatt, University of Pennyslvania

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
```

```
using namespace std;
using namespace cv;

Mat edges, edges_thresholded;
int slider = 50;

void on_slider(int, void *) {
    if(!edges.empty()) {
        Mat edges_thresholded;
        threshold(edges, edges_thresholded, slider, 255, THRESH_TOZERO);
        imshow("Thresholded Scharr edges", edges_thresholded);
    }
}

int main() {
    //Mat image = imread("lena.jpg"), image_blurred;
    Mat image = imread("lena.jpg"), image_blurred;

    // Blur image with a Gaussian kernel to remove edge noise
    GaussianBlur(image, image_blurred, Size(3, 3), 0, 0);

    // Convert to gray
    Mat image_gray;
    cvtColor(image_blurred, image_gray, CV_BGR2GRAY);

    // Gradients in X and Y directions
    Mat grad_x, grad_y;

    Scharr(image_gray, grad_x, CV_32F, 1, 0);
    Scharr(image_gray, grad_y, CV_32F, 0, 1);

    // Calculate overall gradient
    pow(grad_x, 2, grad_x);
    pow(grad_y, 2, grad_y);

    Mat grad = grad_x + grad_y;
    sqrt(grad, grad);

    // Display
    namedWindow("Original image");
    namedWindow("Thresholded Scharr edges");

    // Convert to 8 bit depth for displaying
    grad.convertTo(edges, CV_8U);
    threshold(edges, edges_thresholded, slider, 255, THRESH_TOZERO);

    imshow("Original image", image);
    imshow("Thresholded Scharr edges", edges_thresholded);

    createTrackbar("Threshold", "Thresholded Scharr edges", &slider, 255, on_slider);

    while(char(waitKey(1)) != 'q') {}

    return 0;
}
```

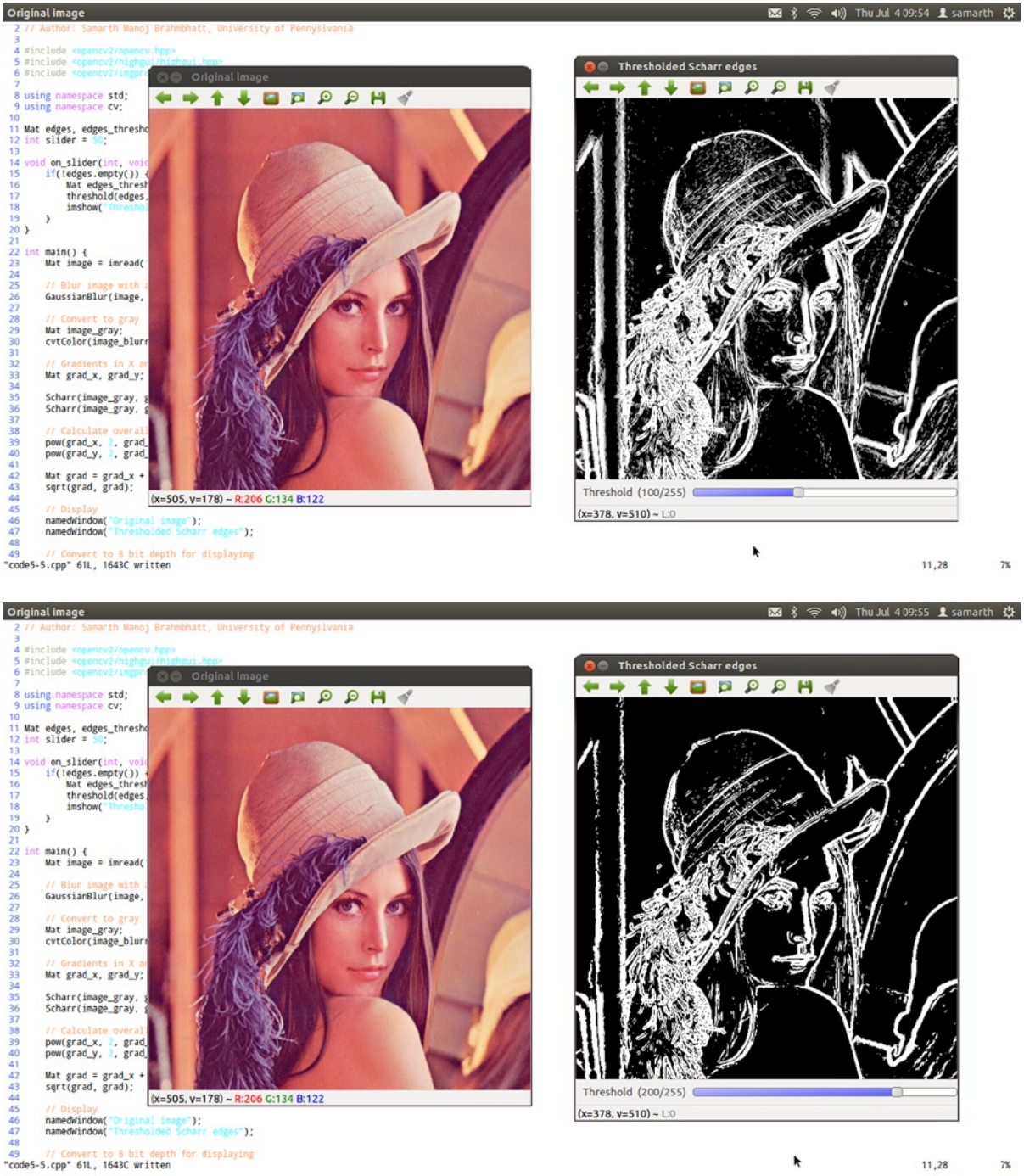


Figure 5-9. *Scharr edge detector with thresholds of 100 (top) and 200 (bottom)*

Canny Edges

The Canny algorithm uses some postprocessing to clean the edge output and gives thin, sharp edges. The steps involved in computing Canny edges are:

- Remove edge noise by convolving the image with a normalized Gaussian kernel of size 5
- Compute X and Y gradients by using two different kernels:

```
-1       0       1
-2       0       2
-1       0       1      for X direction and

-1      -2      -1
 0       0       0
 1       2       1      for Y direction
```

- Find overall gradient strength by Pythagoras' theorem and gradient angle by arctangent as discussed previously. Angle is rounded off to four options: 0, 45, 90, and 135 degrees
- Nonmaximum suppression: A pixel is considered to be on an edge only if its gradient magnitude is larger than that at its neighboring pixels in the gradient direction. This gives sharp and thin edges
- Hysteresis thresholding: This process uses two thresholds. A pixel is accepted as an edge if its gradient magnitude is higher than the upper threshold and rejected if its gradient magnitude is lower than the lower threshold. If the gradient magnitude is between the two thresholds, it will be accepted as an edge only if it is connected to a pixel that is an edge

You can run the OpenCV Canny edge demo on the Lena picture to see the difference between Canny and Scharr edges. Figure 5-10 shows Canny edges extracted from the same Lena photo.

Figure 5-10. *Canny edge detector*

Corners

The OpenCV function `goodFeaturesToTrack()` implements a robust corner detector. It uses the interest-point detector algorithm proposed by Shi and Tomasi. More information about the internal workings of this function can be found in its documentation page at http://docs.opencv.org/modules/imgproc/doc/feature_detection.html?highlight=goodfeaturestotrack#goodfeaturestotrack.

The function takes the following inputs:

- Grayscale image
- A STL vector of Point2d's to store the corner locations in (more on STL vectors later)
- Maximum number of corners to return. If the algorithm detects more corners than this number, only the strongest appropriate number of corners are returned
- Quality level: The minimum accepted quality of the corners. The quality of a corner is defined as the minimum eigenvalue of the matrix of image intensity gradients at a pixel or (if the Harris corner detector is used) the image's response to the Harris function at that pixel. For more details read the documentation for `cornerHarris()` and `cornerMinEigenVal()`
- Minimum Euclidean distance between two returned corner locations
- A flag indicating whether to use the Harris corner detector or minimum eigenvalue corner detector (default is minimum eigenvalue)
- If the Harris corner detector is used, a parameter that tunes the Harris detector (for usage of this parameter, see documentation for `cornerHarris()`)

STL is an abbreviation for Standard Template Library, a library that provides highly useful data structures that can be template into any data type. One of the data structures is a `vector`, and we will use a `vector` of OpenCV's `Point2d`'s to store the location of the corners. As you might recall, `Point2d` is OpenCV's way of storing a pair of integer values (usually location of a point in an image). Listing 5-6 shows code that extracts corners from an image using the `goodFeaturesToTrack()`function, allowing the user to decide the maximum number of corners.

Listing 5-6. Program to detect corners in an image

```
// Program to detect corners in an image
// Author: Samarth Manoj Brahmbhatt, University of Pennyslvania

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <stdlib.h>

using namespace std;
using namespace cv;

Mat image, image_gray;
int max_corners = 20;
```

```
void on_slider(int, void *) {
    if(image_gray.empty()) return;

    max_corners = max(1, max_corners);
    setTrackbarPos("Max no. of corners", "Corners", max_corners);

    float quality = 0.01;
    int min_distance = 10;

    vector<Point2d> corners;

    goodFeaturesToTrack(image_gray, corners, max_corners, quality, min_distance);

    // Draw the corners as little circles
    Mat image_corners = image.clone();
    for(int i = 0; i < corners.size(); i++) {
        circle(image_corners, corners[i], 4, CV_RGB(255, 0, 0), -1);
    }

    imshow("Corners", image_corners);
}

int main() {
    image = imread("building.jpg");
    cvtColor(image, image_gray, CV_RGB2GRAY);

    namedWindow("Corners");

    on_slider(0, 0);

    createTrackbar("Max. no. of corners", "Corners", &max_corners, 250, on_slider);

    while(char(waitKey(1)) != 'q') {}

    return 0;
}
```

In this app, we change the maximum number of returnable corners by a slider. Observe how we use the `circle()` function to draw little red filled circles at locations of the corners. Output produced by the app is shown in Figure 5-11.

Figure 5-11. *Corners at different values of* `max_corners`

Object Detector App

Our first object detector program will use just color information. In fact, it is more of a color bound-checking program than an object detector in the strict sense because there is no machine learning involved. The idea is to take up the problem we discussed at the beginning of this chapter—finding out the rough position of a red colored ball and controlling a simple wheeled robot to intercept it. The most naïve way to detect the red ball is to see if the RGB values of pixels in the image correspond to the red ball, and that is what we will start with. We will also try to improve this app as we keep learning new techniques over the next chapter. If you solved the exercise question to the last chapter you already know which OpenCV function to use—`inRange()`. Here goes Listing 5-7, our first attempt at object detection!

Listing 5-7. Simple color based object detector

```
// Program to display a video from attached default camera device and detect colored blobs using simple
// R G and B thresholding
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

using namespace cv;
using namespace std;

Mat frame, frame_thresholded;
int rgb_slider = 0, low_slider = 30, high_slider = 100;
int low_r = 30, low_g = 30, low_b = 30, high_r = 100, high_g = 100, high_b = 100;

void on_rgb_trackbar(int, void *) {
    switch(rgb_slider) {
        case 0:
            setTrackbarPos("Low threshold", "Segmentation", low_r);
            setTrackbarPos("High threshold", "Segmentation", high_r);
            break;
        case 1:
            setTrackbarPos("Low threshold", "Segmentation", low_g);
            setTrackbarPos("High threshold", "Segmentation", high_g);
            break;
        case 2:
            setTrackbarPos("Low threshold", "Segmentation", low_b);
            setTrackbarPos("High threshold", "Segmentation", high_b);
            break;
    }
}

void on_low_thresh_trackbar(int, void *) {
    switch(rgb_slider) {
        case 0:
            low_r = min(high_slider - 1, low_slider);
            setTrackbarPos("Low threshold", "Segmentation", low_r);
            break;
```

```
        case 1:
            low_g = min(high_slider - 1, low_slider);
            setTrackbarPos("Low threshold", "Segmentation", low_g);
            break;
        case 2:
            low_b = min(high_slider - 1, low_slider);
            setTrackbarPos("Low threshold", "Segmentation", low_b);
            break;
    }
}

void on_high_thresh_trackbar(int, void *) {
    switch(rgb_slider) {
        case 0:
            high_r = max(low_slider + 1, high_slider);
            setTrackbarPos("High threshold", "Segmentation", high_r);
            break;
        case 1:
            high_g = max(low_slider + 1, high_slider);
            setTrackbarPos("High threshold", "Segmentation", high_g);
            break;
        case 2:
            high_b = max(low_slider + 1, high_slider);
            setTrackbarPos("High threshold", "Segmentation", high_b);
            break;
    }
}

int main()
{
        // Create a VideoCapture object to read from video file
        // 0 is the ID of the built-in laptop camera, change if you want to use other camera
        VideoCapture cap(0);

        //check if the file was opened properly
        if(!cap.isOpened())
        {
                cout << "Capture could not be opened succesfully" << endl;
                return -1;
        }

        namedWindow("Video");
        namedWindow("Segmentation");

        createTrackbar("0. R\n1. G\n2.B", "Segmentation", &rgb_slider, 2, on_rgb_trackbar);
        createTrackbar("Low threshold", "Segmentation", &low_slider, 255, on_low_thresh_trackbar);
        createTrackbar("High threshold", "Segmentation", &high_slider, 255, on_high_thresh_trackbar);
```

```
        while(char(waitKey(1)) != 'q' && cap.isOpened())
        {
                cap >> frame;
                // Check if the video is over
                if(frame.empty())
                {
                        cout << "Video over" << endl;
                        break;
                }

                inRange(frame, Scalar(low_b, low_g, low_r), Scalar(high_b, high_g, high_r),
frame_thresholded);

                imshow("Video", frame);
                imshow("Segmentation", frame_thresholded);
        }

        return 0;
}
```

Figure 5-12 shows the program detecting an orange-colored object.

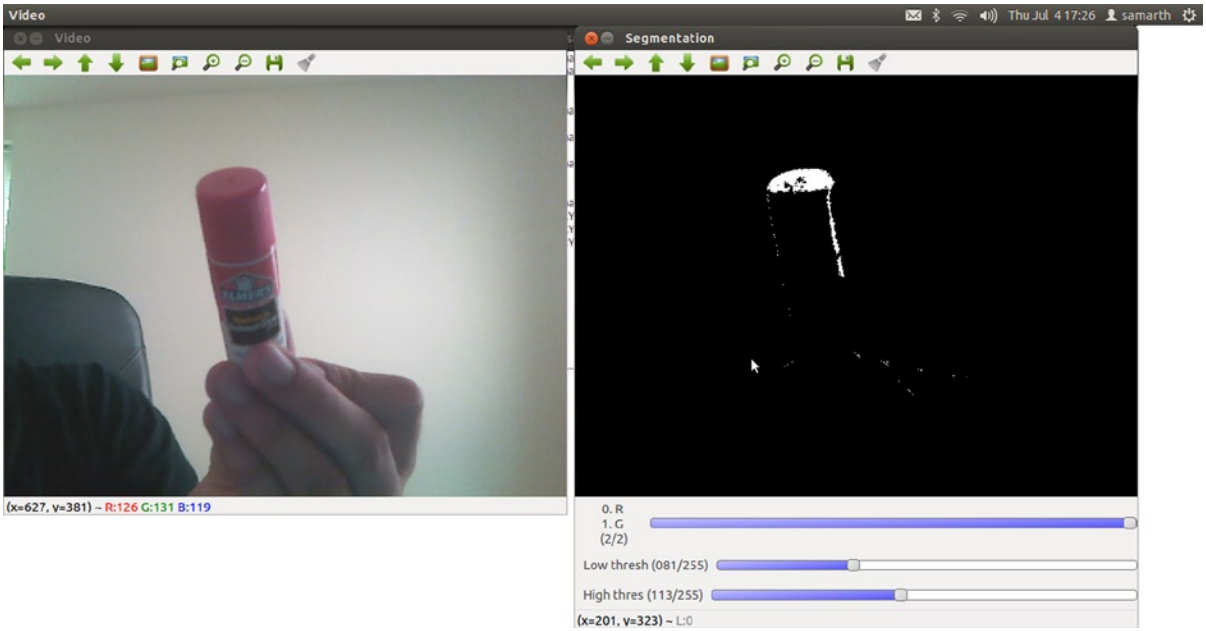


Figure 5-12. *Color-based object detector*

Observe how we have used locking mechanisms to ensure that the lower threshold is never higher than the higher threshold and vice versa. To use this app, first hold the object in front of the camera. Then, hover your mouse over the object in the window named "Video" and observe the R, G and B values. Finally, adjust ranges appropriately in the "Segmentation" window. This app has many shortcomings:

- It cannot detect objects of multiple colors
- It is highly dependent on illumination
- It gives false positives on other objects of the same color

But it is a good start!

Morphological Opening and Closing of Images to Remove Noise

Recall the definitions of morphological erosion and dilation. Opening is obtained by eroding an image followed by dilating it. It will have an effect of removing small white regions in the image. Closing is obtained by dilating an image followed by eroding it; this will have the opposite effect. Both these operations are used frequently to remove noise from an image. Opening removes small white pixels while closing removes small black "holes." Our object detector app is an ideal platform to examine this, because we have some noise in the form of white and black dots in the "Segmentation" window, as you can see in Figure 5-12.

The OpenCV function `morphologyEX()`can be used to perform advanced morphological operations such as opening and closing on images. So we can open and close the output of the `inRange()` function to remove black and white dots by adding the three lines in the `while` loop in the `main()` function of our previous object detector code. The new `main()` function is shown in Listing 5-8.

Listing 5-8. Adding the opening and closing steps to the object detector code

```
int main()
{
        // Create a VideoCapture object to read from video file
        // 0 is the ID of the built-in laptop camera, change if you want to use other camera
        VideoCapture cap(0);

        //check if the file was opened properly
        if(!cap.isOpened())
        {
                cout << "Capture could not be opened succesfully" << endl;
                return -1;
        }

        namedWindow("Video");
        namedWindow("Segmentation");

        createTrackbar("0. R\n1. G\n2.B", "Segmentation", &rgb_slider, 2, on_rgb_trackbar);
        createTrackbar("Low threshold", "Segmentation", &low_slider, 255, on_low_thresh_trackbar);
        createTrackbar("High threshold", "Segmentation", &high_slider, 255, on_high_thresh_trackbar);
```

```
        while(char(waitKey(1)) != 'q' && cap.isOpened())
        {
                cap >> frame;
                // Check if the video is over
                if(frame.empty())
                {
                        cout << "Video over" << endl;
                        break;
                }

                inRange(frame, Scalar(low_b, low_g, low_r), Scalar(high_b, high_g, high_r),
frame_thresholded);
                Mat str_el = getStructuringElement(MORPH_RECT, Size(3, 3));
                morphologyEx(frame_thresholded, frame_thresholded, MORPH_OPEN, str_el);
                morphologyEx(frame_thresholded, frame_thresholded, MORPH_CLOSE, str_el);

                imshow("Video", frame);
                imshow("Segmentation", frame_thresholded);
        }

        return 0;
}
```

Figure 5-13 shows that opening and closing the color-bound checker output does indeed remove speckles and holes.

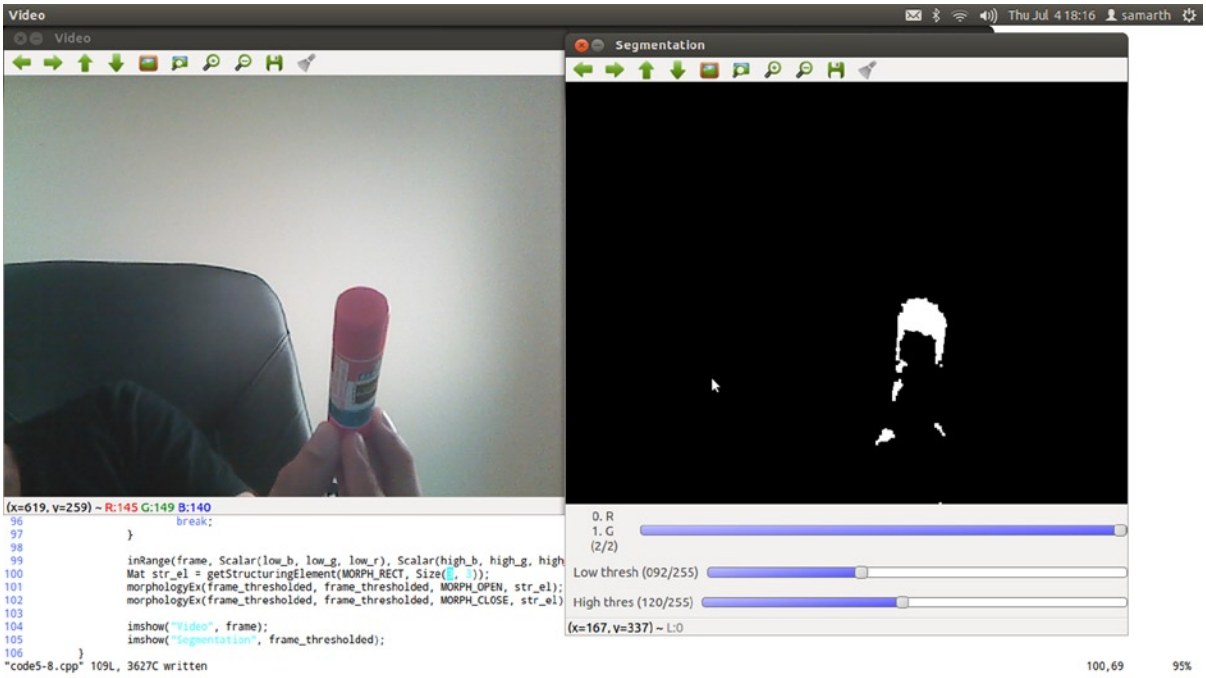


Figure 5-13. *Removing small patches of noisy pixels by opening and closing*

Summary

Image filtering is the basis of all computer vision operations. In the abstract sense, every algorithm applied to an image can be considered as a filtering operation, because you are trying to extract some relevant information out of the large gamut of different kinds of information contained in the image. In this chapter you learned a lot of filter-based image operations that will help you as starting steps in many complicated computer vision projects. Remember, computer vision is complete only when you extract decisive information from the image. You also developed the roots of the simple color based object detector app, which we will continue with in the next chapter.

Whereas this chapter dealt with a lot of low-level algorithms, the next chapter will focus more on algorithms that deal with the form and structure of regions in images.

CHAPTER 6

Shapes in Images

Shapes are one of the first details we notice about objects when we see them. This chapter will be devoted to endowing the computer with that capability. Recognizing shapes in images can often be an important step in making decisions. Shapes are defined by the outlines of images. It is therefore logical that the shape recognition is step is usually applied after detecting edges or contours.

Therefore, we will discuss extracting contours from images first. Then we shall begin our discussion of shapes, which will include:

- The Hough transform, which will enable us to detect regular shapes like lines and circles in images
- Random Sample Consensus (RANSAC), a widely used framework to identify data-points that fit a particular model. We will write code for the algorithm and employ it to detect ellipses in images
- Calculation of bounding boxes, bounding ellipses, and convex hulls around objects
- Matching shapes

Contours

There is a marked difference between contours and edges. Edges are local maxima of intensity gradients in an image (remember Scharr edges in the previous chapter?). As we also saw, these gradient maxima are not all on the outlines of objects and they are very noisy. Canny edges are a bit different, and they are a lot like contours, since they pass through a lot of post-processing steps after gradient maxima extraction. Contours, by contrast, are a set of points connected to each other, most likely to be located on the outline of objects.

OpenCV's Contour extraction works on a binary image (like the output of Canny edge detection or a threshold applied on Scharr edges or a black-and-white image) and extracts a hierarchy of connected points on edges. The hierarchy is organized such that contours higher up the tree are more likely to be outlines of objects, whereas contours lower down are likely to be noisy edges and outlines of "holes" and noisy patches.

The function that implements these features is called `findContours()` and it uses the algorithm described in the paper "Topological Structural Analysis of Digitized Binary Images by Border Following" by S. Suzuki and K. Abe (published in the 1985 edition of CVGIP) for extracting contours and arranging them in a hierarchy. The exact set of rules for deciding the hierarchies are described in the paper but, in a nutshell, a contour is considered to be "parent" to another contour if it surrounds the contour.

To show practically what we mean by hierarchy, we will code a program, shown in Listing 6-1, that uses our favorite tool, the slider, to select the number of levels of this hierarchy to display. Note that the function accepts a binary image only as input. Some means of getting a binary image from an ordinary image are:

- Threshold using `threshold()` or `adaptiveThreshold()`
- Check for bounds on pixel values using `inRange()` as we did for our color-based object detector

- Canny edges
- Thresholded Scharr edges

Listing 6-1. Program to illustrate hierarchical contour extraction

```
// Program to illustrate hierarchical contour extraction
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

using namespace std;
using namespace cv;

Mat img;
vector<vector<Point> > contours;
vector<Vec4i> heirarchy;

int levels = 0;

void on_trackbar(int, void *) {
    if(contours.empty()) return;

    Mat img_show = img.clone();

    // Draw contours of the level indicated by slider
    drawContours(img_show, contours, -1, Scalar(0, 0, 255), 3, 8, heirarchy, levels);
    imshow("Contours", img_show);
}

int main() {
    img = imread("circles.jpg");

    Mat img_b;
    cvtColor(img, img_b, CV_RGB2GRAY);

    Mat edges;
    Canny(img_b, edges, 50, 100);

    // Extract contours and heirarchy
    findContours(edges, contours, heirarchy, CV_RETR_TREE, CV_CHAIN_APPROX_NONE);

    namedWindow("Contours");
    createTrackbar("levels", "Contours", &levels, 15, on_trackbar);

    // Initialize by drawing the top level contours (as 'levels' is initialized to 0)
    on_trackbar(0, 0);

    while(char(waitKey(1)) != 'q') {}

    return 0;
}
```

Note how each contour is an STL vector of points. Hence, the data structure that holds the contours is a vector of vectors of points. The hierarchy is a vector of four-integer vectors. For each contour, its hierarchical position is described by four integers: they are 0-based indexes into the vector of contours indicating the position of next (at the same level), previous (at the same level), parent, and first child contours. If any of these are nonexistent (for example, if a contour does not have a parent contour), the corresponding integer will be negative. Note also how the function `drawContours()` modifies the input image by drawing contours on it according to the hierarchy and maximum allowable level of that hierarchy to draw (consult the docs for the function)!

Figure 6-1 shows the various levels of contours in a convenient picture.

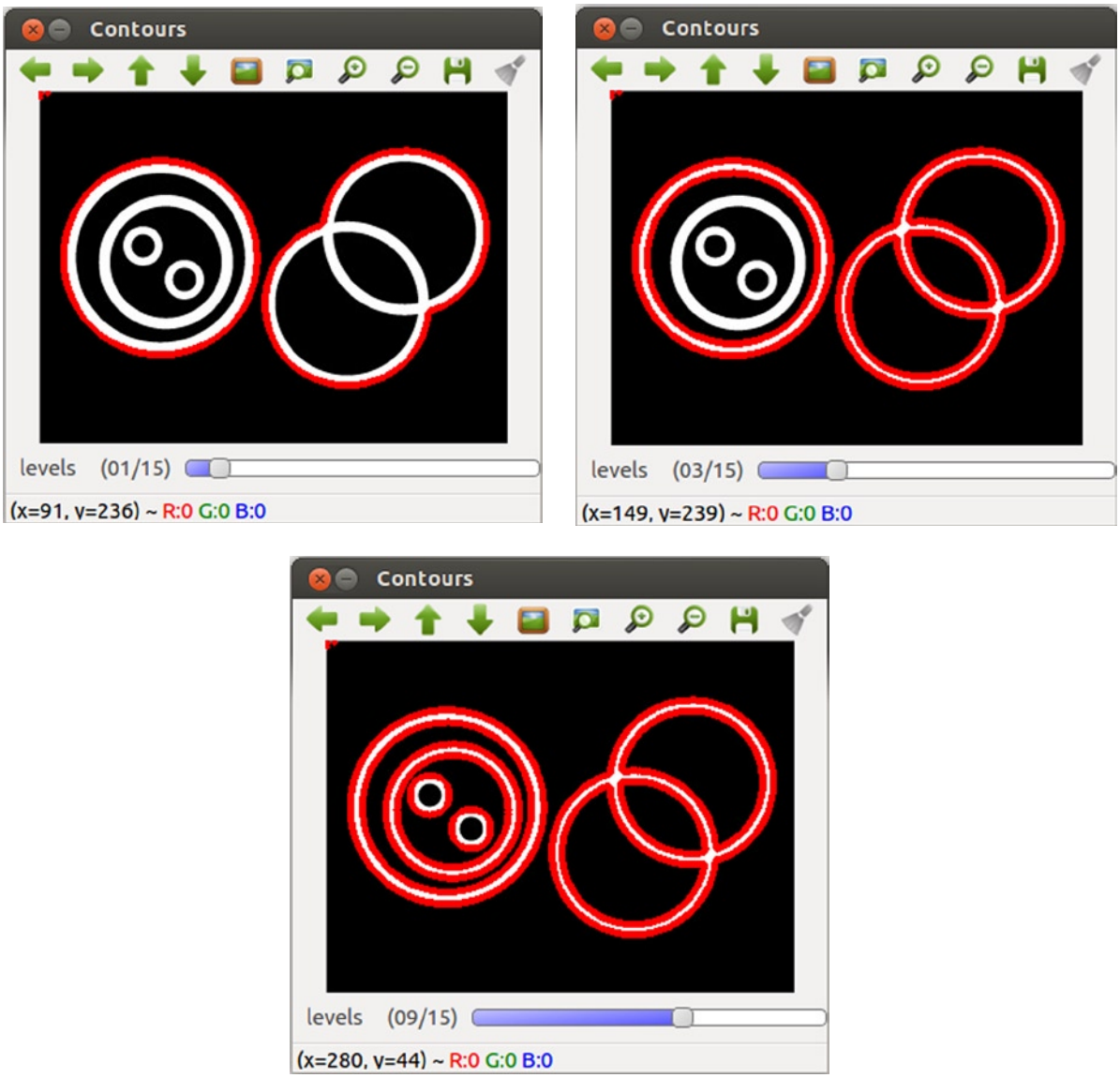


Figure 6-1. *Various levels of the hierarchy of contours*

A function that is used often in conjunction with `findContours()` is `approxPolyDP()`. `approxPolyDP()` approximates a curve or a polygon with another curve with fewer vertices so that the distance between the two curves is less or equal to the specified precision. You also have the option of making this approximated curve closed (i.e., the starting and ending points are the same).

Point Polygon Test

We take a brief detour to describe an interesting feature: the point-polygon test. As you might have guessed, the function `pointPolygonTest()` determines whether a point is inside a polygon. It also returns the signed Euclidean distance to the point from the closest point on the contour if you set the `measureDist` flag on. The distance is positive if the point is inside the curve, negative if outside, and zero if the point is on the contour. If the flag is turned off, the signed distance is replaced by +1, –1, and 0, accordingly.

Let's make an app to demonstrate our newfound knowledge of the point-polygon test and closed-curve approximation—an app that finds the smallest closed contour in the image enclosing a point clicked by the user. It will also illustrate navigation through the contour hierarchy. The code is shown in Listing 6-2.

Listing 6-2. Program to find the smallest contour that surrounds the clicked point

```
// Program to find the smallest contour that surrounds the clicked point
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

using namespace std;
using namespace cv;

Mat img_all_contours;
vector<vector<Point> > closed_contours;
vector<Vec4i> heirarchy;

// Function to approximate contours by closed contours
vector<vector<Point> > make_contours_closed(vector<vector<Point> > contours) {
    vector<vector<Point> > closed_contours;
    closed_contours.resize(contours.size());
    for(int i = 0; i < contours.size(); i++)
        approxPolyDP(contours[i], closed_contours[i], 0.1, true);

    return closed_contours;
}

// Function to return the index of smallest contour in 'closed_contours' surrounding the clicked point
int smallest_contour(Point p, vector<vector<Point> > contours, vector<Vec4i> heirarchy) {
    int idx = 0, prev_idx = -1;
    while(idx >= 0) {
        vector<Point> c = contours[idx];
        // Point-polgon test
        double d = pointPolygonTest(c, p, false);
```

```
        // If point is inside the contour, check its children for an even smaller contour...
        if(d > 0) {
            prev_idx = idx;
            idx = heirarchy[idx][2];
        }
        // ...else, check the next contour on the same level
        else idx = heirarchy[idx][0];
    }

    return prev_idx;
}

void on_mouse(int event, int x, int y, int, void *) {
    if(event != EVENT_LBUTTONDOWN) return;

    // Clicked point
    Point p(x, y);

    // Find index of smallest enclosing contour
    int contour_show_idx = smallest_contour(p, closed_contours, heirarchy);
    // If no such contour, user clicked outside all contours, hence clear image
    if(contour_show_idx < 0) {
        imshow("Contours", img_all_contours);
        return;
    }

    // Draw the smallest contour using a thick red line
    vector<vector<Point> > contour_show;
    contour_show.push_back(closed_contours[contour_show_idx]);

    if(!contour_show.empty()) {
        Mat img_show = img_all_contours.clone();
        drawContours(img_show, contour_show, -1, Scalar(0, 0, 255), 3);
        imshow("Contours", img_show);
    }
}

int main() {
    Mat img = imread("circles.jpg");
    img_all_contours = img.clone();

    Mat img_b;
    cvtColor(img, img_b, CV_RGB2GRAY);

    Mat edges;
    Canny(img_b, edges, 50, 100);

    // Extract contours and heirarchy
    vector<vector<Point> > contours;
    findContours(edges, contours, heirarchy, CV_RETR_TREE, CV_CHAIN_APPROX_NONE);
```

```
    // Make contours closed so point-polygon test is valid
    closed_contours = make_contours_closed(contours);

    // Draw all contours usign a thin green line
    drawContours(img_all_contours, closed_contours, -1, Scalar(0, 255, 0));

    imshow("Contours", img_all_contours);

    // Mouse callback
    setMouseCallback("Contours", on_mouse);

    while(char(waitKey(1)) != 'q') {}

    return 0;
}
```

Comments should help you understand the logic I follow in the code. A bit of detail about navigating the contour hierarchy is needed. If `idx` is the index of a contour in the vector of vector of points and `hierarchy` is the hierarchy:

- `hierarchy[idx][0]` will return the index of the next contour at the same level of hierarchy
- `hierarchy[idx[1]` will return the index of the previous contour at the same level
- `hierarchy[idx][2]` will return the index of the first child contour
- `hierarchy[idx][3]` will return the index of the parent contour

If any of these contours do not exist, the index returned will be negative.
Some screenshots of the app in action are shown in Figure 6-2.

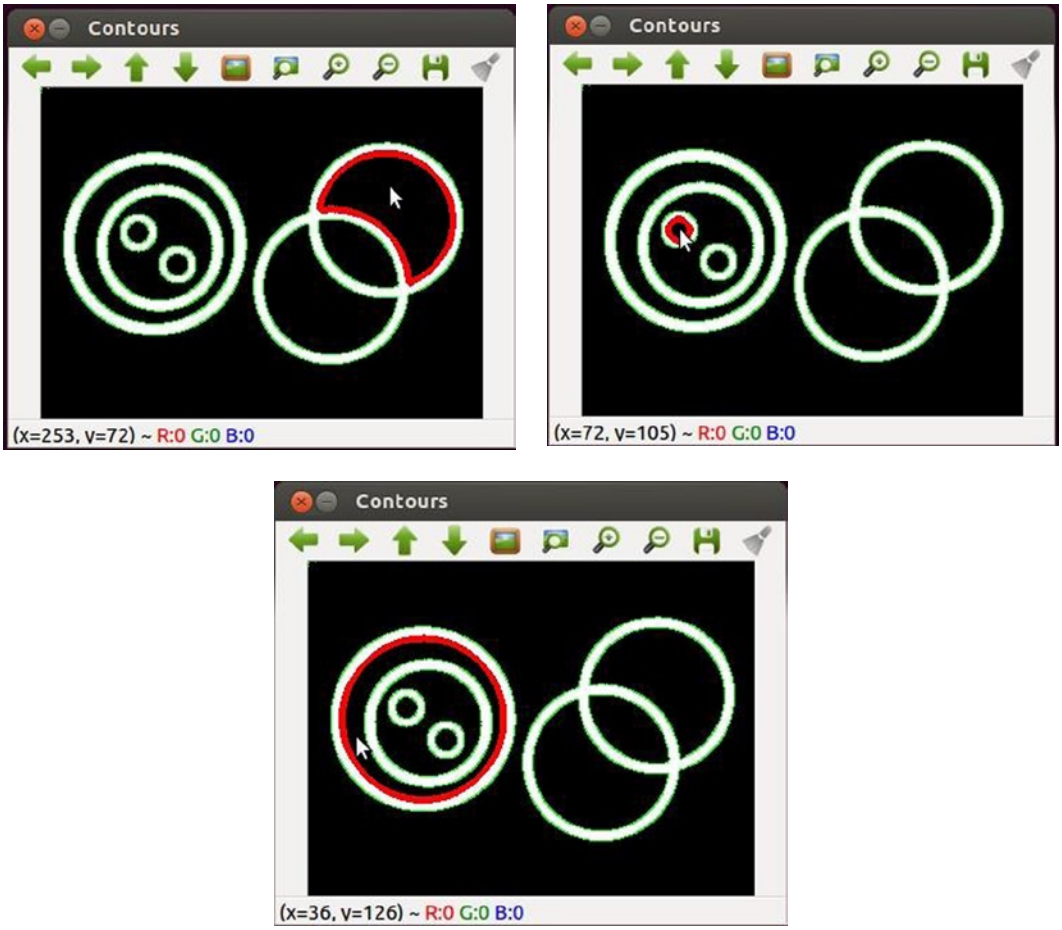


Figure 6-2. *Smallest enclosing contour app*

OpenCV also offers some other functions that can help you filter contours in noisy images by checking some of their properties. These are listed in Table 6-1.

Table 6-1. *Contour post-processing functions in OpenCV*

Function	Description
ArcLength()	Find length of a contour
ContourArea()	Find area and orientation of a contour
BoundingRect()	Compute the upright bounding rectangle of a contour
ConvexHull()	Compute a convex hull around a contour
IsContourConvex()	Tests a contour for convexity
MinAreaRect()	Computes a rotated rectangle of the minimum area around a contour
MinEnclosingCircle()	Finds a circle of minimum area enclosing a contour
FitLine()	Fits a line (in the least-squares) sense to a contour

Hough Transform

The Hough transform transforms a contour from X-Y space to a parameter space. It then uses certain parametric properties of target curves (like lines or circles) to identify the points that fit the target curve in parameter space. For example, let's take the problem of detecting lines in the output of edge detection applied to an image.

Detecting Lines with Hough Transform

A point in a 2D image can be represented in two ways:

- (`X, Y`) coordinates
- (`r, theta`) coordinates: `r` is the distance from (`0, 0`) and `theta` is angle from a reference line, usually the *x*-axis. This representation is shown in Figure 6-3.

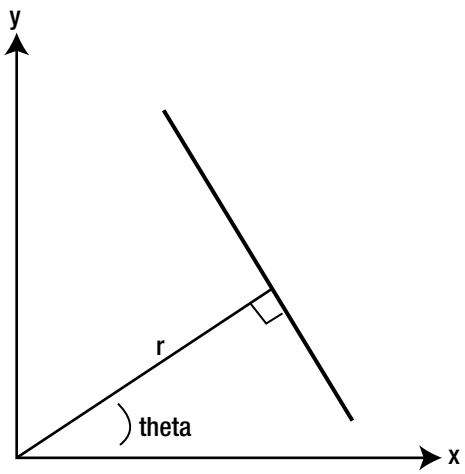


Figure 6-3. *The (`r, theta`) coordinate representation*

The relation between these coordinates is:

```
x*cos(theta) + y*sin(theta) = 2*r
```

As you can see, the plot of a point (`x, y`) in the (`r, theta`) parameter space is a sinusoid. Thus, points that are collinear in the Cartesian space will correspond to different sinusoids in the Hough space, which will intersect at a common (`r, theta`) point. This (r, theta) point represents a line in the Cartesian space passing through all these points. To give you some examples, Figure 6-4 shows the Hough space representations of a point, a pair of points and five points, respectively. The Matlab code shown in Listing 6-3 was used to generate the figures.

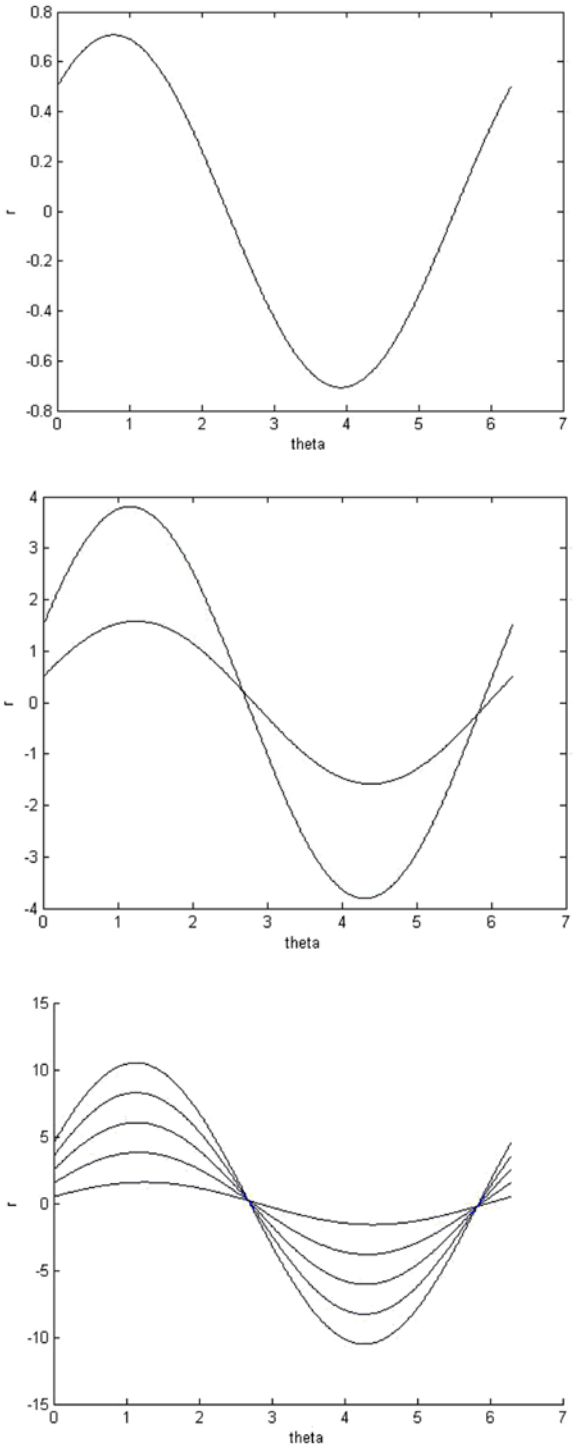


Figure 6-4. *Hough transform of a point, a pair of points, and 5 points (top to bottom, clockwise), respectively*

Listing 6-3. Matlab code to understand the Hough transform

```
%% one point
theta = linspace(0, 2*pi, 500);

x = 1;
y = 1;

r = 0.5 * (x * cos(theta) + y * sin(theta));
figure; plot(theta, r);
xlabel('theta'); ylabel('r');

%% two points
theta = linspace(0, 2*pi, 500);

x = [1 3];
y = 2*x + 1;

r1 = 0.5 * (x(1) * cos(theta) + y(1) * sin(theta));
r2 = 0.5 * (x(2) * cos(theta) + y(2) * sin(theta));

figure; plot(theta, r1, theta, r2);
xlabel('theta'); ylabel('r');

%% five collinear points
theta = linspace(0, 2*pi, 500);

x = [1 3 5 7 9];
y = 2*x + 1;

figure; hold on;
r = zeros(numel(x), numel(theta));
for i = 1 : size(r, 1)
    r(i, :) = 0.5 * (x(i) * cos(theta) + y(i) * sin(theta));
    plot(theta, r(i, :));
end

xlabel('theta'); ylabel('r');
```

We can then detect lines by the following strategy:

- Define a 2-D matrix of the discretized Hough space, for example with `r` values along rows and `theta` values along columns
- For every `(x, y)` point in the edge image, find the list of possible `(r, theta)` values using the equation, and increment the corresponding entries in the Hough space matrix (this matrix is also called the accumulator)
- When you do this for all the edge points, certain `(r, theta)` values in the accumulator will have high values. These are the lines, because every `(r, theta)` point represents a unique line

The OpenCV function `HoughLines()` implements this strategy and takes the following inputs:

- Binary edge image (for example, output of the Canny edge detector)
- `r` and `theta` resolutions
- Accumulator threshold for considering a (`r, theta`) point as a line

Detecting Circles with Hough Transform

A circle can be represented by three parameters: two for the center and one for the radius. Because the center of the circle lies along the normal to every point on the circle, we can use the following strategy:

- Use a 2D accumulator (that maps to points on the image) to accumulate votes for the center. For each edge point, you vote by incrementing the accumulator positions corresponding to pixels along the normal. As you do this for all the pixels on the circle, because the center lies on all the normals, the votes for the center will start increasing. You can find the centers of circles by thresholding this accumulator
- In the next stage, you make a 1D radius histogram per candidate center for estimating radius. Every edge point belonging to the circle around the candidate center will vote for almost the same radius (as they will be at almost the same distance from the center) while other edge points (possibly belonging to circles around other candidate centers) will vote for other spurious radii

The OpenCV function that implements Hough circle detection is called `HoughCircles()`. It takes the following inputs:

- Grayscale image (on which it applies Canny edge detection)
- Inverse ratio of the accumulator resolution to image resolution (for detecting centers of circles). For example, if it is set to 2, the accumulator is half the size of the image
- Minimum distance between centers of detected circles. This parameter can be used to reject spurious centers
- Higher threshold of the Canny edge detector for pre-processing the input image (the lower threshold is set to half of the higher threshold)
- The accumulator threshold
- Minimum and maximum circle radius, can also be used to filter out noisy small and false large circles

Figure 6-5 shows line and circle detection using the Hough transform in action with a slider for the accumulator threshold, and a switch for the choice of detecting lines or circles. The code follows in Listing 6-4.

Figure 6-5. *Circle and line detection at different accumulator thresholds using the Hough transform*

Listing 6-4. Program to illustrate line and circle detection using Hough transform

```
// Program to illustrate line and circle detection using Hough transform
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

using namespace std;
using namespace cv;
```

```
Mat img;
int shape = 0;                  //0 -> lines, 1 -> circles
int thresh = 100;               // Accumulator threshold

void on_trackbar(int, void *) { // Circles
    if(shape == 1) {
        Mat img_gray;
        cvtColor(img, img_gray, CV_RGB2GRAY);
        // Find circles
        vector<Vec3f> circles;
        HoughCircles(img_gray, circles, CV_HOUGH_GRADIENT, 1, 10, 100, thresh, 5);
        // Draw circles
        Mat img_show = img.clone();
        for(int i = 0; i < circles.size(); i++) {
            Point center(cvRound(circles[i][0]), cvRound(circles[i][1]));
            int radius = cvRound(circles[i][2]);
            // draw the circle center
            circle(img_show, center, 3, Scalar(0, 0, 255), -1);
            // draw the circle outline
            circle(img_show, center, radius, Scalar(0, 0, 255), 3, 8, 0);
        }
        imshow("Shapes", img_show);
    }
    else if(shape == 0) {       // Lines
        Mat edges;
        Canny(img, edges, 50, 100);
        // Find lines
        vector<Vec2f> lines;
        HoughLines(edges, lines, 1, CV_PI/180.f, thresh);
        // Draw lines
        Mat img_show = img.clone();
        for(int i = 0; i < lines.size(); i++) {
            float rho = lines[i][0];
            float theta = lines[i][1];
            double a = cos(theta), b = sin(theta);
            double x0 = a * rho, y0 = b * rho;
            Point pt1(cvRound(x0 + 1000 * (-b)), cvRound(y0 + 1000 * (a)));
            Point pt2(cvRound(x0 - 1000 * (-b)), cvRound(y0 - 1000 * (a)));
            line(img_show, pt1, pt2, Scalar(0, 0, 255));
        }
        imshow("Shapes", img_show);
    }
}

int main() {
    img = imread("hough.jpg");

    namedWindow("Shapes");
```

```
    // Create sliders
    createTrackbar("Lines/Circles", "Shapes", &shape, 1, on_trackbar);
    createTrackbar("Acc. Thresh.", "Shapes", &thresh, 300, on_trackbar);

    // Initialize window
    on_trackbar(0, 0);

    while(char(waitKey(1)) != 'q') {}

    return 0;
}
```

Generalized Hough Transform

The generalized Hough Transform can be used to detect irregular shapes in an image that do not follow a simple equation. Theoretically, any shape can be represented by an equation, but recall that as the number of parameters in the equation increases, so do the dimensions of the required accumulator. The Wikipedia article and http://homepages.inf.ed.ac.uk/rbf/HIPR2/hough.htm are good introductions to the generalized Hough transform.

RANdom Sample Consensus (RANSAC)

RANSAC is a powerful framework for finding data-points that fit to a particular model. The model that we will consider here to apply RANSAC on is the conic section model of the ellipse. RANSAC is quite resistant to "outliers"—data-points that do not fit the given model. To explain the algorithm itself, let us consider the problem of finding a line in a noisy dataset of 2D points. The strategy is:

- Randomly sample points from the dataset
- Find the equation of a line that fits those points
- Find "inliers"—points that fit this model of the line. For determining if a point is an inlier or outlier with respect to a model, we need a measure of distance. Here, that measure will be the Euclidean distance of the line from the point. We will decide a value of this distance measure that acts as threshold for inlier/outlier determination
- Iterate till you find a line that has more inliers than a certain pre-decided number

Sounds simple, right? Yet this simple mechanism is really robust to noise, as you will observe in the ellipse detection example. Before going further, you should read:

- The Wikipedia article on RANSAC, because we will use the RANSAC algorithm framework outlined there, and it has some good visualizations that will help you develop an intuitive understanding of the algorithm
- The Wolfram Alpha article on ellipses (http://mathworld.wolfram.com/Ellipse.html), especially equations 15 to 23 for the mathematical formulas that we will use to calculate various properties of the ellipse
- The article http://nicky.vanforeest.com/misc/fitEllipse/fitEllipse.html, which describes the strategy we will use to fit an ellipse to a set of points. Understanding this article requires knowledge of matrix algebra, especially eigenvalues and eigenvectors

For this app, I decided to adopt an object oriented strategy, to leverage the actual power of C++. Functional programming (like the programming we have been doing so far—using functions for different tasks) is okay for small applications. However, it is significantly advantageous in terms of code readability and strategy formation to use an object oriented strategy for larger applications, and it runs just as fast. The flow of the algorithm is as follows:

- Detect Canny edges in the image using a tight threshold and extract contours so that spurious contours are avoided. Also reject contours that are less than a certain threshold in length, using the function `arcLength()` to measure contour length
- Have a certain number of iterations for each contour in which:
  - a certain number of points are selected at random from the contour and fitted to an ellipse to those points. Decide the number of inliers to this ellipse in the current contour based on a distance threshold
  - the ellipse that best fits (has the greatest number of inliers) to the current contour is found. One should not use all the points in the contour at once to fit the ellipse, because a contour is likely to be noisy, and consequently the ellipse fitted to all the points in the contour might not be the best. By randomly selecting points and doing this many times, one gives the algorithm more chance to find the best fit
- Repeat this process for all the contours, and find the ellipse that has the most number of inliers

For accomplishing this algorithm I use the following RANSAC parameters:

- Number of iterations where we choose random points. Decreasing this lessens the chance of finding the best set of random points to represent the ellipse, while increasing this parameters increases the chance but causes the algorithm to take more processing time
- Number of random points to be considered at each iteration. Choosing too low a value, such as 4 or 5, will not define an ellipse properly, while choosing too high a value will increase the chance of drawing a point from an undesirable part of the contour. Hence, this parameter must be turned carefully
- Threshold on distance between point and ellipse for the point to be considered inlier to the ellipse
- Minimum number of inliers for an ellipse to be considered for further processing. This is checked on the ellipse fitted to the random points chosen at each iteration. Increasing this parameter makes the algorithm stricter

The code itself is quite simple. It is long, and I have also written a bunch of debug functions that you can activate to see some debugging images (e.g., the best contour chosen by the algorithm, and so forth). The function `debug()` allows you to fit an ellipse to a specific contour. Be advised that this uses all the points in the contour at once, and the result is likely to be not as good as the random points strategy discussed earlier. We use the Eigen library for finding eigenvalues and eigenvectors to fit an ellipse to a set of points, as for unknown reasons the OpenCV `eigen()` does not work as expected. The code in Listing 6-5 is not optimized, so it might take some time to process bigger images with lots of contours. It is recommended that you use as tight Canny thresholds as possible to reduce the number of contours the algorithm has to process. You can also change the RANSAC parameters if needed.

Because we use the Eigen library in addition to OpenCV, you should of course install it if you don't have it (although most Ubuntu systems should have an installation of Eigen), and compile the code as follows:

```
g++ -I/usr/include/eigen3 'pkg-config opencv --cflags' findEllipse.cpp -o findEllipse 'pkg-config
opencv --libs'
```

You can install Eigen by typing `sudo apt-get install libeigen3-dev` in a terminal.

The code is shown in Listing 6-5.

Listing 6-5. Program to find the largest ellipse using RANSAC

```
// Program to find the largest ellipse using RANSAC
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania

#include <Eigen/Dense>
#include <opencv2/core/eigen.hpp>
#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <algorithm>
#include <math.h>

#define PI 3.14159265

using namespace std;
using namespace cv;
using namespace Eigen;

// Class to hold RANSAC parameters
class RANSACparams {
    private:
        //number of iterations
        int iter;
        //minimum number of inliers to further process a model
        int min_inliers;
        //distance threshold to be conted as inlier
        float dist_thresh;
        //number of points to select randomly at each iteration
        int N;
    public:
        RANSACparams(int _iter, int _min_inliers, float _dist_thresh, int _N) { //constructor
            iter = _iter;
            min_inliers = _min_inliers;
            dist_thresh = _dist_thresh;
            N = _N;
        }

        int get_iter() {return iter;}
        int get_min_inliers() {return min_inliers;}
        float get_dist_thresh() {return dist_thresh;}
        int get_N() {return N;}
};

// Class that deals with fitting an ellipse, RANSAC and drawing the ellipse in the image
class ellipseFinder {
    private:
        Mat img;                                  // input image
        vector<vector<Point> > contours;          // contours in image
        Mat Q;                                    // Matrix representing conic section of detected ellipse
        Mat fit_ellipse(vector<Point>);           // function to fit ellipse to a contour
        Mat RANSACellipse(vector<vector<Point> >); // function to find ellipse in contours using RANSAC
```

```
        bool is_good_ellipse(Mat);              // function that determines whether given conic
section represents a valid ellipse
        vector<vector<Point> > choose_random(vector<Point>); //function to choose points at
random from contour
        vector<float> distance(Mat, vector<Point>);  //function to return distance of points from the ellipse
        float distance(Mat, Point);                 //overloaded function to return signed distance of
point from ellipse
        void draw_ellipse(Mat);                 //function to draw ellipse in an image
        vector<Point> ellipse_contour(Mat);     //function to convert equation of ellipse to a
contour of points
        void draw_inliers(Mat, vector<Point>);  //function to debug inliers

        // RANSAC parameters
        int iter, min_inliers, N;
        float dist_thresh;

    public:
        ellipseFinder(Mat _img, int l_canny, int h_canny, RANSACparams rp) { // constructor
            img = _img.clone();

            // Edge detection and contour extraction
            Mat edges; Canny(img, edges, l_canny, h_canny);
            vector<vector<Point> > c;
            findContours(edges, c, CV_RETR_LIST, CV_CHAIN_APPROX_NONE);
            // Remove small spurious short contours
            for(int i = 0; i < c.size(); i++) {
                bool is_closed = false;

                vector<Point> _c = c[i];

                Point p1 = _c.front(), p2 = _c.back();
                float d = sqrt(pow(p1.x - p2.x,2) + pow(p1.y - p2.y,2));
                if(d <= 0.5) is_closed = true;

                d = arcLength(_c, is_closed);

                if(d > 50) contours.push_back(_c);
            }

            iter = rp.get_iter();
            min_inliers = rp.get_min_inliers();
            N = rp.get_N();
            dist_thresh = rp.get_dist_thresh();

            Q = Mat::eye(6, 1, CV_32F);

            /*
            //for debug
            Mat img_show = img.clone();
            drawContours(img_show, contours, -1, Scalar(0, 0, 255));
            imshow("Contours", img_show);
```

```
            //imshow("Edges", edges);
            */
            cout << "No. of Contours = " << contours.size() << endl;
        }

        void detect_ellipse(); //final wrapper function
        void debug();          //debug function
};

vector<float> ellipseFinder::distance(Mat Q, vector<Point> c) {
    vector<Point> ellipse = ellipse_contour(Q);
    vector<float> distances;
    for(int i = 0; i < c.size(); i++) {
        distances.push_back(float(pointPolygonTest(ellipse, c[i], true)));
    }

    return distances;
}

float ellipseFinder::distance(Mat Q, Point p) {
    vector<Point> ellipse = ellipse_contour(Q);
    return float(pointPolygonTest(ellipse, p, true));
}

vector<vector<Point> > ellipseFinder::choose_random(vector<Point> c) {
    vector<vector<Point> > cr;
    vector<Point> cr0, cr1;
    // Randomly shuffle all elements of contour
    std::random_shuffle(c.begin(), c.end());
    // Put the first N elements into cr[0] as consensus set (see Wikipedia RANSAC algorithm) and
    // the rest in cr[1] to check for inliers
    for(int i = 0; i < c.size(); i++) {
        if(i < N) cr0.push_back(c[i]);
        else cr1.push_back(c[i]);
    }
    cr.push_back(cr0);
    cr.push_back(cr1);

    return cr;
}

Mat ellipseFinder::fit_ellipse(vector<Point> c) {
    /*
    // for debug
    Mat img_show = img.clone();
    vector<vector<Point> > cr;
    cr.push_back(c);
    drawContours(img_show, cr, -1, Scalar(0, 0, 255), 2);
    imshow("Debug fitEllipse", img_show);
    */
    int N = c.size();
```

```
    Mat D;
    for(int i = 0; i < N; i++) {
        Point p = c[i];
        Mat r(1, 6, CV_32FC1);
        r = (Mat_<float>(1, 6) << (p.x)*(p.x), (p.x)*(p.y), (p.y)*(p.y), p.x, p.y, 1.f);
        D.push_back(r);
    }
    Mat S = D.t() * D, _S;

    double d = invert(S, _S);
    //if(d < 0.001) cout << "S matrix is singular" << endl;

    Mat C = Mat::zeros(6, 6, CV_32F);
    C.at<float>(2, 0) = 2;
    C.at<float>(1, 1) = -1;
    C.at<float>(0, 2) = 2;

    // Using EIGEN to calculate eigenvalues and eigenvectors
    Mat prod = _S * C;
    Eigen::MatrixXd prod_e;
    cv2eigen(prod, prod_e);
    EigenSolver<Eigen::MatrixXd> es(prod_e);

    Mat evec, eval, vec(6, 6, CV_32FC1), val(6, 1, CV_32FC1);

    eigen2cv(es.eigenvectors(), evec);
    eigen2cv(es.eigenvalues(), eval);

    evec.convertTo(evec, CV_32F);
    eval.convertTo(eval, CV_32F);

    // Eigen returns complex parts in the second channel (which are all 0 here) so select just the
first channel
    int from_to[] = {0, 0};
    mixChannels(&evec, 1, &vec, 1, from_to, 1);
    mixChannels(&eval, 1, &val, 1, from_to, 1);

    Point maxLoc;
    minMaxLoc(val, NULL, NULL, NULL, &maxLoc);

    return vec.col(maxLoc.y);
}

bool ellipseFinder::is_good_ellipse(Mat Q) {
    float a = Q.at<float>(0, 0),
          b = (Q.at<float>(1, 0))/2,
          c = Q.at<float>(2, 0),
          d = (Q.at<float>(3, 0))/2,
          f = (Q.at<float>(4, 0))/2,
          g = Q.at<float>(5, 0);
```

```
    if(b*b - a*c == 0) return false;

    float thresh = 0.09,
        num = 2 * (a*f*f + c*d*d + g*b*b - 2*b*d*f - a*c*g),
        den1 = (b*b - a*c) * (sqrt((a-c)*(a-c) + 4*b*b) - (a + c)),
        den2 = (b*b - a*c) * (-sqrt((a-c)*(a-c) + 4*b*b) - (a + c)),
        a_len = sqrt(num / den1),
        b_len = sqrt(num / den2),
        major_axis = max(a_len, b_len),
        minor_axis = min(a_len, b_len);

    if(minor_axis < thresh*major_axis || num/den1 < 0.f || num/den2 < 0.f || major_axis >
max(img.rows, img.cols)) return false;
    else return true;
}

Mat ellipseFinder::RANSACellipse(vector<vector<Point> > contours) {
    int best_overall_inlier_score = 0;
    Mat Q_best = 777 * Mat::ones(6, 1, CV_32FC1);
    int idx_best = -1;

    //for each contour...
    for(int i = 0; i < contours.size(); i++) {
        vector<Point> c = contours[i];
        if(c.size() < min_inliers) continue;

        Mat Q;
        int best_inlier_score = 0;
        for(int j = 0; j < iter; j++) {
            // ...choose points at random...
            vector<vector<Point> > cr = choose_random(c);
            vector<Point> consensus_set = cr[0], rest = cr[1];
            // ...fit ellipse to those points...
            Mat Q_maybe = fit_ellipse(consensus_set);
            // ...check for inliers...
            vector<float> d = distance(Q_maybe, rest);
            for(int k = 0; k < d.size(); k++)
                if(abs(d[k]) < dist_thresh) consensus_set.push_back(rest[k]);
            // ...and find the random set with the most number of inliers
            if(consensus_set.size() > min_inliers && consensus_set.size() > best_inlier_score) {
                Q = fit_ellipse(consensus_set);
                best_inlier_score = consensus_set.size();
            }
        }
        // find cotour with ellipse that has the most number of inliers
        if(best_inlier_score > best_overall_inlier_score && is_good_ellipse(Q)) {
            best_overall_inlier_score = best_inlier_score;
            Q_best = Q.clone();
            if(Q_best.at<float>(5, 0) < 0) Q_best *= -1.f;
            idx_best = i;
        }
    }
```

```
    /*
    //for debug
    Mat img_show = img.clone();
    drawContours(img_show, contours, idx_best, Scalar(0, 0, 255), 2);
    imshow("Best Contour", img_show);

    cout << "inliers " << best_overall_inlier_score << endl;
    */
    if(idx_best >= 0) draw_inliers(Q_best, contours[idx_best]);
    return Q_best;
}

vector<Point> ellipseFinder::ellipse_contour(Mat Q) {
    float a = Q.at<float>(0, 0),
          b = (Q.at<float>(1, 0))/2,
          c = Q.at<float>(2, 0),
          d = (Q.at<float>(3, 0))/2,
          f = (Q.at<float>(4, 0))/2,
          g = Q.at<float>(5, 0);

    vector<Point> ellipse;
    if(b*b - a*c == 0) {
        ellipse.push_back(Point(0, 0));
        return ellipse;
    }

    Point2f center((c*d - b*f)/(b*b - a*c), (a*f - b*d)/(b*b - a*c));

    float num = 2 * (a*f*f + c*d*d + g*b*b - 2*b*d*f - a*c*g),
        den1 = (b*b - a*c) * (sqrt((a-c)*(a-c) + 4*b*b) - (a + c)),
        den2 = (b*b - a*c) * (-sqrt((a-c)*(a-c) + 4*b*b) - (a + c)),
        a_len = sqrt(num / den1),
        b_len = sqrt(num / den2),
        major_axis = max(a_len, b_len),
        minor_axis = min(a_len, b_len);

    //angle of rotation of ellipse
    float alpha = 0.f;
    if(b == 0.f && a == c) alpha = PI/2;
    else if(b != 0.f && a > c) alpha = 0.5 * atan2(2*b, a-c);
    else if(b != 0.f && a < c) alpha = PI/2 - 0.5 * atan2(2*b, a-c);
```

```
    // 'draw' the ellipse and put it into a STL Point vector so you can use drawContours()
    int N = 200;
    float theta = 0.f;
    for(int i = 0; i < N; i++, theta += 2*PI/N) {
        float x = center.x + major_axis*cos(theta)*cos(alpha) + minor_axis*sin(theta)*sin(alpha);
        float y = center.y - major_axis*cos(theta)*sin(alpha) + minor_axis*sin(theta)*cos(alpha);
        Point p(x, y);
        if(x < img.cols && y < img.rows) ellipse.push_back(p);
    }
    if(ellipse.size() == 0) ellipse.push_back(Point(0, 0));

    return ellipse;
}

void ellipseFinder::detect_ellipse() {
    Q = RANSACellipse(contours);
    cout << "Q" << Q << endl;
    draw_ellipse(Q);
}

void ellipseFinder::debug() {
    int i = 1; //index of contour you want to debug
    cout << "No. of points in contour " << contours[i].size() << endl;
    Mat a = fit_ellipse(contours[i]);
    Mat img_show = img.clone();
    drawContours(img_show, contours, i, Scalar(0, 0, 255), 3);
    imshow("Debug contour", img_show);
    draw_inliers(a, contours[i]);
    draw_ellipse(a);
}

void ellipseFinder::draw_ellipse(Mat Q) {
    vector<Point> ellipse = ellipse_contour(Q);
    vector<vector<Point> > c;
    c.push_back(ellipse);
    Mat img_show = img.clone();
    drawContours(img_show, c, -1, Scalar(0, 0, 255), 3);
    imshow("Ellipse", img_show);
}

void ellipseFinder::draw_inliers(Mat Q, vector<Point> c) {
    vector<Point> ellipse = ellipse_contour(Q);
    vector<vector<Point> > cs;
    cs.push_back(ellipse);
    Mat img_show = img.clone();
```

```
    // draw all contours in thin red
    drawContours(img_show, contours, -1, Scalar(0, 0, 255));
    // draw ellipse in thin blue
    drawContours(img_show, cs, 0, Scalar(255, 0, 0));
    int count = 0;
    // draw inliers as green points
    for(int i = 0; i < c.size(); i++) {
        double d = pointPolygonTest(ellipse, c[i], true);
        float d1 = float(d);
        if(abs(d1) < dist_thresh) {
            circle(img_show, c[i], 1, Scalar(0, 255, 0), -1);
            count ++;
        }
    }
    imshow("Debug inliers", img_show);
    cout << "inliers " << count << endl;
}

int main() {
    Mat img = imread("test4.jpg");
    namedWindow("Ellipse");

    // object holding RANSAC parameters, initialized using the constructor
    RANSACparams rp(400, 100, 1, 5);

    // Canny thresholds
    int canny_l = 250, canny_h = 300;
    // Ellipse finder object, initialized using the constructor
    ellipseFinder ef(img, canny_l, canny_h, rp);
    ef.detect_ellipse();
    //ef.debug();

    while(char(waitKey(1)) != 'q') {}

    return 0;
}
```

Figure 6-6 shows the algorithm in action.

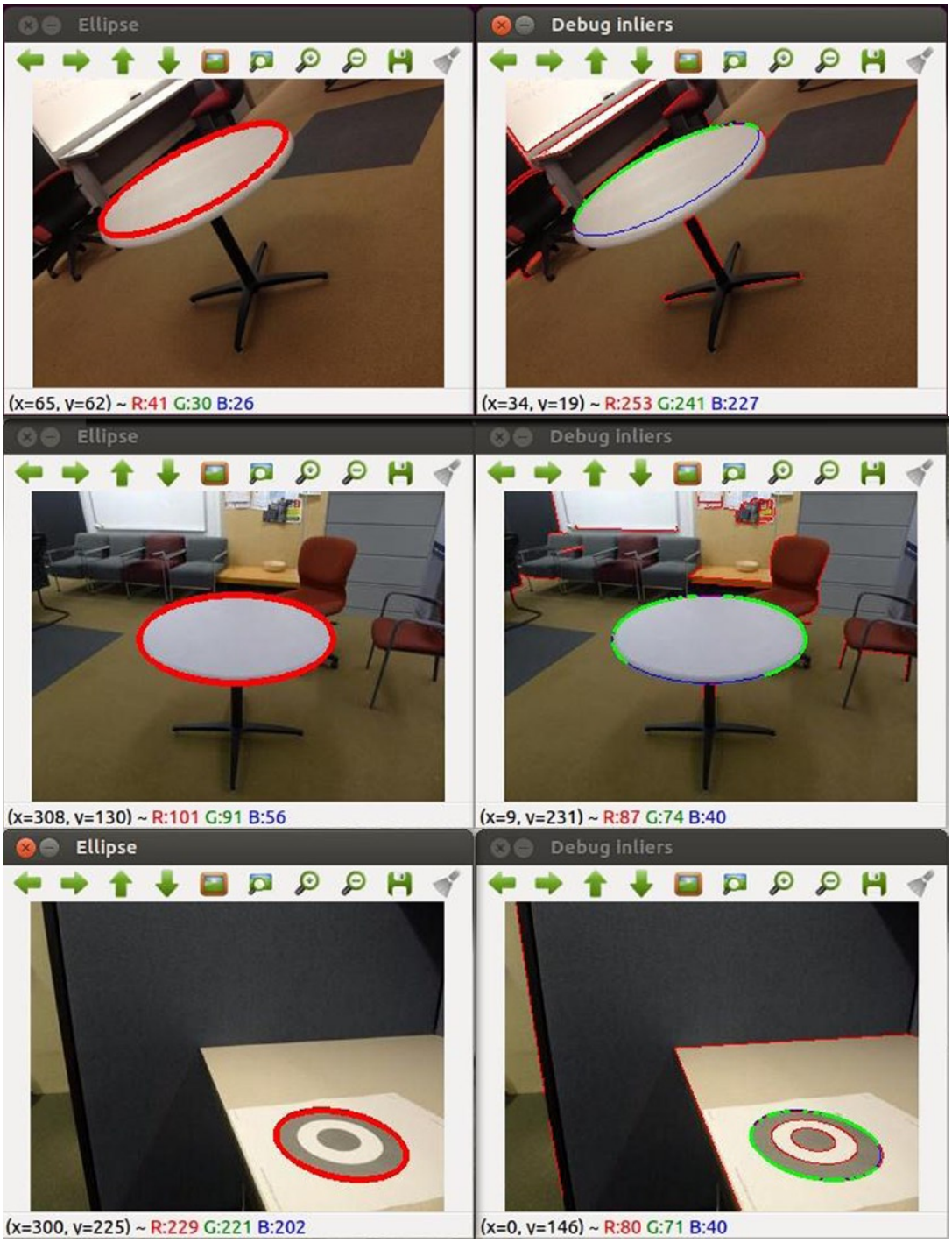


Figure 6-6. *Finding the largest ellipse in the image using RANSAC*

Bounding Boxes and Circles

OpenCV offers functions that compute a rectangle or circle of the minimum area enclosing a set of points. The rectangle can be upright (in which case it will not be the rectangle with the minimum area enclosing the set of points) or rotated (in which case it will be). These shapes can be useful in two ways:

- Many high-level feature detection functions in OpenCV accept an upright rectangle as Region of Interest (ROI). A ROI is used often to speed up computation. Many times we want to use a computationally intensive function, for example the stereo block matching function that gives disparity from left and right images. However, we are interested in only a certain part of the image. By specifying a proper ROI, one can tell the function to ignore the other parts of the image, thereby not wasting computation on parts of images that we are not interested in.
- One can use the properties of these bounding boxes and circles (e.g., area, aspect ratio) to infer roughly the size of an object or distance of an object from the camera.

The function `minAreaRrect()` calculates a rotated rectangle, `minEnclosingCircle()` calculates a circle, and `boundingRect()` calculates the upright rectangle of the smallest size enclosing a set of points. The set of points is usually specified as a STL vector, but you can do so using a Mat, too. In Listing 6-6, take a look at the modified version of `draw_ellipse()` from our ellipse detector code to learn how to use these functions. This version of the app not only finds an ellipse in an image but also shows the bounding rectangles and circle around the ellipse as shown in Figure 6-7.

Listing 6-6. Modified draw_ellipse() to show bounding circle and rectangles

```
void ellipseFinder::draw_ellipse(Mat Q) {
    vector<Point> ellipse = ellipse_contour(Q);
    vector<vector<Point> > c;
    c.push_back(ellipse);
    Mat img_show = img.clone();
    //draw ellipse
    drawContours(img_show, c, -1, Scalar(0, 0, 255), 3);

    //compute bounding shapes
    RotatedRect r_rect = minAreaRect(ellipse);
    Rect rect = boundingRect(ellipse);
    Point2f center; float radius; minEnclosingCircle(ellipse, center, radius);

    //draw bounding shapes
    rectangle(img_show, rect, Scalar(255, 0, 255));                          //magenta
    circle(img_show, center, radius, Scalar(255, 0, 0));                     //blue
    Point2f vertices[4]; r_rect.points(vertices);
    for(int i = 0; i < 4; i++)
        line(img_show, vertices[i], vertices[(i + 1) % 4], Scalar(0, 255, 0)); //green

    imshow("Ellipse", img_show);
}
```

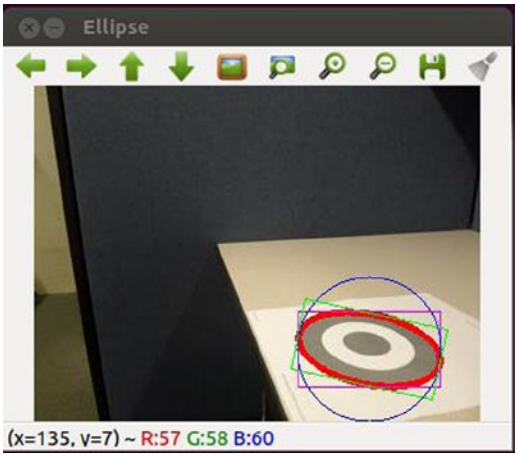


Figure 6-7. *Bounding circle and rectangles (upright and rotated) for the ellipse detected using RANSAC*

Convex Hulls

A convex hull for a set of 2D points is the smallest convex closed contour that encloses all the points. Because it is convex, the line joining any two points on the convex hull does not intersect the hull boundary. Because it has to be the smallest, the convex hull of a set of points is a subset of this set. The OpenCV function `convexHull()` can calculate the hull for a set of points. It gives its output in two different forms—a vector of indices into the input contour for points that constitute the hull, or the hull points themselves. OpenCV has a nice convex hull demo, which randomly selects a set of points and draws the hull around it. Figure 6-8 shows it in action.

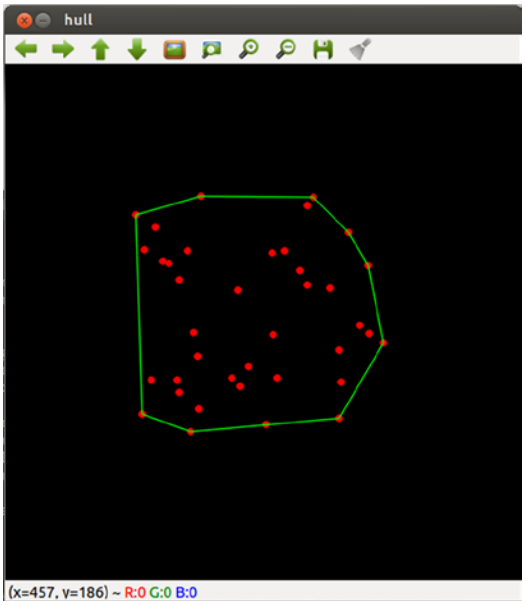


Figure 6-8. *OpenCV convex hull demo*

The motivation behind computing the convex hull is that you get the smallest possible closed contour that is convex and encloses all the points in your set. You can then use `contourArea()` and `arcLength()` to estimate the area and perimeter of your set of points, respectively.

As an exercise, you might want to take the color-based object detector code from the last chapter and modify it to get the area and perimeter of the red object. Here's what you will need to do:

- Extract contours in the binary output of the detector
- Convert those contours to closed contours using either `approxPolyDP()` or `convexHull()`. `convexHull()` will probably be overkill for this and slower, too.
- Find area and/or perimeter of closed contours with `contourArea()` and/or `arcLength()` respectively, and display the contour with the largest area as that is most likely to be your desired object

Summary

This chapter talked a lot about dealing with shapes in images. You might have noticed that we have now moved on to a lot of "higher"-level algorithms and I do not dwell much on lower pixel-level procedures. This is because I assume that you have read through the previous chapters carefully! Shapes are a simple way to recognize known objects in images. But they are also prone to a lot of errors, especially because of occlusion of a part of the object by another object. The other method to recognize objects, the keypoint-feature-based approach, solves this, and we will discuss keypoint-based object recognition in a later chapter.

Meanwhile, I want to stress the importance of the ellipse detection program that I introduced in this chapter. It is important because it has a vital trait of a real-world computer vision program—it does not just put together a bunch of built-in OpenCV functions, but it involves using them to aid your own indigenous algorithm instead. It is also object-oriented, which is the preferred style of writing large programs. If you do not understand any line in that code, please refer the online OpenCV documentation for that function and/or consult the previous chapters.

CHAPTER 7

Image Segmentation and Histograms

Welcome to the seventh chapter! After discussing shapes and contours in the last chapter, I want to talk about the very important topic of image segmentation, which is one of the most fundamental computer vision problems attracting a lot of research today. We will discuss some of the segmentation algorithms that OpenCV has up its sleeve, and also how you can use other techniques such as morphological operations to make your own custom segmentation algorithm.

Specifically, we will begin with the naïve segmentation technique—thresholding and work our way up the complexity ladder with floodFill, watershed segmentation and grabCut segmentation. We will also make our own segmentation algorithm that builds on top of floodFill to count objects in a photo.

Toward the end of the chapter, I have also discussed image histograms and their applications, which are useful preprocessing steps.

Image Segmentation

Image segmentation can be defined as a procedure to separate visually different parts of the image. In fact, we have been doing some rudimentary image segmentation for our color-based object detector app already! In this chapter, you will learn some techniques that will help you to better the object detector app by a large margin.

Now, as being "visually" different is a subjective property that can change with the problem at hand, segmentation is also usually considered correct when it is able to divide the image properly for the desired purpose. For example, let us consider the same image (Figure 7-1) for which we want to solve two different problems—counting balls and extracting foreground. Both problems are, as you might have realized, problems of proper image segmentation. For the problem of counting balls we will need a segmentation algorithm that will divide the image into all the visually different regions—background, hand, and balls. On the contrary, for the other problem (that of extracting foreground), it is acceptable that the hand and balls be regarded as one single region.

Figure 7-1. *Image for two segmentation problems: Foreground extraction and counting objects*

A word of caution about code and syntax: I will now introduce functions and discuss strategies to use them and not spend too much time on the syntax. This is because I expect you to consult the online OpenCV documentation (`http://docs.opencv.org`) for those functions. Moreover, introduction of a function is usually followed by code that uses that function in a constructive manner. Seeing that usage example will further help you understand the syntax aspect of the function.

Simple Segmentation by Thresholding

One of the simplest segmentation strategies is to threshold color values (and that is what we do for the color-based object detector). OpenCV has functions `threshold()` (threshold values in a `Mat` with different options for the action taken when the threshold is exceeded and when the threshold is followed—check out the documentation!), `inRange()` (apply a low and high threshold on values in a `Mat`), and `adaptiveThreshold()` (same as `threshold()`, except that the threshold for each pixel depends values of its neighbors in a patch, the size of which is defined by the user) to help you with thresholding. We have so far used `inRange()` in the RGB values of pixels, and found that simple RGB values are very sensitive to illumination.

However, illumination just affects the intensity of pixels. The problem with thresholding in the RBG colorspace is that the intensity information is shared by all of R, G, and B (Intensity = 30% R + 59% G + 11% B). The HSV (hue, saturation, value) colorspace, by contrast, encodes intensity (brightness) in just V. H and S encode solely color information. H represents the actual color, while S represents its "strength" or purity. `http://www.dig.cs.gc.cuny.edu/manuals/Gimp2/Grokking-the-GIMP-v1.0/node51.html` is a good place if you want to read more on the HSV color space. This is great, because we can now just threshold the H and S channels in the image, and get a lot of illumination invariance. Listing 7-1 therefore is the same as our last object detector code, except that it converts the frames to the HSV colorspace first using `cvtColor()` and thresholds the H and S channels. Figure 7-2 shows the code in action, but you should experiment with it under different illumination conditions to see how much invariance you get.

Listing 7-1. Color-based object detection using Hue and Saturation thresholding

```
// Program to display a video from attached default camera device and detect colored blobs
using H and S thresholding
// Remove noise using opening and closing morphological operations
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania
```

```
#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

using namespace cv;
using namespace std;

int hs_slider = 0, low_slider = 30, high_slider = 100;
int low_h = 30, low_s = 30, high_h = 100, high_s = 100;

void on_hs_trackbar(int, void *) {
    switch(hs_slider) {
        case 0:
            setTrackbarPos("Low threshold", "Segmentation", low_h);
            setTrackbarPos("High threshold", "Segmentation", high_h);
            break;
        case 1:
            setTrackbarPos("Low threshold", "Segmentation", low_s);
            setTrackbarPos("High threshold", "Segmentation", high_s);
            break;
    }
}

void on_low_thresh_trackbar(int, void *) {
    switch(hs_slider) {
        case 0:
            low_h = min(high_slider - 1, low_slider);
            setTrackbarPos("Low threshold", "Segmentation", low_h);
            break;
        case 1:
            low_s = min(high_slider - 1, low_slider);
            setTrackbarPos("Low threshold", "Segmentation", low_s);
            break;
    }
}

void on_high_thresh_trackbar(int, void *) {
    switch(hs_slider) {
        case 0:
            high_h = max(low_slider + 1, high_slider);
            setTrackbarPos("High threshold", "Segmentation", high_h);
            break;
        case 1:
            high_s = max(low_slider + 1, high_slider);
            setTrackbarPos("High threshold", "Segmentation", high_s);
            break;
    }
}
```

```
int main()
{
    // Create a VideoCapture object to read from video file
    // 0 is the ID of the built-in laptop camera, change if you want to use other camera
    VideoCapture cap(0);

    //check if the file was opened properly
    if(!cap.isOpened())
    {
            cout << "Capture could not be opened succesfully" << endl;
            return -1;
    }

    namedWindow("Video");
    namedWindow("Segmentation");

    createTrackbar("0. H\n1. S", "Segmentation", &hs_slider, 1, on_hs_trackbar);
    createTrackbar("Low threshold", "Segmentation", &low_slider, 255, on_low_thresh_trackbar);
    createTrackbar("High threshold", "Segmentation", &high_slider, 255, on_high_thresh_trackbar);

    while(char(waitKey(1)) != 'q' && cap.isOpened())
    {
        Mat frame, frame_thresholded, frame_hsv;

        cap >> frame;

        cvtColor(frame, frame_hsv, CV_BGR2HSV);

        // Check if the video is over
        if(frame.empty())
        {
                cout << "Video over" << endl;
                break;
        }

        // extract the Hue and Saturation channels
        int from_to[] = {0,0, 1,1};
        Mat hs(frame.size(), CV_8UC2);
        mixChannels(&frame_hsv, 1, &hs, 1, from_to, 2);

        // check the image for a specific range of H and S
        inRange(hs, Scalar(low_h, low_s), Scalar(high_h, high_s), frame_thresholded);

        // open and close to remove noise
        Mat str_el = getStructuringElement(MORPH_ELLIPSE, Size(7, 7));
        morphologyEx(frame_thresholded, frame_thresholded, MORPH_OPEN, str_el);
        morphologyEx(frame_thresholded, frame_thresholded, MORPH_CLOSE, str_el);

        imshow("Video", frame);
        imshow("Segmentation", frame_thresholded);
    }

    return 0;
}
```

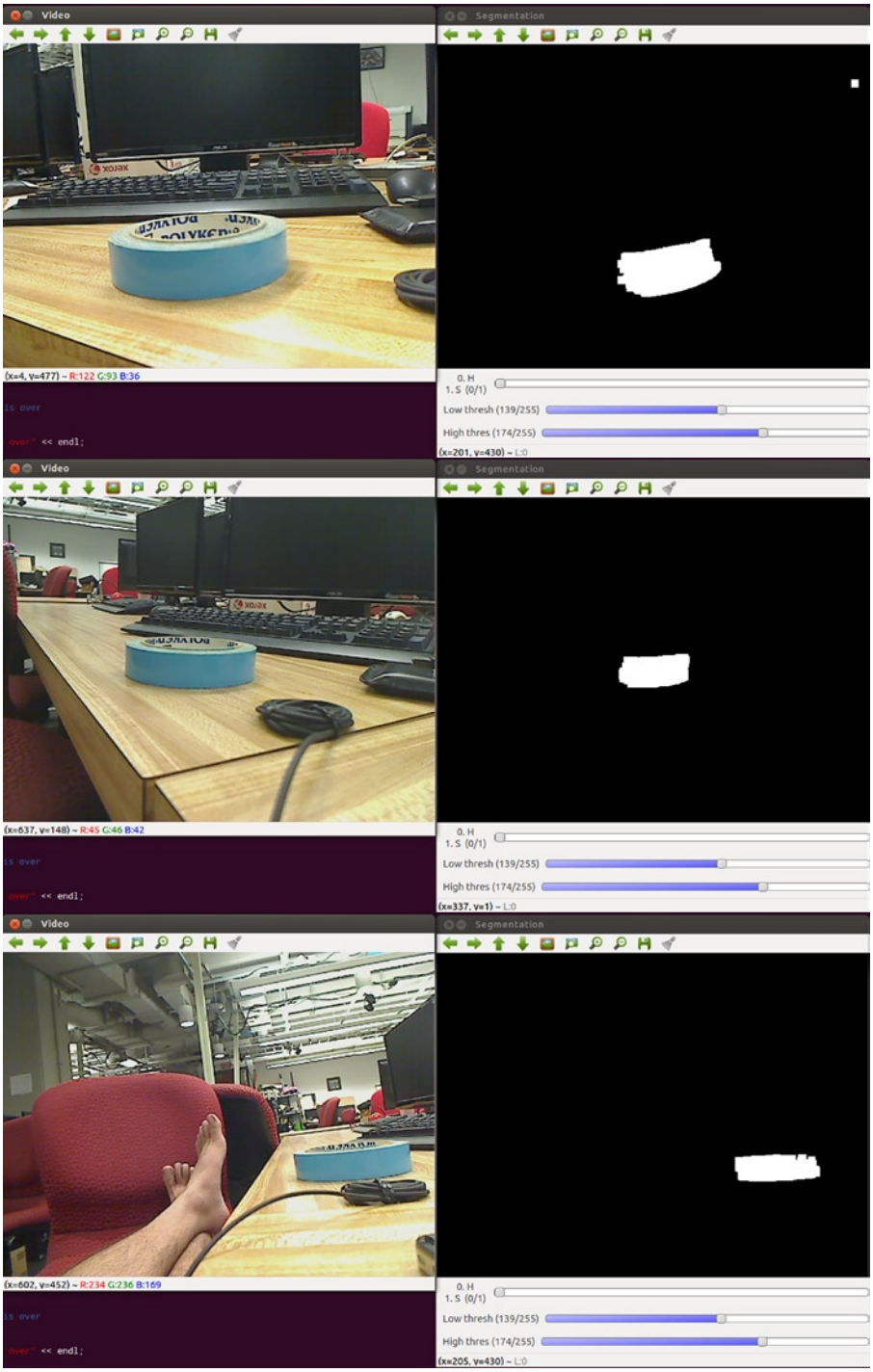


Figure 7-2. *Object detection by Hue and Saturation range-checking*

As you can see in Figure 7-2, I was trying to detect a blue colored object. You will face some difficulty if you try to set the ranges for red, because of an inherent property of the HSV colorspace. Hue is generally represented as a circle from 0 to 360 (OpenCV halves this number to store in a `CV_8U` image) with red wrapping around. This means the range of hue for red is roughly > 340 and < 20, with 0 coming after 360. So you will not be able to specify the whole range of the hue for red with the kind of slider processing we are using. It is an exercise for you to figure out how to process your slider values to address this problem. Hint: because the Hue is halved, the maximum possible hue in any image is 180. Make use of the extra space (255 – 180 = 75) that you have.

Floodfill

OpenCV's `floodFill()` function determines pixels in the image that are similar to a seed pixel, and connected to it. The similarity is usually defined by gray level in case of grayscale images, and by RGB values in case of a color image. The function takes the range of gray (or RGB) values around the gray (or RGB) value of the seed point that define whether a pixel should be considered similar to the seed pixel. There are two ways to determine if a pixel is connected to another pixel: 4-connectivity and 8-connectivity, which are evident from the cases shown in Figure 7-3.

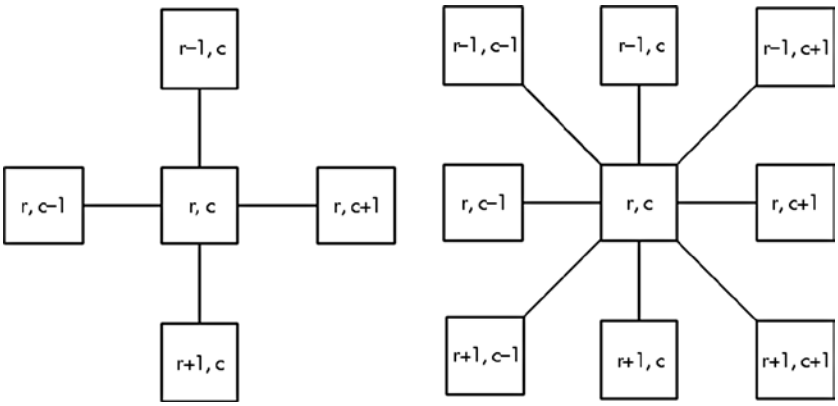


Figure 7-3. *4-connectivity (left) and 8-connectivity (right)*

The OpenCV floodFill demo is a lot of fun to play with. It allows you to click on an image to specify the seed point and use sliders to specify the upper and lower range around the seed point for defining similarity.

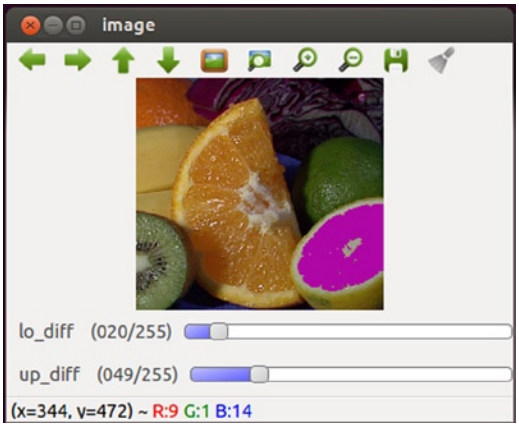


Figure 7-4. *OpenCV floodFill demo*

One can use the following strategy to utilize `floodFill()` to automate the color-based object detector app:

- Ask the user to click on that object
- Get the similar connected pixels to this point by `floodFill()`
- Convert this collection of pixels to HSV and, from it, decide the range of H and S values to check for using `inRange()`

Check out Listing 7-2, which is now our "intelligent" color-based object detector app!

Listing 7-2. Program to use floodFill to make the color-based object detector "intelligent"

```
// Program to automate the color-based object detector using floodFill
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

using namespace cv;
using namespace std;

Mat frame_hsv, frame, mask;

int low_diff = 10, high_diff = 10, conn = 4, val = 255, flags = conn + (val << 8) +
CV_FLOODFILL_MASK_ONLY;
double h_h = 0, l_h = 0, h_s = 0, l_s = 0;

bool selected = false;

void on_low_diff_trackbar(int, void *) {}

void on_high_diff_trackbar(int, void *) {}

void on_mouse(int event, int x, int y, int, void *) {
    if(event != EVENT_LBUTTONDOWN) return;

    selected = true;

    //seed point
    Point p(x, y);

    // make mask using floodFill
    mask = Scalar::all(0);
    floodFill(frame, mask, p, Scalar(255, 255, 255), 0, Scalar(low_diff, low_diff, low_diff),
Scalar(high_diff, high_diff, high_diff), flags);

    // find the H and S range of piexels selected by floodFill
    Mat channels[3];
    split(frame_hsv, channels);
    minMaxLoc(channels[0], &l_h, &h_h, NULL, NULL, mask.rowRange(1, mask.rows-1).colRange(1,
mask.cols-1));
    minMaxLoc(channels[1], &l_s, &h_s, NULL, NULL, mask.rowRange(1, mask.rows-1).colRange(1,
mask.cols-1));
}
```

```
int main() {
    // Create a VideoCapture object to read from video file
    // 0 is the ID of the built-in laptop camera, change if you want to use other camera
    VideoCapture cap(0);

    //check if the file was opened properly
    if(!cap.isOpened()) {
        cout << "Capture could not be opened succesfully" << endl;
        return -1;
    }

    namedWindow("Video");
    namedWindow("Segmentation");

    createTrackbar("Low Diff", "Segmentation", &low_diff, 50, on_low_diff_trackbar);
    createTrackbar("High Diff ", "Segmentation", &high_diff, 50, on_high_diff_trackbar);

    setMouseCallback("Video", on_mouse);

    while(char(waitKey(1)) != 'q' && cap.isOpened()) {
        cap >> frame;
        if(!selected) mask.create(frame.rows+2, frame.cols+2, CV_8UC1);
        // Check if the video is over
        if(frame.empty()) {
            cout << "Video over" << endl;
            break;
        }
        cvtColor(frame, frame_hsv, CV_BGR2HSV);

        // extract the hue and saturation channels
        int from_to[] = {0,0, 1,1};
        Mat hs(frame.size(), CV_8UC2);
        mixChannels(&frame_hsv, 1, &hs, 1, from_to, 2);

        // check for the range of H and S obtained from floodFill
        Mat frame_thresholded;
        inRange(hs, Scalar(l_h, l_s), Scalar(h_h, h_s), frame_thresholded);

        // open and close to remove noise
        Mat str_el = getStructuringElement(MORPH_RECT, Size(5, 5));
        morphologyEx(frame_thresholded, frame_thresholded, MORPH_OPEN, str_el);
        morphologyEx(frame_thresholded, frame_thresholded, MORPH_CLOSE, str_el);

        imshow("Video", frame);
        imshow("Segmentation", frame_thresholded);
    }

    return 0;
}
```

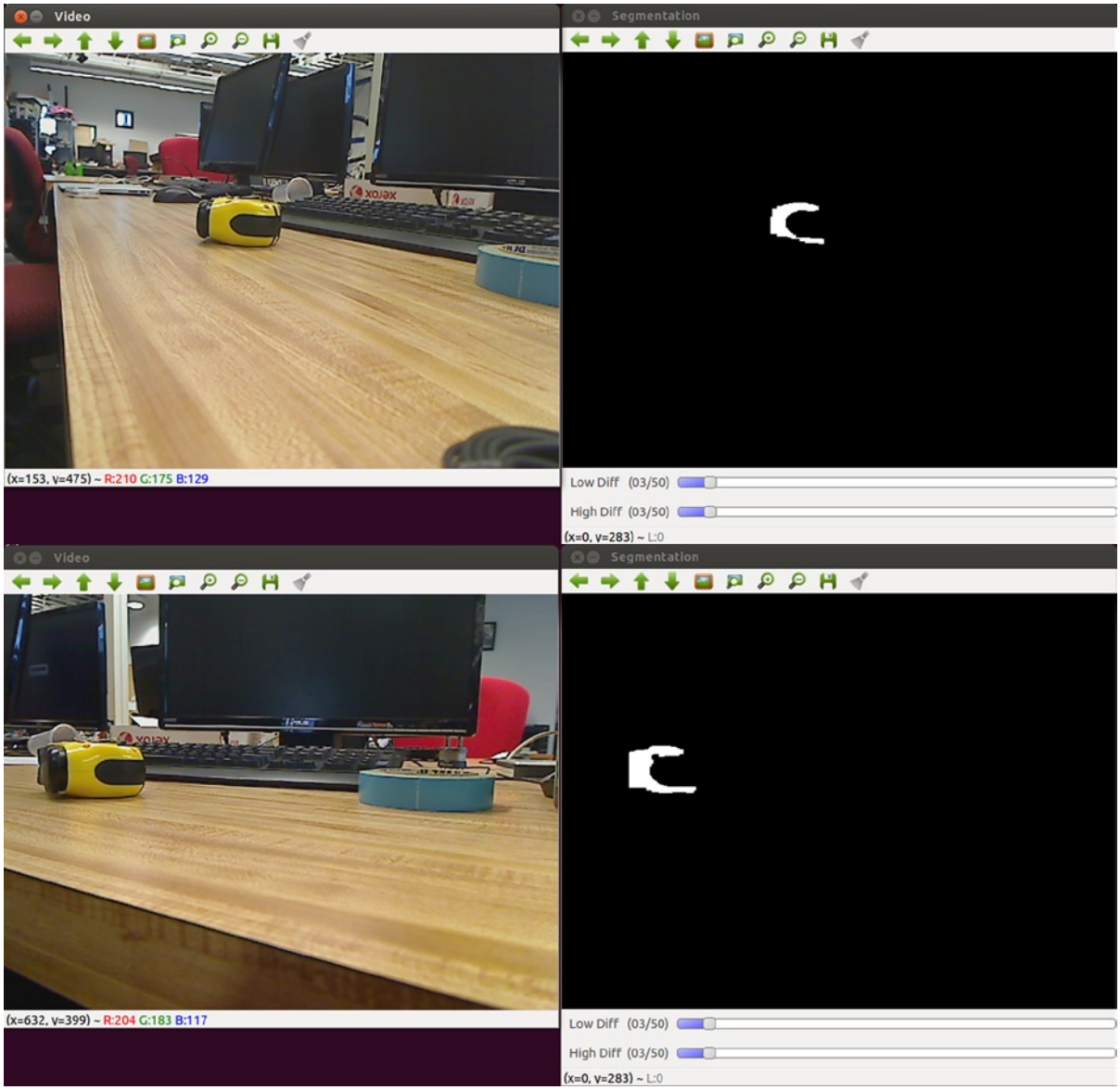


Figure 7-5. *Using floodFill in the color-based object detector*

Watershed Segmentation

The watershed algorithm is a good way to segment an image with objects that are touching each other, but the edges are not strong enough for accurate segmentation. Consider the example of an image of the top view a box of fruits, and the problem of segmenting the individual fruits to count them. Canny edges at a tight threshold are still too noisy, as shown in Figure 7-6. Fitting Hough circles (with a limit on the minimum radius of the circles) to these edges is obviously not going to work, as you can see in the same figure.

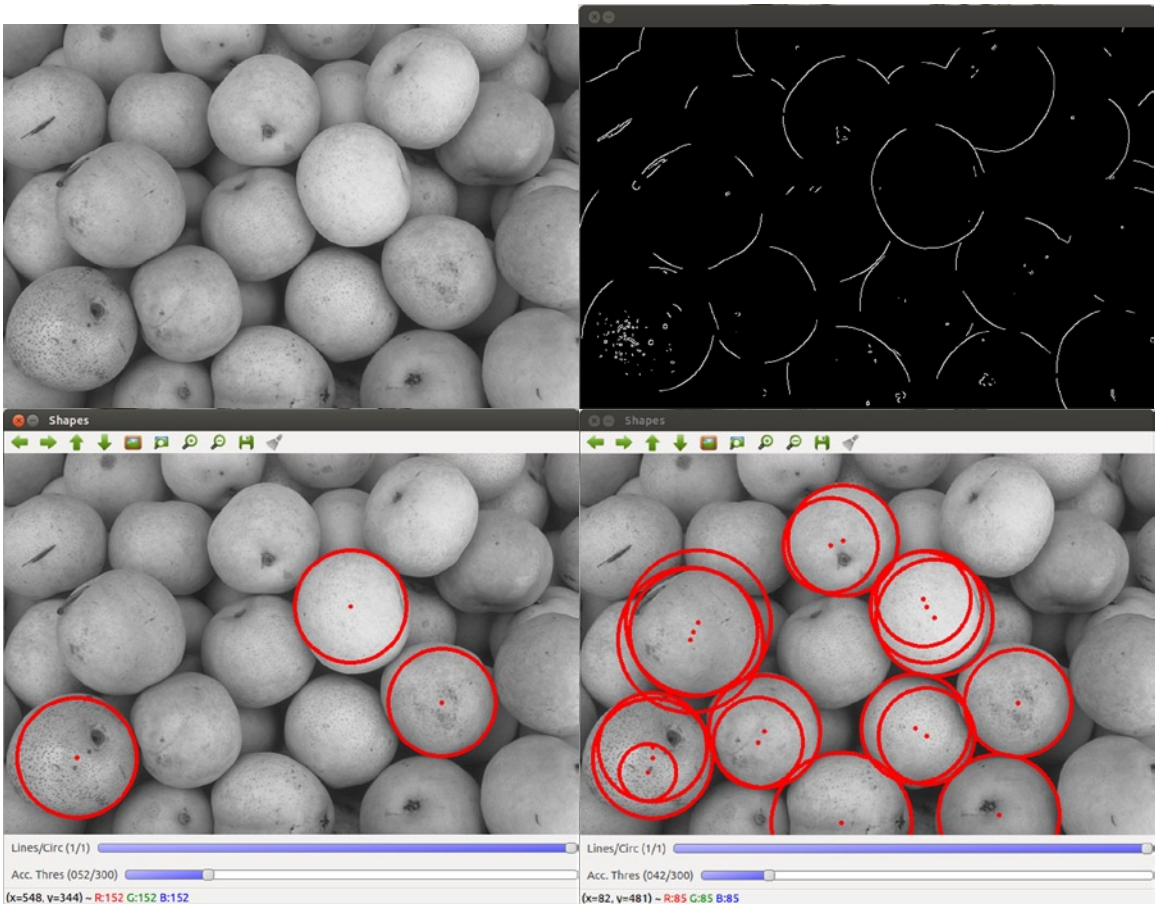


Figure 7-6. *(clockwise, from top left) original image, Canny edges with a tight threshold and attempts to fit circles to the image at different Hough accumulator thresholds*

Now let us see how watershed segmentation can help us here. Here is how it works:

- Use some function to decide the "height" of pixels in an image, for example, gray level
- Reconstruct the image to force all the regional minima to the same level. Hence "basins" are formed around the minima
- Flood this depth structure with liquid from the minima till the liquid reaches the highest point in the structure. Wherever liquids from two basins meet, a "dam" or "watershed line" is constructed to prevent them from mixing
- After the process is complete, the basins are returned as different regions of the image, while the watershed lines are the boundaries of the regions

The OpenCV function `watershed()` implements marker controlled watershed segmentation, which makes things a little bit easier for us. The user gives certain markers as input to the algorithm, which first reconstructs the image to have local minima only at these marker points. The rest of the process continues as described earlier. The OpenCV watershed demo makes the user mark these markers on the image (Figure 7-7).

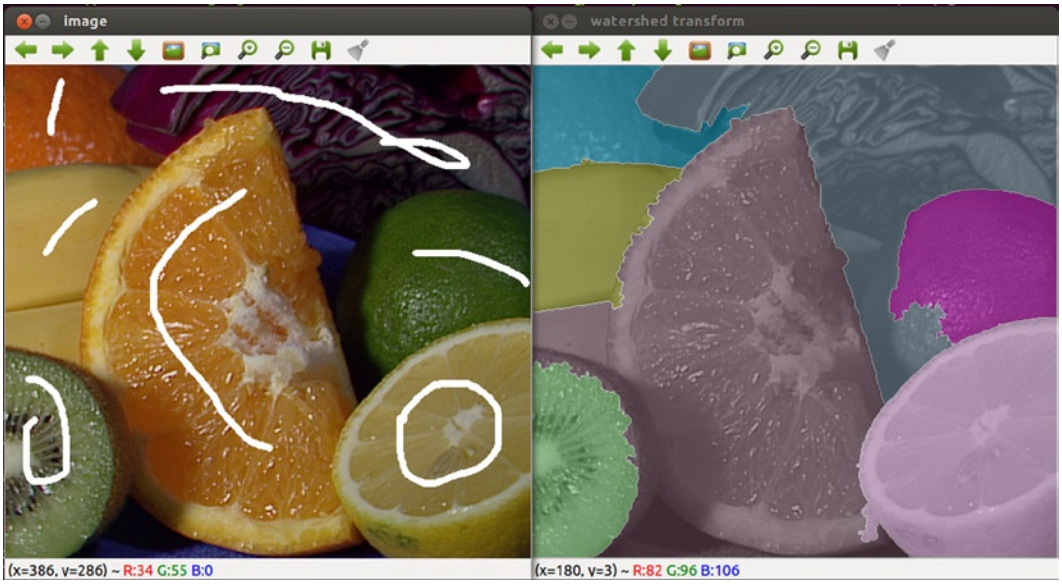


Figure 7-7. *OpenCV watershed demo—marker-based watershed segmentation*

The main challenge in using the watershed transform for an object counter application is figuring out the markers by code rather than having the human input them. The following strategy, which uses a lot of morphological operations that you learnt before, is used in Listing 7-3:

- Equalize the gray level histogram of the image to improve contrast, as shown in Figure 7-8. We will discuss how this works later on in this chapter. For now, just bear in mind that it improves contrast in the image and can be implemented by using `equalizeHist()`

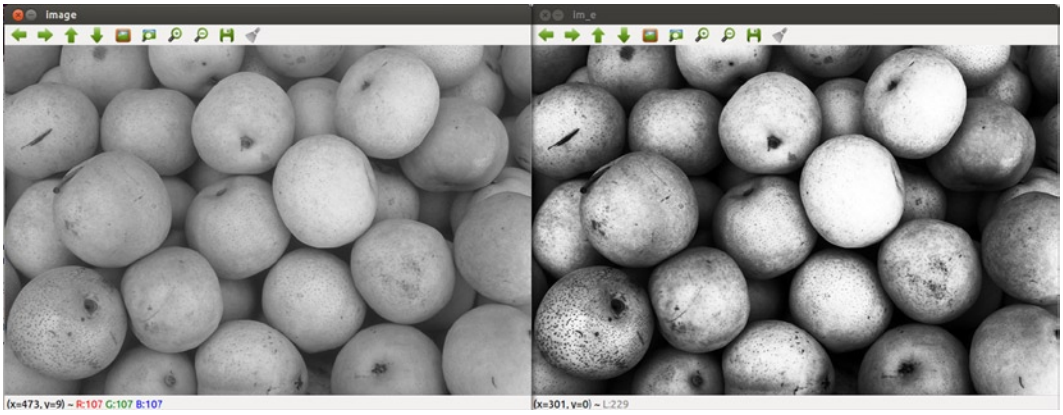


Figure 7-8. *Histogram equalization improves image contrast*

- Dilate the image to remove small black spots (Figure 7-9)

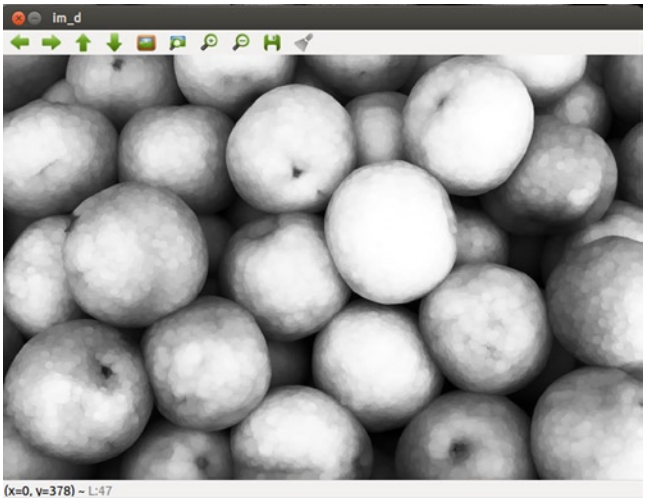


Figure 7-9. *Dilating the image removes black spots*

- Open and close to highlight foreground objects (Figure 7-10)

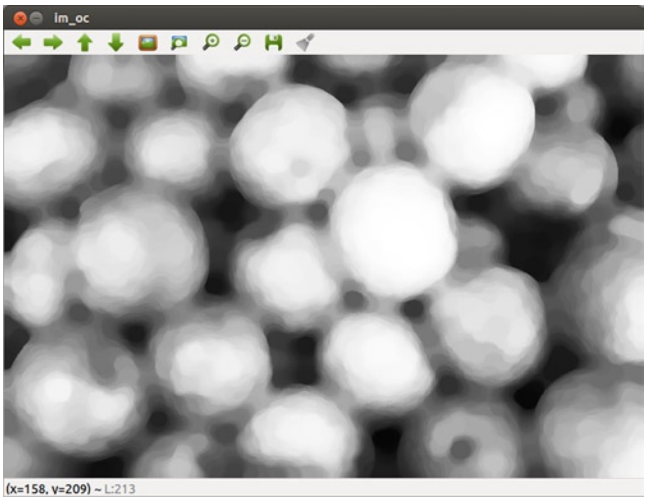


Figure 7-10. *Opening and closing with a relatively large circular structuring element highlights foreground objects*

- Apply adaptive threshold to create a binary mask that is 255 at regions of foreground objects and 0 otherwise (Figure 7-11)

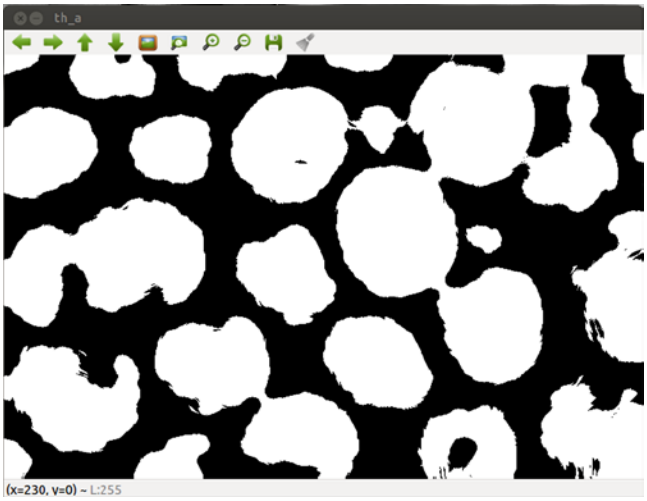


Figure 7-11. *Adaptively thresholding the previous image (almost) isolates the different objects*

- Erode twice to separate some regions in the mask (Figure 7-12)

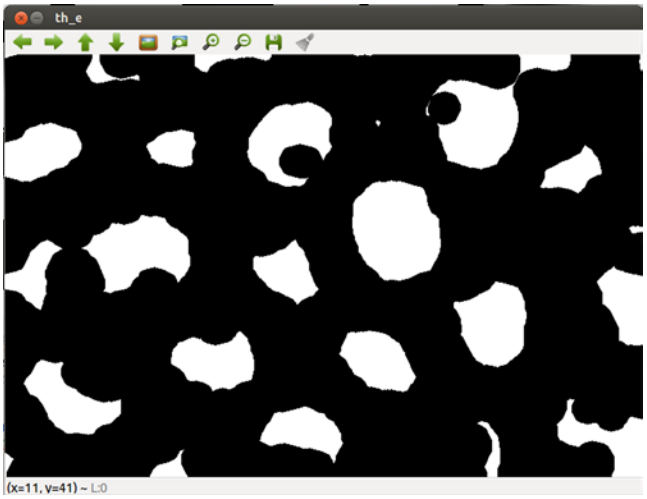


Figure 7-12. *Eroding the previous image twice completely isolates the objects*

The regions in this mask can now we be used as markers in the `watershed()` function. But before that, these regions have to be labeled by positive integers. We do this by detecting contours and then filling these contours by `drawContours()` using successively increasing integers. Figure 7-13 shows how the final output of the watershed transform looks. Note the little neat tricks that I borrowed from the OpenCV watershed demo code to color the regions and transparently display them. Listing 7-3 shows the actual program.

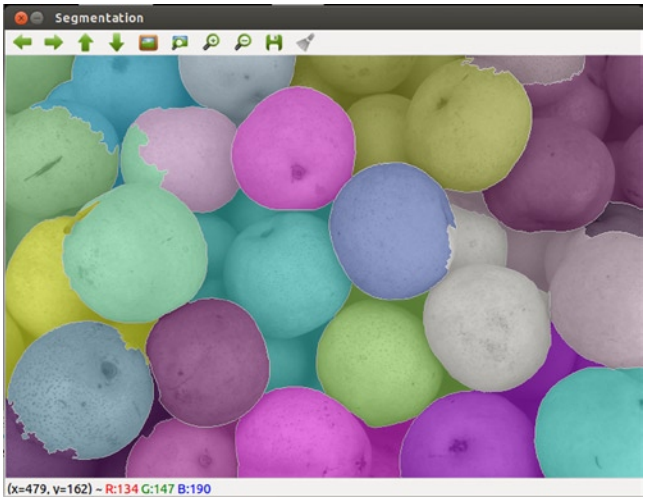


Figure 7-13. *Segmentation after the watershed transform*

Listing 7-3. Program to count the number of objects using morphology operations and the watershed transform

```
// Program to count the number of objects using morphology operations and the watershed transform
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

using namespace cv;
using namespace std;

class objectCounter {
    private:
        Mat image, gray, markers, output;
        int count;
    public:
        objectCounter(Mat);  //constructor
        void get_markers();  //function to get markers for watershed segmentation
        int count_objects(); //function to implement watershed segmentation and count catchment basins
};

objectCounter::objectCounter(Mat _image) {
    image = _image.clone();
    cvtColor(image, gray, CV_BGR2GRAY);
    imshow("image", image);
}

void objectCounter::get_markers() {
    // equalize histogram of image to improve contrast
    Mat im_e; equalizeHist(gray, im_e);
    //imshow("im_e", im_e);
```

```
    // dilate to remove small black spots
    Mat strel = getStructuringElement(MORPH_ELLIPSE, Size(9, 9));
    Mat im_d; dilate(im_e, im_d, strel);
    //imshow("im_d", im_d);

    // open and close to highlight objects
    strel = getStructuringElement(MORPH_ELLIPSE, Size(19, 19));
    Mat im_oc; morphologyEx(im_d, im_oc, MORPH_OPEN, strel);
    morphologyEx(im_oc, im_oc, MORPH_CLOSE, strel);
    //imshow("im_oc", im_oc);

    // adaptive threshold to create binary image
    Mat th_a; adaptiveThreshold(im_oc, th_a, 255, ADAPTIVE_THRESH_MEAN_C, THRESH_BINARY, 105, 0);
    //imshow("th_a", th_a);

    // erode binary image twice to separate regions
    Mat th_e; erode(th_a, th_e, strel, Point(-1, -1), 2);
    //imshow("th_e", th_e);

    vector<vector<Point> > c, contours;
    vector<Vec4i> hierarchy;
    findContours(th_e, c, hierarchy, CV_RETR_CCOMP, CV_CHAIN_APPROX_NONE);

    // remove very small contours
    for(int idx = 0; idx >= 0; idx = hierarchy[idx][0])
        if(contourArea(c[idx]) > 20) contours.push_back(c[idx]);

    cout << "Extracted " << contours.size() << " contours" << endl;

    count = contours.size();
    markers.create(image.rows, image.cols, CV_32SC1);
    for(int idx = 0; idx < contours.size(); idx++)
        drawContours(markers, contours, idx, Scalar::all(idx + 1), -1, 8);
}

int objectCounter::count_objects() {
    watershed(image, markers);

    // colors generated randomly to make the output look pretty
    vector<Vec3b> colorTab;
    for(int i = 0; i < count; i++) {
        int b = theRNG().uniform(0, 255);
        int g = theRNG().uniform(0, 255);
        int r = theRNG().uniform(0, 255);

        colorTab.push_back(Vec3b((uchar)b, (uchar)g, (uchar)r));
    }

    // watershed output image
    Mat wshed(markers.size(), CV_8UC3);
```

```
    // paint the watershed output image
    for(int i = 0; i < markers.rows; i++)
        for(int j = 0; j < markers.cols; j++) {
            int index = markers.at<int>(i, j);
            if(index == -1)
                wshed.at<Vec3b>(i, j) = Vec3b(255, 255, 255);
            else if(index <= 0 || index > count)
                wshed.at<Vec3b>(i, j) = Vec3b(0, 0, 0);
            else
                wshed.at<Vec3b>(i, j) = colorTab[index - 1];
        }

    // superimpose the watershed image with 50% transparence on the grayscale original image
    Mat imgGray; cvtColor(gray, imgGray, CV_GRAY2BGR);
    wshed = wshed*0.5 + imgGray*0.5;
    imshow("Segmentation", wshed);

    return count;
}

int main() {
    Mat im = imread("fruit.jpg");

    objectCounter oc(im);
    oc.get_markers();

    int count = oc.count_objects();

    cout << "Counted " << count << " fruits." << endl;

    while(char(waitKey(1)) != 'q') {}

    return 0;
}
```

GrabCut Segmentation

GrabCut is a graph-based approach to segmentation that allows you to specify a rectangle around an object of interest and then tries to segment the object out of the image. While I will restrict discussion of this algorithm to the OpenCV demo, you can check out the paper on this algorithm, "GrabCut: Interactive foreground extraction using iterated graph cuts" by C. Rother, V. Kolmogorov, and A. Blake. The OpenCV demo allows you to run several iterations of the algorithm, with each iteration giving successively better results. Notice that the initial rectangle that you specify must be as tight as possible.

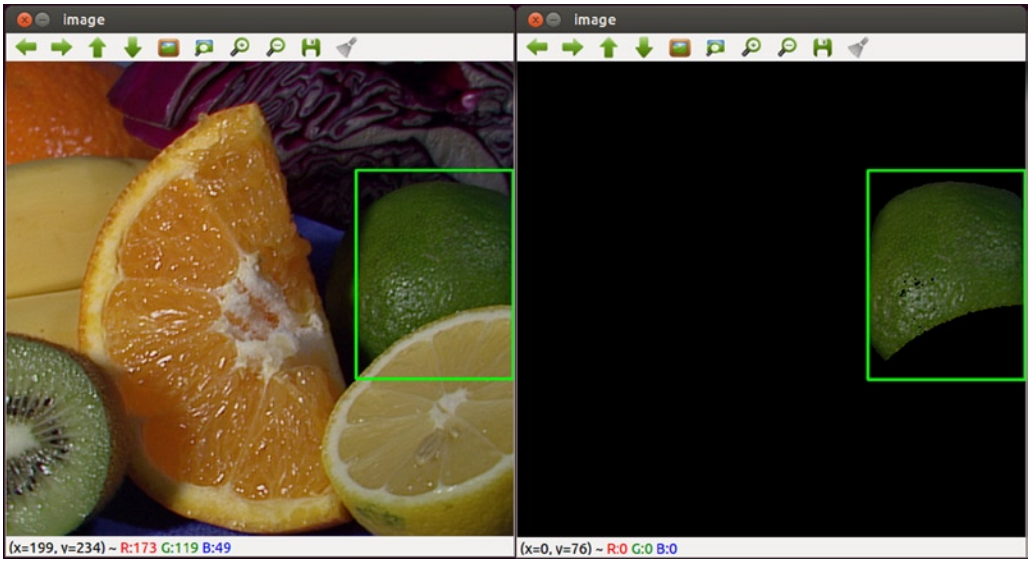


Figure 7-14. *OpenCV GrabCut demo*

Histograms

Histograms are a simple yet powerful way of representing distributions of data. If you do not know what a histogram is, I suggest reading the Wikipedia page. In images, the simplest histograms can be made from the gray levels (or R, G, or B values) of all pixels of the image. This will allow you to spot general trends in the image data distribution. For example, in a dark image, the histogram will have most of its peaks in the lower part of the 0 to 255 range.

Equalizing Histograms

The simplest application of histograms is to normalize the brightness and improve contrast of the image. This is done by first thresholding out very low values from the histogram and then "stretching" it so that the histogram occupies the entire 0 to 255 range. A very efficient way of doing this is:

- Calculate the histogram and normalize it so that the sum of all elements in the histogram is 255
- Calculate the cumulative sum of the histogram:

$$H'(i) = \sum_{j=0}^{j=i} H(j)$$

- Make the new image `dst` from original image `src` as follows:

$$dst(x,y) = H'(src(x,y))$$

The OpenCV function `equalizeHist()` implements this. Listing 7-4 is a small app that can show you the difference between a normal image and its histogram-equalized version. You can apply `equalizeHist()` to RGB images, too, by applying it individually to all the three channels. Figure 7-15 shows the magic of histogram equalization!

Listing 7-4. Program to illustrate histogram equalization

```
// Program to illustrate histogram equalization in RGB images
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

using namespace cv;
using namespace std;

Mat image, image_eq;
int choice = 0;

void on_trackbar(int, void*) {
    if(choice == 0) // normal image
        imshow("Image", image);
    else            // histogram equalized image
        imshow("Image", image_eq);
}

int main() {
    image = imread("scene.jpg");
    image_eq.create(image.rows, image.cols, CV_8UC3);

    //separate channels, equalize histograms and them merge them
    vector<Mat> channels, channels_eq;

    split(image, channels);

    for(int i = 0; i < channels.size(); i++) {
        Mat eq;
        equalizeHist(channels[i], eq);
        channels_eq.push_back(eq);
    }

    merge(channels_eq, image_eq);

    namedWindow("Image");

    createTrackbar("Normal/Eq.", "Image", &choice, 1, on_trackbar);
    on_trackbar(0, 0);

    while(char(waitKey(1)) != 'q') {}

    return 0;
}
```

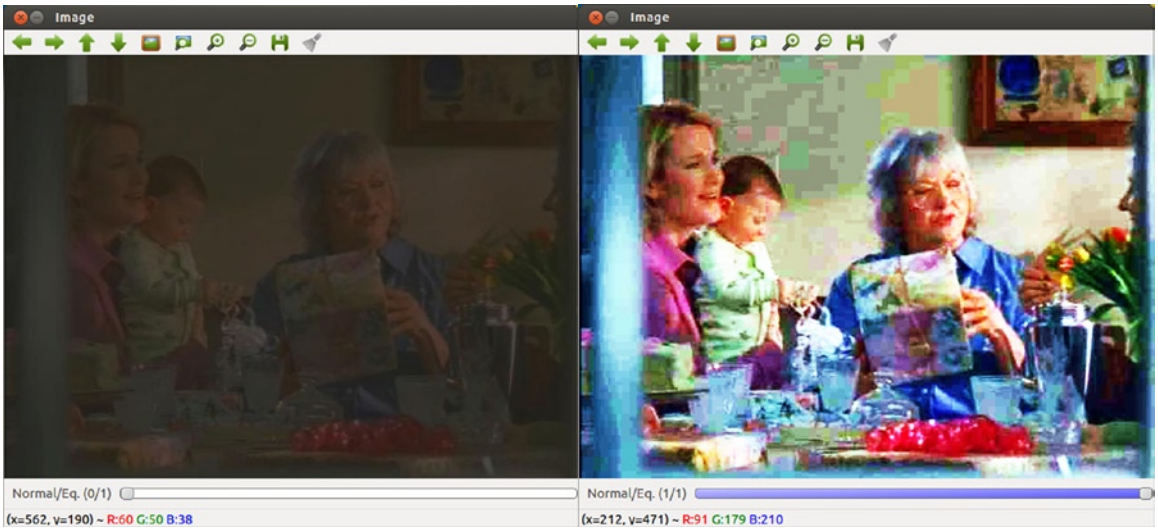


Figure 7-15. *Histogram equalization*

Histogram Backprojections

Histogram backprojection is the reverse process of calculating a histogram. Say you already have a histogram. You can backproject it into another image by replacing every pixels value of that image with the value of the histogram at the bin in which the pixel value falls. This effectively fills regions in the image that match the histogram distribution with high value and the rest with small values. The OpenCV functions `calcHist()` and `calcBackProject()` can be used to calculate histograms and backproject them, respectively. To give you an example of their usage, Listing 7-5 is a modification of the floodFill-automated object detector:

- It allows the user to click at an object in the window
- It calculates connected similar points using `floodFill()`
- It calculates the hue and saturation histogram of the points selected by `floodFill()`
- It backprojects this histogram into successive video frames

Figure 7-16 shows the backprojection application in action.

Listing 7-5. Program to illustrate histogram backprojection

```
// Program to illustrate histogram backprojection
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

using namespace cv;
using namespace std;
```

```
Mat frame_hsv, frame, mask;
MatND hist; //2D histogram
int conn = 4, val = 255, flags = conn + (val << 8) + CV_FLOODFILL_MASK_ONLY;

bool selected = false;

// hue and saturation histogram ranges
float hrange[] = {0, 179}, srange[] = {0, 255};
const float *ranges[] = {hrange, srange};

void on_mouse(int event, int x, int y, int, void *) {
    if(event != EVENT_LBUTTONDOWN) return;

    selected = true;

    // floodFill
    Point p(x, y);
    mask = Scalar::all(0);
    floodFill(frame, mask, p, Scalar(255, 255, 255), 0, Scalar(10, 10, 10), Scalar(10, 10, 10), flags);
    Mat _mask = mask.rowRange(1, mask.rows-1).colRange(1, mask.cols-1);

    // number of bins in the histogram for each channel
    int histSize[] = {50, 50}, channels[] = {0, 1};

    // calculate and normalize histogram
    calcHist(&frame_hsv, 1, channels, _mask, hist, 2, histSize, ranges);
    normalize(hist, hist, 0, 255, NORM_MINMAX, -1, Mat());
}

int main() {
    // Create a VideoCapture object to read from video file
    // 0 is the ID of the built-in laptop camera, change if you want to use other camera
    VideoCapture cap(0);

    //check if the file was opened properly
    if(!cap.isOpened()) {
        cout << "Capture could not be opened succesfully" << endl;
        return -1;
    }

    namedWindow("Video");
    namedWindow("Backprojection");

    setMouseCallback("Video", on_mouse);

    while(char(waitKey(1)) != 'q' && cap.isOpened()) {
        cap >> frame;
        if(!selected) mask.create(frame.rows+2, frame.cols+2, CV_8UC1);
        // Check if the video is over
        if(frame.empty()) {
            cout << "Video over" << endl;
            break;
        }
        cvtColor(frame, frame_hsv, CV_BGR2HSV);
```

```
        // backproject on the HSV image
        Mat frame_backprojected = Mat::zeros(frame.size(), CV_8UC1);
        if(selected) {
            int channels[] = {0, 1};
            calcBackProject(&frame_hsv, 1, channels, hist, frame_backprojected, ranges);
        }

        imshow("Video", frame);
        imshow("Backprojection", frame_backprojected);
    }

    return 0;
}
```

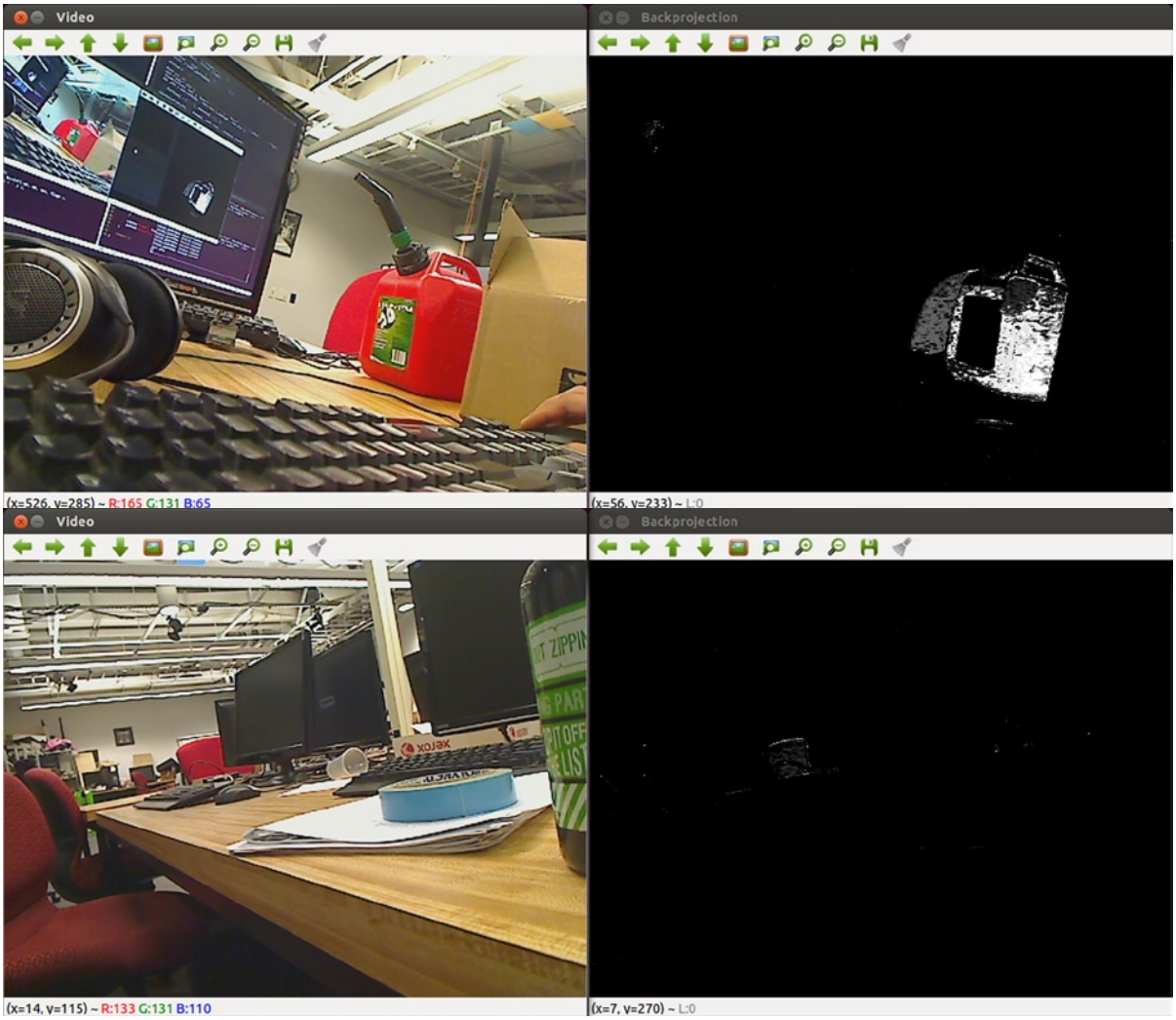


Figure 7-16. *Histogram backprojection*

Meanshift and Camshift

Meanshift is an algorithm that finds an object in a backprojected histogram image. It takes an initial search window as input and then iteratively shifts this search window such that the mass center of the backprojection inside this window is at the center of the window.

Camshift is a histogram backprojection–based object tracking algorithm that uses meanshift at its heart. It takes the detection window output by meanshift and figures out the best size and rotation of that window to track the object. OpenCV functions `meanShift()` and `CamShift()` implement these algorithms and you can see the OpenCV implementation of camshift in the built-in demo.

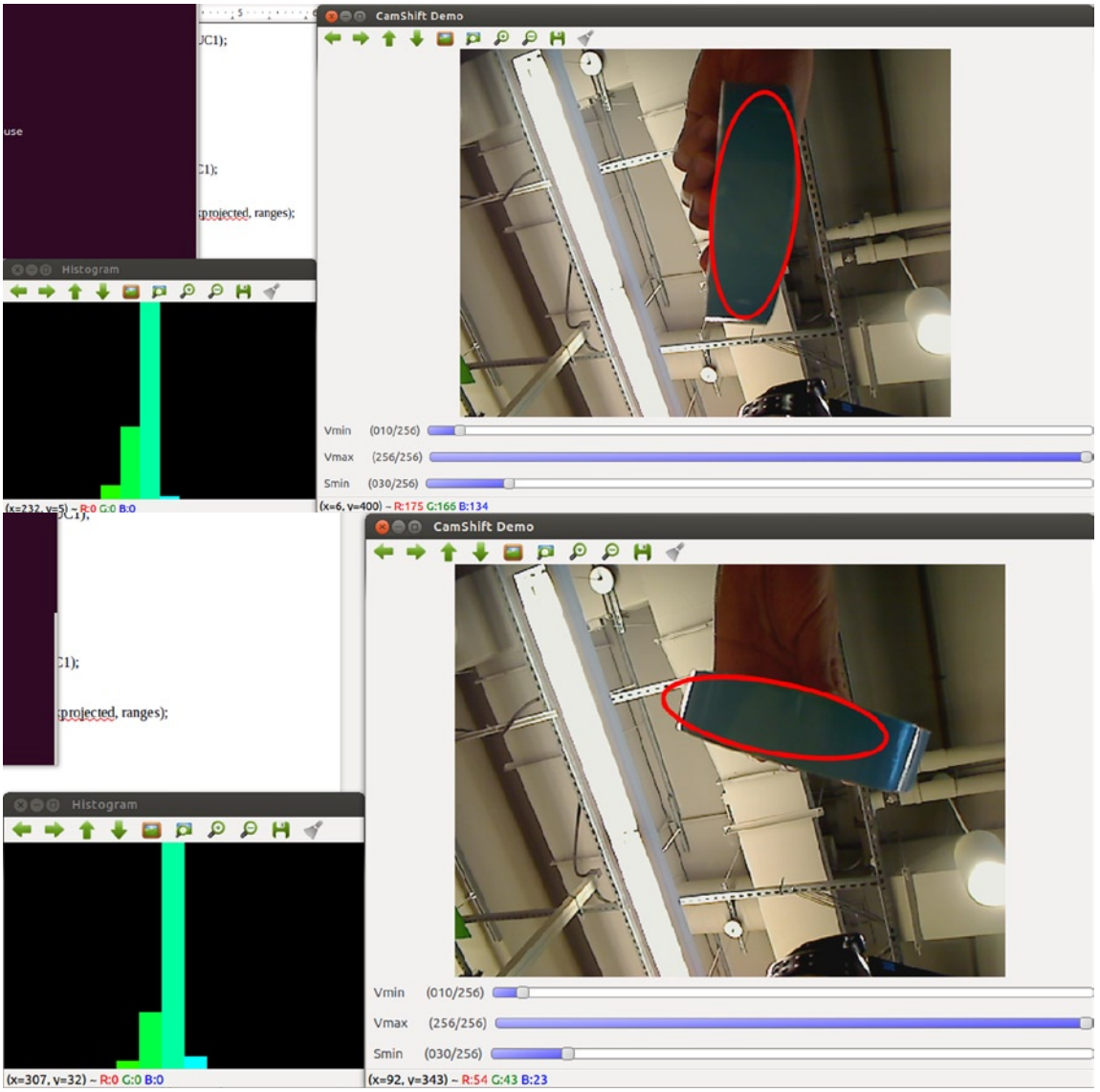


Figure 7-17. *OpenCV camshift demo*

Summary

This chapter will have made you aware of the segmentation algorithms that OpenCV provides. But what I really want you to take away from this chapter is a sense of how you can combine your image-processing knowledge to put together your own segmentation algorithm, as we did for the fruit counter application. This is because, as I said in the beginning of the chapter, segmentation is differently defined for different problems, and it requires you to be creative. Histogram equalization and backprojection can be important preprocessing steps for some higher-level algorithms, so don't forget them, too!

CHAPTER 8

Basic Machine Learning and Object Detection Based on Keypoints

In this exciting chapter I plan to discuss the following:

- Some general terminology including definitions of keypoints and keypoint descriptors
- Some state-of-the-art keypoint extraction and description algorithms that OpenCV boasts, including SIFT, SURF, FAST, BRIEF, and ORB
- Basic machine learning concepts and the working of the Support Vector Machine (SVM)
- We will use our knowledge to make a keypoint-based object detector app that uses machine learning to detect multiple objects in real time using the bag-of-visual-words framework. This app will be robust to illumination invariance and clutter in the scene

Keypoints and Keypoint Descriptors: Introduction and Terminology

The color-based object detector app we have been working on until now leaves a lot to be desired. There are two obvious pitfalls of detecting objects based on color (or grayscale intensity):

- Works well only for single-colored objects. You can backproject a hue histogram of a textured object, but that is likely to include a lot of colors and this will cause a very high amount of false positives
- Can be fooled by different objects of the same color

But color-based object detection is very fast. This makes it ideal for applications where the environment is strictly controlled, but almost useless in unknown environments. I will give you an example from my work for the Robocup humanoid soccer team of the University of Pennsylvania. The competition is about making a team of three humanoid robots play 100% autonomous soccer against other similar robot teams. These robots require fast ball, field, field line, and goalpost detection in order to play effectively, and since the rules mandate all these objects to be of a specific unique solid color, we use color-based object detection verified by some logical checks (like ball height and goalpost height-to-width aspect ratio). The color-based object detection works nicely here, because the environment is controlled and the objects are all unique solid colors. But if you want to design the vision system for a search-and-rescue robot, you should obviously not rely on color for detecting objects, because you do not know what the working environment of your robot will look like, and the objects of your interest can consist of arbitrarily many colors. With this motivation, let us learn what keypoints and keypoint descriptors mean!

General Terms

In object detection problems you usually have two sets of images. The training set will be used for showing the computer what the desired object looks like. This can be done by calculating hue histograms (as you already know) or by computing keypoints and keypoint descriptors (as you will learn soon). Obviously, it is preferable that training images are either annotated (locations of objects of interest are specified by bounding boxes) or just contain the objects of interest and little else. The testing set contains images on which your algorithm will be used—in short, the use case of your application.

How Does the Keypoint-Based Method Work?

The philosophy of this method is that one should not make the computer "learn" the characteristics of the whole object template (like calculating the histogram of flood-filled points) and look for similar instances in other images. Instead, one should find certain "important" points (keypoints) in the object template and store information about the neighborhood of those keypoints (keypoint descriptors) as a description of the object. In other test images, one should find keypoint descriptors of keypoints in the whole image and try to 'match' the two descriptor sets (one from the object template and one from the test image) using some notion of similarity, and see how many descriptors match. For test images that contain an instance of the object in the object template, you will get a lot of matches and these matches will have a regular trend. Figure 8-1 will help you understand this better. It shows a training and testing image, each with its SIFT (Scale Invariant Feature Transform—a famous algorithm for keypoint extraction and description) keypoints, and the matching descriptors between the two images. See how all of the matches follow a trend.

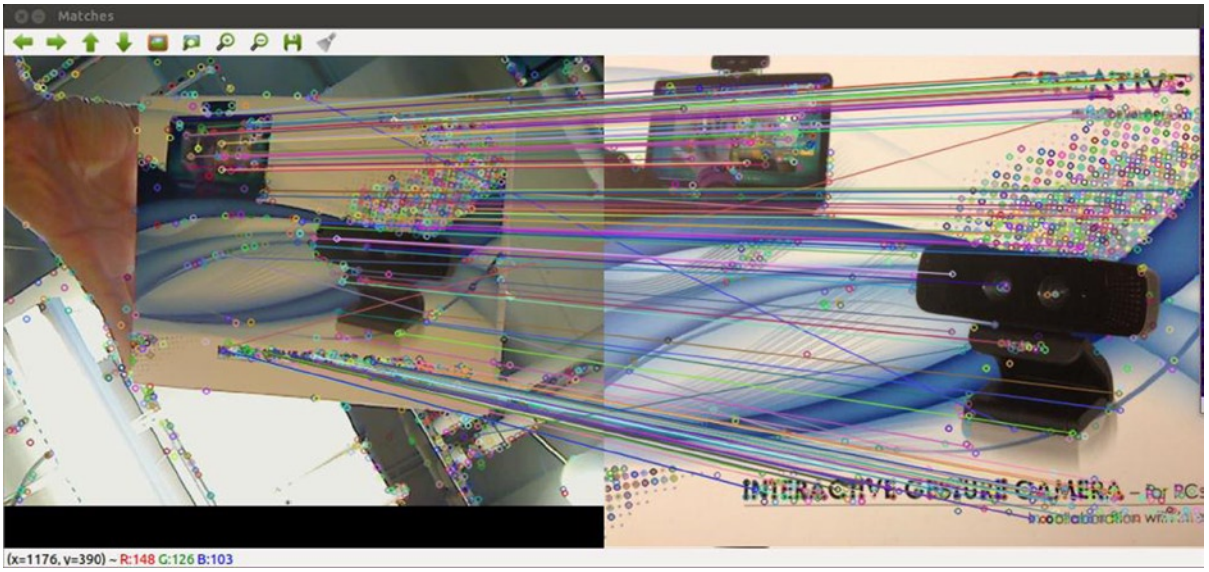


Figure 8-1. *SIFT keypoint and feature matching*

Looks cool? The next section describes how SIFT works.

SIFT Keypoints and Descriptors

SIFT is arguably the most famous and widely implemented keypoint detection and description algorithm out there today. That is why I chose to introduce it first, hoping to introduce some other concepts associated with keypoint detection and description along the way. You can get a very detailed description of the algorithm in the classic paper "Distinctive Image Features from Scale-Invariant Keypoints" by David G. Lowe.[1] For all the other algorithms, I will limit discussion to salient points in the method and provide reference to the papers on which they are based.

Keypoint descriptors are also often called features. Object detection using SIFT is scale and rotation invariant. This means the algorithm will detect objects that:

- Have the same appearance but a bigger or smaller size in the test image compared with the training image(s) (scale invariance)
- Are rotated (about a scale perpendicular to the image) compared to the training object(s)
- Exhibit a combination of these two conditions (rotation invariance)

This is because the keypoints extracted by SIFT have a scale and orientation associated with them. The term "scale" requires some explanation, as it is widely used in computer vision literature. Scale refers to the notion of the size at which the object is seen, or how far the camera is from the object. Scale is also almost always relative. A higher scale means the object looks smaller (because the camera moves further away) and vice versa. Higher scale is usually achieved by smoothing the image using a Gaussian kernel (remember `gaussianBlur()`?) and downsampling. Lower scale is achieved by a similar process of smoothing and upsampling. That is why scale is also often specified by the variance of the Gaussian kernel used to achieve it (with $\sigma = 1$ for the current scale).

Keypoint Detection and Orientation Estimation

Keypoints can be something as simple as corners. However, because SIFT requires keypoints to have a scale and an orientation associated with them, and also for some efficiency reasons that will be clear once you know how keypoint descriptors are constructed, SIFT uses the following method to compute keypoint locations in the image:

- It convolves the image with Gaussians of successively increasing variance, thereby increasing the scale (and downsampling by a factor of 2 every time the scale is doubled; i.e., the variance of the Gaussian is increased by a factor of 4). Thus, a "scale pyramid" is created as shown in Figure 8-2. Remember that a 2-D Gaussian with a variance σ^2 is given by:

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2}$$

[1] http://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf

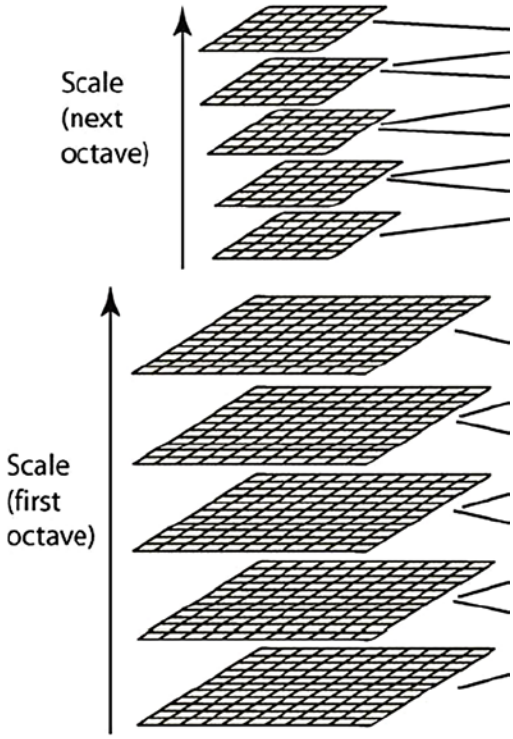


Figure 8-2. *Scale Pyramid (From "Distinctive Image Features from Scale-Invariant Keypoints," David G. Lowe. Reprinted by permission of Springer Science+Business Media.)*

- Successive images in this pyramid are subtracted from each other. The resulting images are said to be the output of the Difference of Gaussians (DoG) operator on the original image, as shown in Figure 8-3.

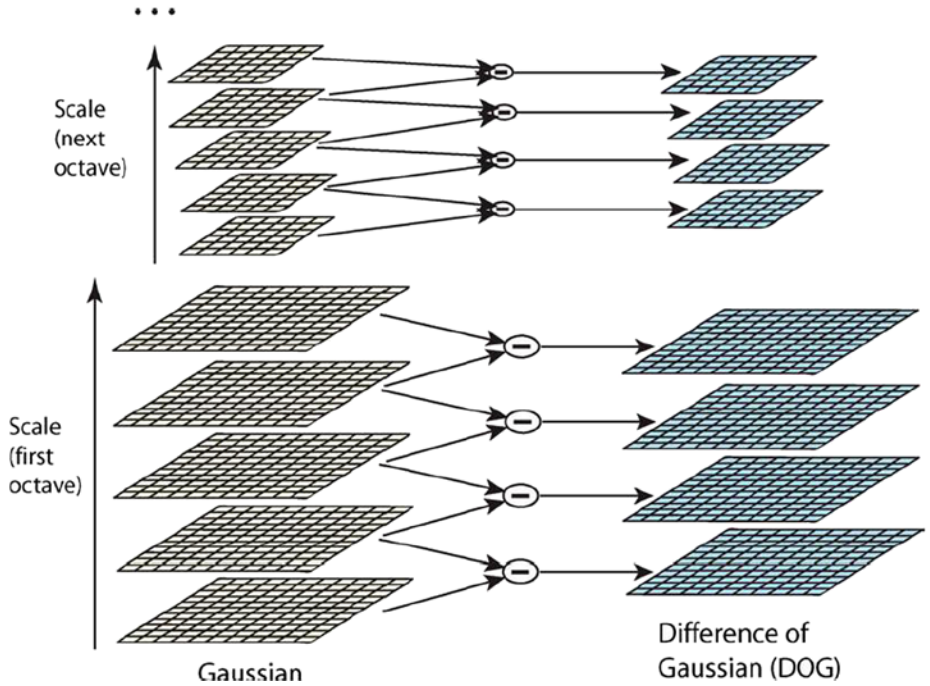


Figure 8-3. *Difference of Gaussians Pyramid (From "Distinctive Image Features from Scale-Invariant Keypoints," David G. Lowe. Reprinted by permission of Springer Science+Business Media.)*

- Lowe in his paper has shown mathematically (through the solution of the heat diffusion equation) that the output of applying the DoG operator is directly proportional to a close approximation of the output of the Laplacian of Gaussian operator applied to the image. The point of going through this whole process is that it has been shown in the literature that the maxima and minima of the output of the Laplacian of Gaussians operator applied to the image are more stable keypoints than those obtained by conventional gradient-based methods. Application of the Laplacian of Gaussian operator to the image involves second order differentiation after convolution with Gaussian kernels, whereas here we get a DoG pyramid (which is a close approximation to the LoG pyramid) using just image differences after applying Gaussian kernels. Maxima and minima of the DoG pyramid are obtained by comparing a pixel to its 26 neighbors in the pyramid as shown in Figure 8-4. A point is selected as a keypoint only if it higher or lower than all its 26 neighbors. The cost of this check is reasonably low because most points will be eliminated in the first few checks. The scale of the keypoint is the square rooted variance of the smaller of the two Gaussians which were used to produce that particular level of the DoG pyramid.

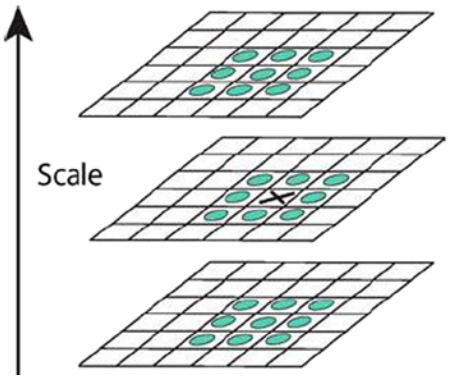


Figure 8-4. *Maxima and minima selection by comparison with 26 neighbors (From "Distinctive Image Features from Scale-Invariant Keypoints," David G. Lowe. Reprinted by permission of Springer Science+Business Media.)*

- Keypoint locations (with scales) thus computed are further filtered by checking that they have sufficient contrast and that they are not a part of an edge by using some ingenious math (which you can appreciate if you read section 4 of the paper)
- For assigning an orientation to the keypoints, their scale is used to select the Gaussian smoothed image with the closest scale (all of which we have already computed and stored for constructing the DoG pyramid) so that all calculations are performed in a scale-invariant manner. A square region around the keypoint corresponding to a circle with a radius of 1.5 times the scale is selected in the image (Figure 8-5), and gradient orientation at every point in this region is computed by the following equation (L is the Gaussian smoothed image)

$$\theta(x,y) = tan^{-1}\left(\frac{L(x,y+1) - L(x,y-1)}{L(x+1,y) - L(x-1,y)}\right)$$

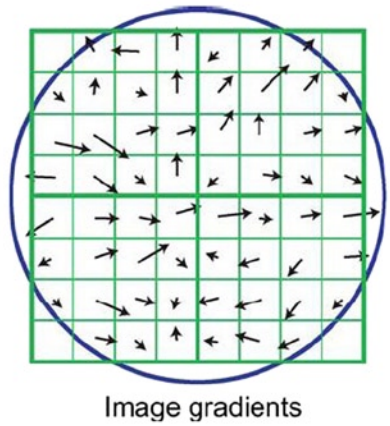


Figure 8-5. *Square patch around a keypoint location and gradient orientations with gradient magnitudes (From "Distinctive Image Features from Scale-Invariant Keypoints," David G. Lowe. Reprinted by permission of Springer Science+Business Media.)*

An orientation histogram with 36 10-degree bins is used to collect these orientations. Contributions of gradient orientations to the histogram are weighted according to a Gaussian kernel of σ equal to 1.5 times the scale of the keypoint. Peaks in the histogram correspond to the orientation of the keypoints. If there are multiple peaks (values within 80% of the highest peak), multiple keypoints are created at the same location with multiple orientations.

SIFT Keypoint Descriptors

Every SIFT keypoint has a 128-element descriptor associated with it. The square region shown in Figure 8-5 is divided into 16 equal blocks (the figure just shows four blocks). For each block, gradient orientations are histogrammed into eight bins, with the contributions equal to the product of the gradient magnitudes and a Gaussian kernel with σ equal to half the width of the square. In order to achieve rotation invariance, the coordinates of points in the square region and the gradient orientations at those points are rotated relative to the orientation of the keypoint. Figure 8-6 shows four such eight-bin histograms (with length of the arrow indicating the value of the corresponding bin) for the four blocks. Thus, 16 eight-bin histograms make the 128-element SIFT descriptor. Finally, the 128-element vector is normalized to unit length to provide illumination invariance.

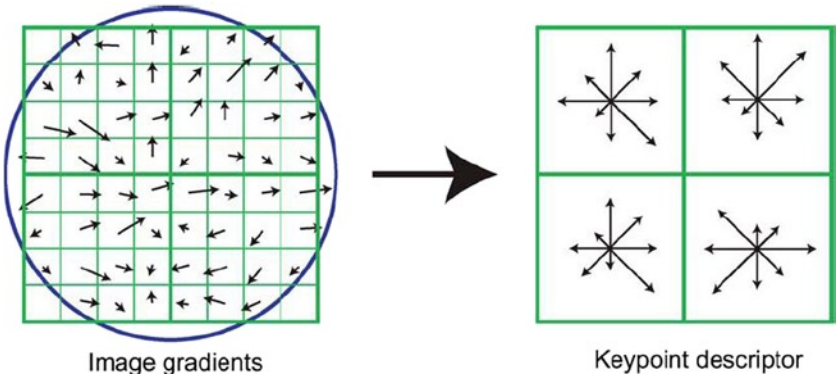


Figure 8-6. *Gradient orientation histograms for keypoint description in SIFT (From "Distinctive Image Features from Scale-Invariant Keypoints," David G. Lowe. Reprinted by permission of Springer Science+Business Media.)*

Matching SIFT Descriptors

Two 128-element SIFT descriptors are considered to match if the Euclidean (a.k.a. L2) distance between them is low. Lowe also came up with a condition for filtering these matches to make the matching more robust. Suppose you have two sets of descriptors, one from the training image and the other from a test image, and you want to find robust matches. Usually, special algorithms called "nearest neighbor searches" are used to find out the test descriptor closest in the Euclidean sense to each train descriptor. Suppose that you used the nearest neighbor search to get 2 nearest neighbors to all the train descriptor. A descriptor match is eliminated if the distance from the train descriptor to its 1st nearest neighbor is greater than a threshold times the distance to the 2nd nearest neighbor. This threshold obviously is less than 1 and is usually set to 0.8. This forces the matches to be unique and highly "discriminative." This requirement can be made more stringent by reducing the value of this threshold.

OpenCV has a rich set of functions (that are implemented as classes) for all sorts of keypoint detection, descriptor extraction and descriptor matching. The advantage is that you can extract keypoints using one algorithm and extract descriptors around those keypoints using another algorithm, potentially tailoring the process to your needs.

`FeatureDetector` is the base class from which all specific keypoint detection classes (such as SIFT, SURF, ORB, etc.) are inherited. The `detect()` method in this parent class takes in an image and detects keypoints, returning them in a STL vector of KeyPoint objects. `KeyPoint` is a generic class that has been designed to store keypoints (with location, orientation,

scale, filter response and some other information). `SiftFeatureDetector` is the class inherited from `FeatureDetector` that extracts keypoints from a grayscale image using the algorithm I outlined earlier.

Similar to `FeatureDetector`, `DescriptorExtractor` is the base class for all kinds of descriptor extractors implemented in OpenCV. It has a method called `compute()` that takes in the image and keypoints as input and gives a Mat in which every row is the descriptor for the corresponding keypoint. For SIFT, you can use `SiftDescriptorExtractor`.

All descriptor matching algorithms are inherited from the `DescriptorMatcher` class, which has some really useful methods. The typical flow is to put the train descriptors in the matcher object with the `add()` method, and initialize the nearest neighbor search data structures with the `train()` method. Then, you can find closest matches or nearest neighbors for query test descriptors using the `match()` or `knnMatch()` method respectively. There are two ways to get nearest neighbors: doing a brute force search (go through all train points for each test point, `BFmatcher()`) or using the OpenCV wrapper for Fast Library for Approximate Nearest Neighbors (FLANN) implemented in the `FlannBasedMatcher()` class. Listing 8-1 is very simple: it uses SIFT keypoints, SIFT descriptors and gets nearest neighbors using the brute force approach. Be sure to go through it carefully for syntax details. Figure 8-7 shows our first keypoint-based object detector in action!

Listing 8-1. Program to illustrate SIFT keypoint and descriptor extraction, and matching using brute force

```
// Program to illustrate SIFT keypoint and descriptor extraction, and matching using brute force
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/nonfree/features2d.hpp>
#include <opencv2/features2d/features2d.hpp>

using namespace cv;
using namespace std;

int main() {
    Mat train = imread("template.jpg"), train_g;
    cvtColor(train, train_g, CV_BGR2GRAY);

    //detect SIFT keypoints and extract descriptors in the train image
    vector<KeyPoint> train_kp;
    Mat train_desc;

    SiftFeatureDetector featureDetector;
    featureDetector.detect(train_g, train_kp);
    SiftDescriptorExtractor featureExtractor;
    featureExtractor.compute(train_g, train_kp, train_desc);

    // Brute Force based descriptor matcher object
    BFMatcher matcher;
    vector<Mat> train_desc_collection(1, train_desc);
    matcher.add(train_desc_collection);
    matcher.train();

    // VideoCapture object
    VideoCapture cap(0);
```

```
    unsigned int frame_count = 0;

    while(char(waitKey(1)) != 'q') {
        double t0 = getTickCount();
        Mat test, test_g;
        cap >> test;
        if(test.empty())
            continue;

        cvtColor(test, test_g, CV_BGR2GRAY);

        //detect SIFT keypoints and extract descriptors in the test image
        vector<KeyPoint> test_kp;
        Mat test_desc;
        featureDetector.detect(test_g, test_kp);
        featureExtractor.compute(test_g, test_kp, test_desc);

        // match train and test descriptors, getting 2 nearest neighbors for all test descriptors
        vector<vector<DMatch> > matches;
        matcher.knnMatch(test_desc, matches, 2);

        // filter for good matches according to Lowe's algorithm
        vector<DMatch> good_matches;
        for(int i = 0; i < matches.size(); i++) {
            if(matches[i][0].distance < 0.6 * matches[i][1].distance)
                good_matches.push_back(matches[i][0]);
        }

        Mat img_show;
        drawMatches(test, test_kp, train, train_kp, good_matches, img_show);
        imshow("Matches", img_show);

        cout << "Frame rate = " << getTickFrequency() / (getTickCount() - t0) << endl;
    }

    return 0;
}
```

Figure 8-7. *SIFT keypoint based object detector using the Brute Force matcher*

Note the use of the very convenient `drawMatches()` function to visualize the detection and the method I use to measure the frame rate. You can see that we get a frame rate of around 2.1 fps when there is no instance of the object and 1.7 when there is an instance, which is not that good if you have a real-time application in mind.

Listing 8-2 and Figure 8-8 show that the use of FLANN based matcher increases the frame rate to 2.2 fps and 1.8 fps.

Listing 8-2. Program to illustrate SIFT keypoint and descriptor extraction, and matching using FLANN

```
// Program to illustrate SIFT keypoint and descriptor extraction, and matching using FLANN
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/nonfree/features2d.hpp>
#include <opencv2/features2d/features2d.hpp>

using namespace cv;
using namespace std;

int main() {
    Mat train = imread("template.jpg"), train_g;
    cvtColor(train, train_g, CV_BGR2GRAY);

    //detect SIFT keypoints and extract descriptors in the train image
    vector<KeyPoint> train_kp;
    Mat train_desc;

    SiftFeatureDetector featureDetector;
    featureDetector.detect(train_g, train_kp);
    SiftDescriptorExtractor featureExtractor;
    featureExtractor.compute(train_g, train_kp, train_desc);

    // FLANN based descriptor matcher object
    FlannBasedMatcher matcher;
    vector<Mat> train_desc_collection(1, train_desc);
    matcher.add(train_desc_collection);
    matcher.train();

    // VideoCapture object
    VideoCapture cap(0);

    unsigned int frame_count = 0;

    while(char(waitKey(1)) != 'q') {
        double t0 = getTickCount();
        Mat test, test_g;
        cap >> test;
        if(test.empty())
            continue;

        cvtColor(test, test_g, CV_BGR2GRAY);

        //detect SIFT keypoints and extract descriptors in the test image
        vector<KeyPoint> test_kp;
        Mat test_desc;
        featureDetector.detect(test_g, test_kp);
        featureExtractor.compute(test_g, test_kp, test_desc);
```

```
        // match train and test descriptors, getting 2 nearest neighbors for all test descriptors
        vector<vector<DMatch> > matches;
        matcher.knnMatch(test_desc, matches, 2);

        // filter for good matches according to Lowe's algorithm
        vector<DMatch> good_matches;
        for(int i = 0; i < matches.size(); i++) {
            if(matches[i][0].distance < 0.6 * matches[i][1].distance)
                good_matches.push_back(matches[i][0]);
        }

        Mat img_show;
        drawMatches(test, test_kp, train, train_kp, good_matches, img_show);
        imshow("Matches", img_show);

        cout << "Frame rate = " << getTickFrequency() / (getTickCount() - t0) << endl;
    }

    return 0;
}
```

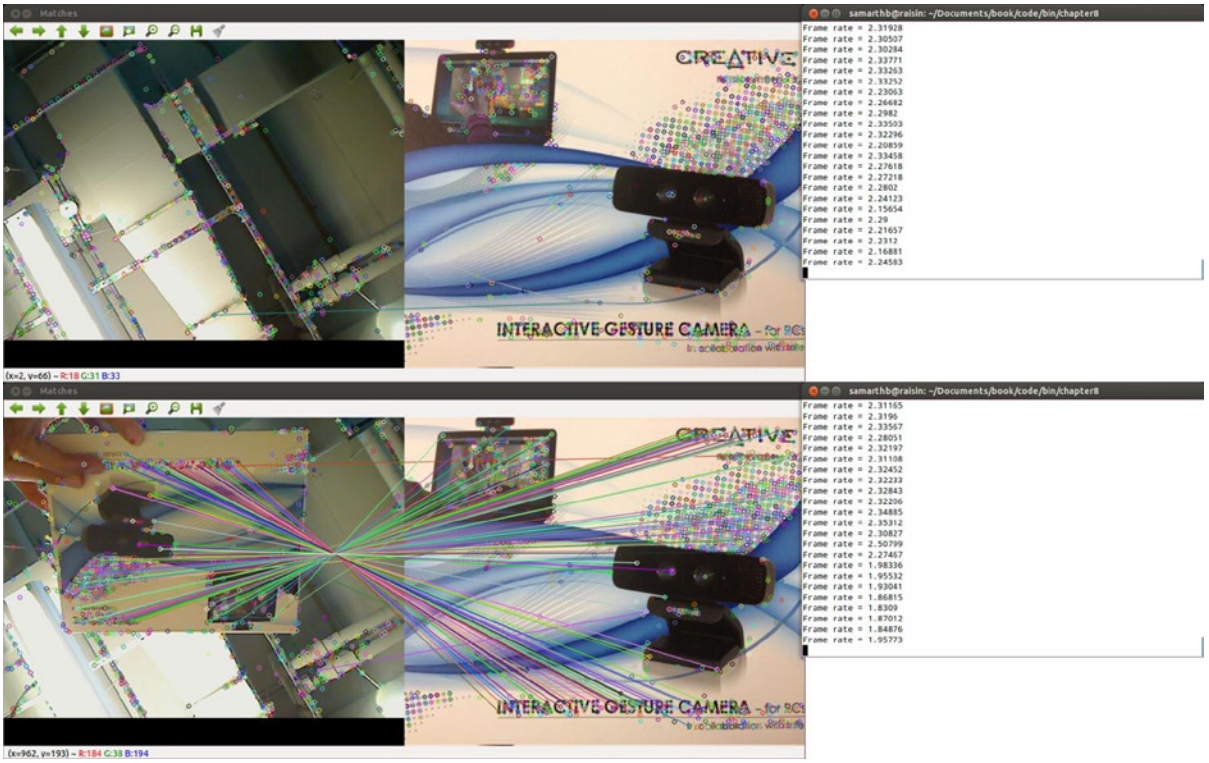


Figure 8-8. *SIFT keypoint based object detector using the FLANN based matcher*

SURF Keypoints and Descriptors

Herbert Bay et al., in their paper "SURF: Speeded Up Robust Features," presented a keypoint detection and description algorithm that is as robust and repeatable as SIFT, yet more than twice as fast. Here I will outline the salient points of this algorithm. (Please refer to the paper if you want more detailed information.)

SURF Keypoint Detection

The reason why SURF is fast is because it uses rectangular discretized integer approximations for complicated continuous real-valued functions. SURF uses maxima of the Hessian matrix determinant. Now the Hessian matrix is defined as:

$$H(x,\sigma) = \begin{bmatrix} Lxx(x,\sigma) & Lxy(x,\sigma) \\ Lxy(x,\sigma) & Lyy(x,\sigma) \end{bmatrix}$$

Where:
$Lxx(x,\sigma)$ is the convolution of the Gaussian second-order derivative $\frac{\partial^2}{\partial x^2} g(\sigma)$ with the image `I` at point `x`.

Figure 8-9 shows how the second-order Gaussian derivatives look and their SURF rectangular integer approximations.

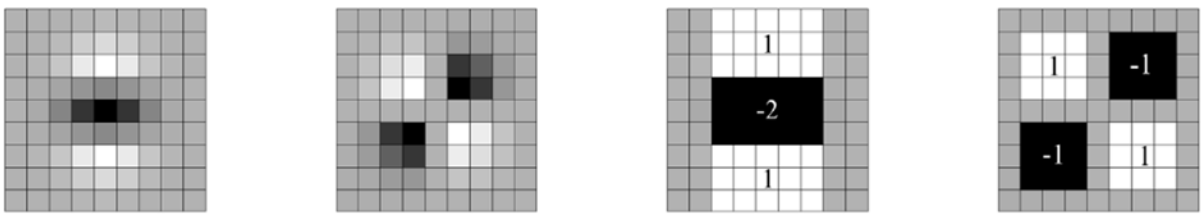


Figure 8-9. *Second order Gaussian derivatives in the Y and XY direction (left); their box-filter approximations (right). (From "SURF: Speeded Up Robust Features," by Herbert Bay et al. Reprinted by permission of Springer Science+Business Media)*

To understand why Hessian matrix determinant extrema represent corner-like keypoints, remember that convolution is associative. So the elements of the Hessian matrix can also be thought of as second-order spatial derivatives of the image after smoothing it with a Gaussian filter. Second-order spatial derivatives of the image will have a peak if there are sudden intensity changes. However, edges can also count as sudden intensity changes. The determinant of the matrix helps us to distinguish edges from corners. Now, the determinant of the Hessian matrix is given by:

$$\det(H) = Lxx(x,\sigma) * Lyy(x,\sigma) - Lxx(x,\sigma)^2$$

From the construction of the filters it is clear that Lxx and Lyy respond to vertical and horizontal edges respectively, whereas Lxy responds most favorably to corners formed by diagonal edges. The determinant therefore will have a high value when there is an intersecting pair of horizontal and vertical edges (making a corner), or when there is a corner made from diagonal edges. This is what we want.

The rectangular-looking approximations that SURF uses are also called box filters. The use of box filters enables the use of integral images, which is an ingenious construction to speed up the convolution operation by orders of magnitude.

The concept of integral images and its use in speeding up convolution with box filters merits discussion, as it is widely used in image processing. So let us take a brief detour.

For any image, its integral image at a pixel is its cumulative sum until that pixel, starting from the origin (top-left corner). Mathematically, if `I` is an image and `H` is the integral image, the pixel `(x, y)` of `H` is given by:

$$H(x,y) = \sum_{i<x, j<y} I(i,j)$$

The integral image can be computed from the original image in linear time using the recursive equation:

$$H(x+1, y+1) = I(x+1, y+1) + H(x, y+1) + H(x+1, y) - H(x, y)$$

One of the most interesting properties of the integral image is that, once you have it, you can use it to get the sum of a box of any size of pixels in the original image using just 4 operations, using this equation (see Figure 8-10 also):

$$Sum\ of\ shaded\ region = H(i,j) - H(i-w, j) - H(i, j-h) + H(i-w, j-h)$$

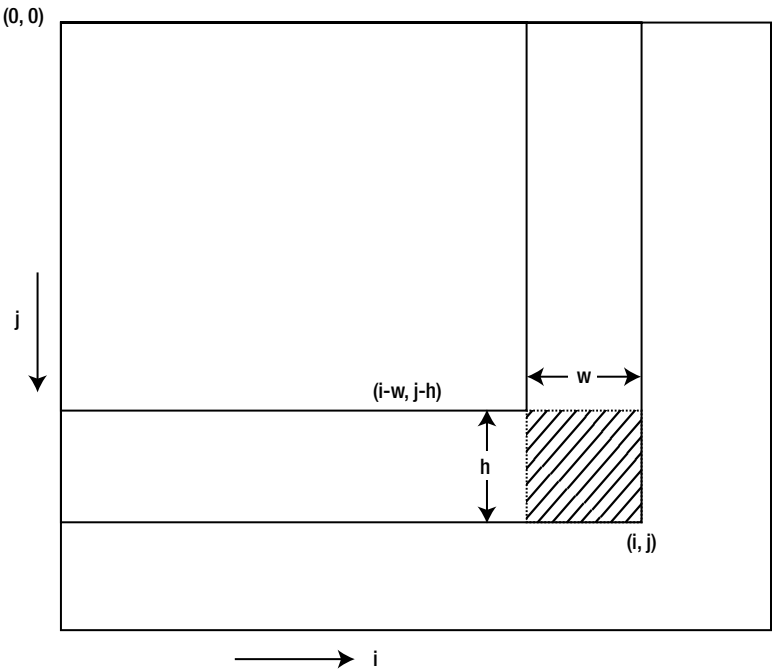


Figure 8-10. *Using integral image to sum up sum across a rectangular region*

Convolution with a kernel is just element-wise multiplication of the pixel values with the kernel elements and then summing up. Now if the kernel elements are constant, things become a lot simpler. We can just sum up the pixel value under the kernel and multiply the sum by the constant kernel value. Now do you see why box filters (kernels with constant elements in rectangular regions) and integral images (images that help you sum up pixel values in a rectangular region very fast) make such a great pair?

To construct the scale-space pyramid, SURF increases the size of the Gaussian filter rather than reducing the size of the image. After constructing that it finds the extrema of the Hessian matrix determinant values at different scales by comparing a point with its 26 neighbors in the pyramid just like SIFT. This gives the SURF keypoints with their scales.

Keypoint orientation is decided by selecting a circular neighborhood of radius 6 times the scale of the keypoint around a keypoint. At every point in this neighborhood, responses to horizontal and vertical box filters (called Haar wavelets, shown in Figure 8-11) are recorded.

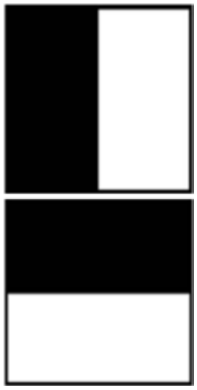


Figure 8-11. *Vertical (top) and horizontal (bottom) box filters used for orientation assignment*

Once responses are weighted with a Gaussian ($\sigma = 2.5$ times scale), they are represented as vectors in a space with horizontal response strength along the *x*-axis and vertical response strength along the *y*-axis. A sliding arc with angle of 60 degrees sweeps a rotation through this space (Figure 8-12).

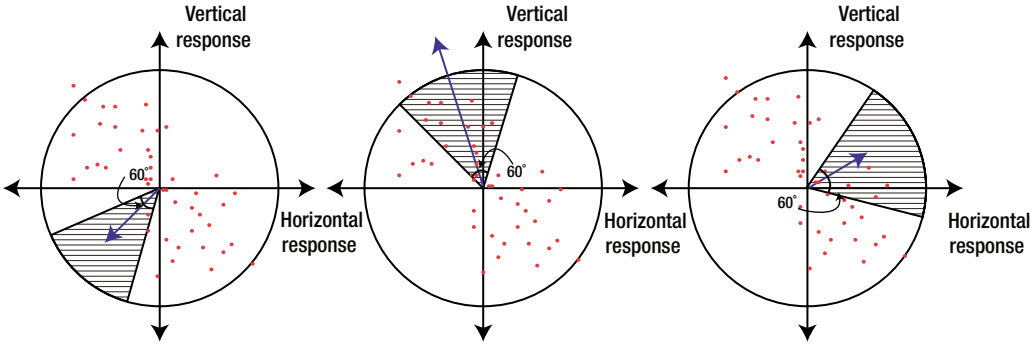


Figure 8-12. *Sliding orientation windows used in SURF*

All the responses within the window are summed to give a new vector; there are as many of these vectors as there are iterations of the sliding window. The largest of these vectors lends its orientation to the keypoint.

SURF Descriptor

The following steps are involved in calculating the SURF descriptor once you get oriented keypoints:

- Construct a square region around the keypoint, with side length equal to 20 times the scale of the keypoint and oriented by the orientation of the keypoint, as shown in Figure 8-13.

Figure 8-13. *Oriented square patches around SURF keypoints*

- This region is split up into 16 square subregions. In each of these subregions, a set of four features are calculated at 5 x 5 regularly spaced grid points. These features include Harr wavelet response in the horizontal and vertical directions and their absolute values
- These 4 features are summed up over each individual subregion and make up a four-element descriptor for each subregion. Sixteen such subregions make up the 64-element descriptor for the keypoint

SURF descriptors are matched using the same nearest neighbor distance ratio strategy as SIFT.

SURF has been shown to be at least twice as fast as SIFT without sacrificing performance. Listing 8-3 shows the SURF version of the keypoint-based object detector app, which uses the SurfFeatureDetector and SurfDescriptorExtractor classes. Figure 8-14 shows that SURF is just as accurate as SIFT, while giving us frame rates of up to 6 fps.

Listing 8-3. Program to illustrate SURF keypoint and descriptor extraction, and matching using FLANN

```
// Program to illustrate SURF keypoint and descriptor extraction, and matching using FLANN
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/nonfree/features2d.hpp>
#include <opencv2/features2d/features2d.hpp>

using namespace cv;
using namespace std;

int main() {
    Mat train = imread("template.jpg"), train_g;
    cvtColor(train, train_g, CV_BGR2GRAY);
```

```
    //detect SIFT keypoints and extract descriptors in the train image
    vector<KeyPoint> train_kp;
    Mat train_desc;

    SurfFeatureDetector featureDetector(100);
    featureDetector.detect(train_g, train_kp);
    SurfDescriptorExtractor featureExtractor;
    featureExtractor.compute(train_g, train_kp, train_desc);

    // FLANN based descriptor matcher object
    FlannBasedMatcher matcher;
    vector<Mat> train_desc_collection(1, train_desc);
    matcher.add(train_desc_collection);
    matcher.train();

    // VideoCapture object
    VideoCapture cap(0);

    unsigned int frame_count = 0;

    while(char(waitKey(1)) != 'q') {
        double t0 = getTickCount();
        Mat test, test_g;
        cap >> test;
        if(test.empty())
            continue;

        cvtColor(test, test_g, CV_BGR2GRAY);

        //detect SIFT keypoints and extract descriptors in the test image
        vector<KeyPoint> test_kp;
        Mat test_desc;
        featureDetector.detect(test_g, test_kp);
        featureExtractor.compute(test_g, test_kp, test_desc);

        // match train and test descriptors, getting 2 nearest neighbors for all test descriptors
        vector<vector<DMatch> > matches;
        matcher.knnMatch(test_desc, matches, 2);

        // filter for good matches according to Lowe's algorithm
        vector<DMatch> good_matches;
        for(int i = 0; i < matches.size(); i++) {
            if(matches[i][0].distance < 0.6 * matches[i][1].distance)
                good_matches.push_back(matches[i][0]);
        }

        Mat img_show;
        drawMatches(test, test_kp, train, train_kp, good_matches, img_show);
        imshow("Matches", img_show);

        cout << "Frame rate = " << getTickFrequency() / (getTickCount() - t0) << endl;
    }

    return 0;
}
```

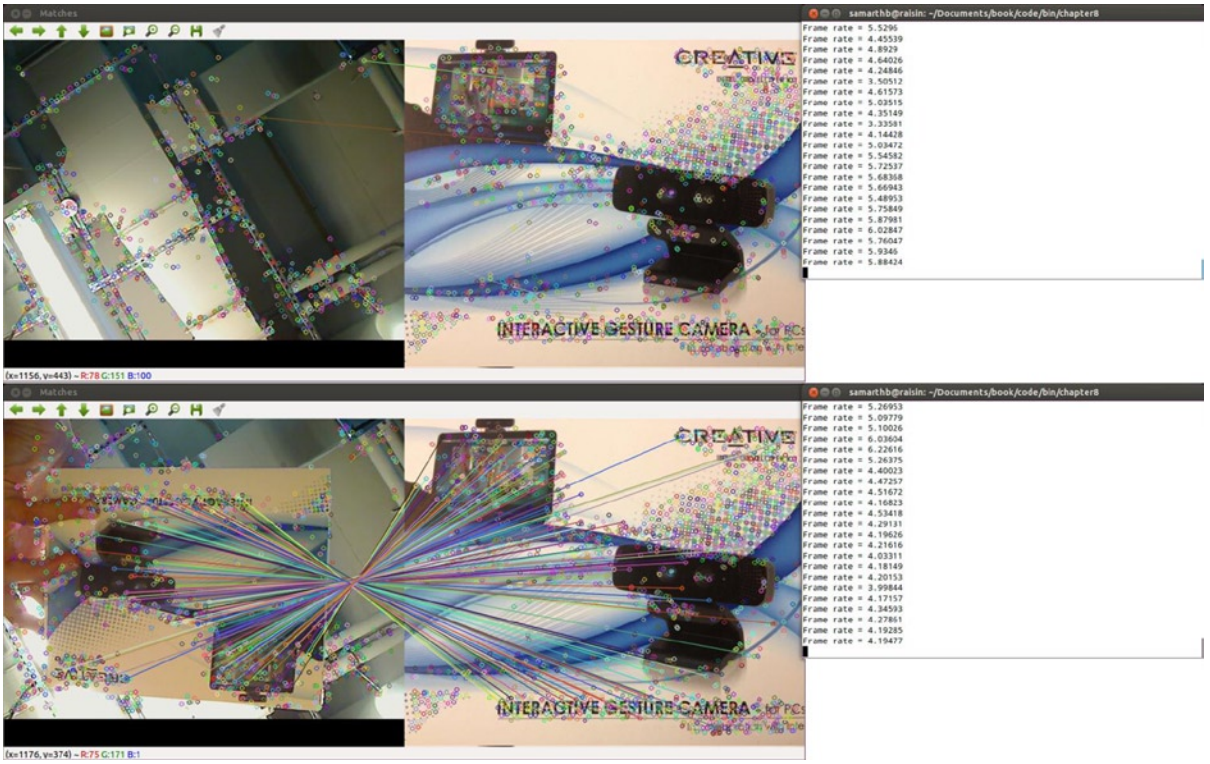


Figure 8-14. *SURF-based object detector using FLANN matching*

Note that both SIFT and SURF algorithms are patented in the United States (and maybe some other countries as well), so you will not be able to use them in a commercial application.

ORB (Oriented FAST and Rotated BRIEF)

ORB is a keypoint detection and description technique that is catching up fast in popularity with SIFT and SURF. Its claim to fame is extremely fast operation, while sacrificing very little on performance accuracy. ORB is scale and rotation invariant, robust to noise and affine transformations, and still manages to deliver a frame rate of 25 fps!

The algorithm is actually a combination of the FAST (Features from Accelerated Segment Test) keypoint detection with oriented added to the algorithm, and the BRIEF (Binary Robust Independent Elementary Features) keypoint descriptor algorithm modified to handle oriented keypoints. Before going further into describing ORB, here are the papers you can read to get detailed information about these algorithms:

- "ORB: An efficient alternative to SIFT or SURF," by Ethan Rublee et al.
- "Faster and better: A machine learning approach to corner detection," by Edward Rosten et al.
- "BRIEF: Binary Robust Independent Elementary Features," by Michael Colander et al.

Oriented FAST Keypoints

The original FAST keypoint detector tests for 16 pixels in a circle around a pixel. If the central pixel is darker or brighter than a threshold number of pixels out of 16, it is determined to be a corner. To make this procedure faster, a machine learning approach is used to decide an efficient order of checking the 16 pixels. The adaptation of FAST in ORB detects corners at multiple scales by making a scale pyramid of the image, and adds orientation to these corners by finding the intensity centroid. The intensity centroid of a patch of pixels is given by:

$$C = \left(\frac{m10}{m00} - \frac{m01}{m00}\right)$$

where

$$mpq = \sum_{x,y} x^p y^q I(x, y)$$

The orientation of the patch is the orientation of the vector connecting the patch center to the intensity centroid. Specifically:

$$\theta = atan2(m01, m10)$$

BRIEF Descriptors

BRIEF is based on the philosophy that a keypoint in an image can be described sufficiently by a series of binary pixel intensity tests around the keypoint. This is done by picking pairs of pixels around the keypoint (according to a random or nonrandom sampling pattern) and then comparing the two intensities. The test returns 1 if the intensity of the first pixel is higher than that of the second, and 0 otherwise. Since all these outputs are binary, one can just pack them into bytes for efficient memory storage. A big advantage is also that since the descriptors are binary, the distance measure between two descriptors is Hamming and not Euclidean. Hamming distance between two binary strings of the same length is the number of bits that differ between them. Hamming distance can be very efficiently implemented by doing a bitwise XOR operation between the two descriptors and then counting the number of 1s.

The BRIEF implementation in ORB uses a machine learning algorithm to get a pair picking pattern to pick 256 pairs that will capture the most information, and looks something like Figure 8-15.

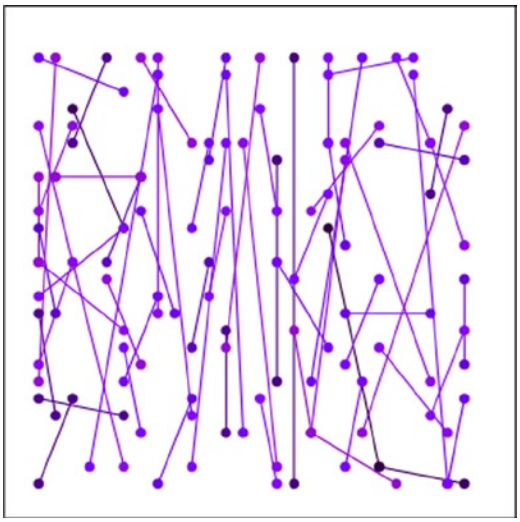


Figure 8-15. *Typical pair-picking pattern for BRIEF*

To compensate for the orientation of the keypoint, the coordinates of the patch around the keypoint are rotated by the orientation before picking pairs and performing the 256 binary tests.

Listing 8-4 implements the keypoint-based object detector using the ORB keypoints and features, and Figure 8-16 shows that the frame rate goes up to 28 fps, without the performance being compromised much. We have also used the locally sensitive hashing (LSH) algorithm for performing the FLANN based search, which speeds up Hamming distance based nearest neighbor searches even further.

Listing 8-4. Program to illustrate ORB keypoint and descriptor extraction, and matching using FLANN-LSH

```
// Program to illustrate ORB keypoint and descriptor extraction, and matching using FLANN-LSH
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/nonfree/features2d.hpp>
#include <opencv2/features2d/features2d.hpp>

using namespace cv;
using namespace std;

int main() {
    Mat train = imread("template.jpg"), train_g;
    cvtColor(train, train_g, CV_BGR2GRAY);

    //detect SIFT keypoints and extract descriptors in the train image
    vector<KeyPoint> train_kp;
    Mat train_desc;

    OrbFeatureDetector featureDetector;
    featureDetector.detect(train_g, train_kp);
    OrbDescriptorExtractor featureExtractor;
    featureExtractor.compute(train_g, train_kp, train_desc);
```

```
    cout << "Descriptor depth " << train_desc.depth() << endl;

    // FLANN based descriptor matcher object
    flann::Index flannIndex(train_desc, flann::LshIndexParams(12, 20, 2),
cvflann::FLANN_DIST_HAMMING);

    // VideoCapture object
    VideoCapture cap(0);

    unsigned int frame_count = 0;

    while(char(waitKey(1)) != 'q') {
        double t0 = getTickCount();
        Mat test, test_g;
        cap >> test;
        if(test.empty())
            continue;

        cvtColor(test, test_g, CV_BGR2GRAY);

        //detect SIFT keypoints and extract descriptors in the test image
        vector<KeyPoint> test_kp;
        Mat test_desc;
        featureDetector.detect(test_g, test_kp);
        featureExtractor.compute(test_g, test_kp, test_desc);

        // match train and test descriptors, getting 2 nearest neighbors for all test descriptors
        Mat match_idx(test_desc.rows, 2, CV_32SC1), match_dist(test_desc.rows, 2, CV_32FC1);
        flannIndex.knnSearch(test_desc, match_idx, match_dist, 2, flann::SearchParams());

        // filter for good matches according to Lowe's algorithm
        vector<DMatch> good_matches;
        for(int i = 0; i < match_dist.rows; i++) {
            if(match_dist.at<float>(i, 0) < 0.6 * match_dist.at<float>(i, 1)) {
                DMatch dm(i, match_idx.at<int>(i, 0), match_dist.at<float>(i, 0));
                good_matches.push_back(dm);
            }
        }

        Mat img_show;
        drawMatches(test, test_kp, train, train_kp, good_matches, img_show);
        imshow("Matches", img_show);

        cout << "Frame rate = " << getTickFrequency() / (getTickCount() - t0) << endl;
    }

    return 0;
}
```

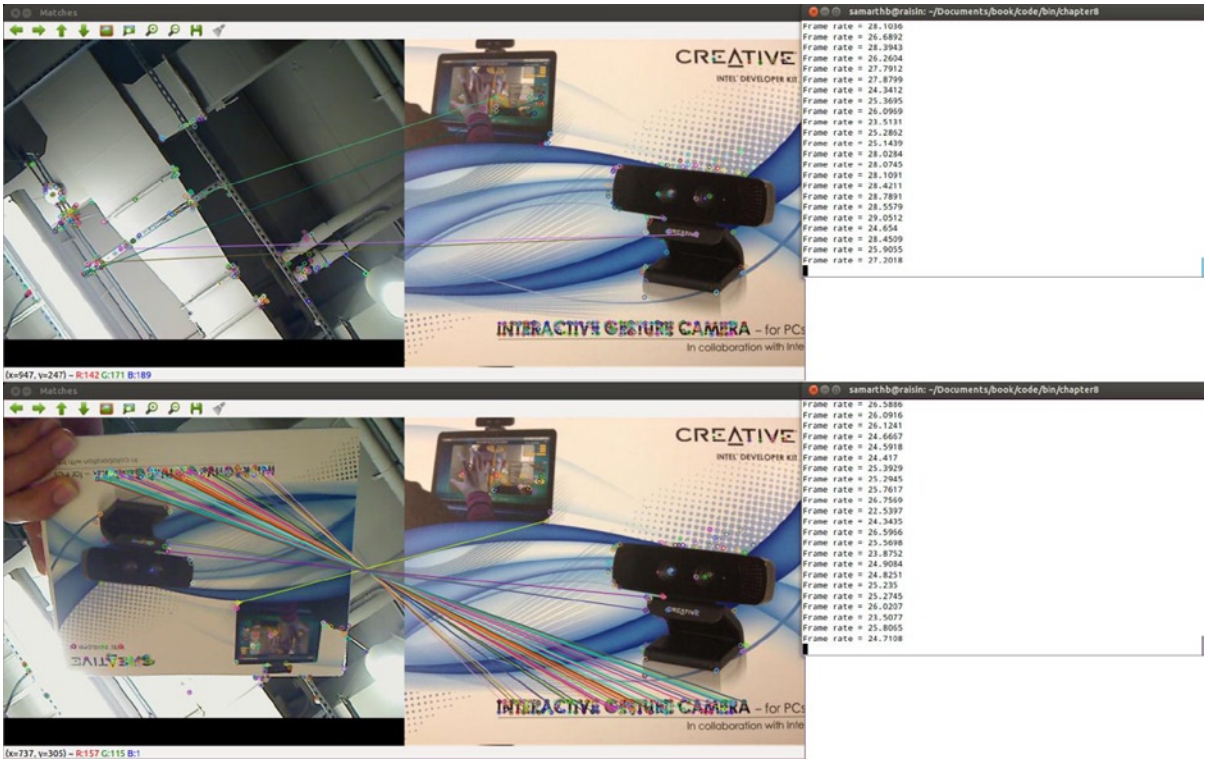


Figure 8-16. *ORB keypoint based object detecor*

Basic Machine Learning

In this section we will discuss some entry-level machine learning concepts, which we will employ in the next section to make our ultimate object detector app! Machine Learning (ML) basically deals with techniques for making a machine (computer) learn from *experience*. There are two types of problems the machine can learn to solve:

- Classification: Predict which category a data sample belongs to. Hence the output of a classification algorithm is a category, which is discrete—not continuous. For example, the machine learning problem we will deal with later on in this chapter is to decide which object (from a range of different objects) is present in an image. This is a classification problem, as the output is the object category, which is a discrete category number
- Regression: Predict the output of a function which is very difficult to express in an equation. For example, an algorithm that learns to predict the price of a stock from previous data is a regression algorithm. Note that the output is a continuous number—it can take any value

Training input to an ML algorithm consists of two parts:

- Features: The parameters one thinks are relevant to the problem at hand. As an example, for the object categorization problem we discussed earlier, features can be SURF descriptors of the image. Typically, an ML algorithm is given features from not one but many instances of data, to "teach" it many possible input conditions

- Labels: Each set of features associated with a data instance has a corresponding "correct response." This is called a label. For example, labels in the object classification problem can be the object category present in the training image. Classification problems will have discrete labels, while regression algorithms will have continuous labels.

Together, features and labels make up the "experience" required for the ML algorithm.

There is a plethora of ML algorithms out there. Naive Bayes classifiers, logistic regressors, support vector machines and hidden Markov models are some examples. Each algorithm has its own pros and cons. We will briefly discuss the workings of the support vector machine (SVM) here, and then move on to using the OpenCV implementation of SVMs in our object detector app.

SVMs

SVMs are the most widely used ML algorithms today because of their versatility and ease of use. The typical SVM is a classification algorithm, but some researchers have modified it to perform regression too. Basically, given a set of data points with labels, SVM tries to find a hyperplane that best separates the data correctly. A hyperplane is a linear structure in the given dimensionality. For example, it is a line if the data has two dimensions, a plane in three dimensions, a "four-dimensional plane" in four dimensions, and so on. By "best separation," I mean that the distance from the hyperplane to the nearest points on both sides of the hyperplane must be equal and the maximum possible. A naïve example is shown for two dimensions in the Figure 8-17, where the line is the hyperplane.

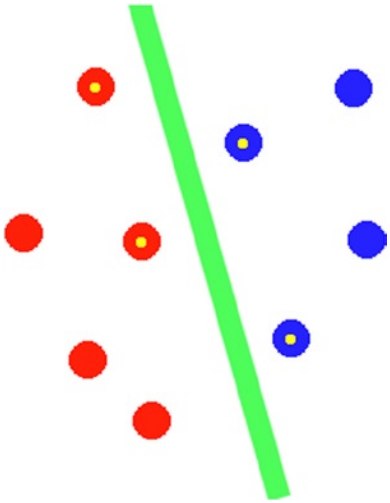


Figure 8-17. *SVM classifier hyperplane*

The astute reader might argue that not all arrangements of data points are correctly separable by a hyperplane. For example, if you have four points with labels as shown in Figure 8-18, no line can separate all of them correctly.

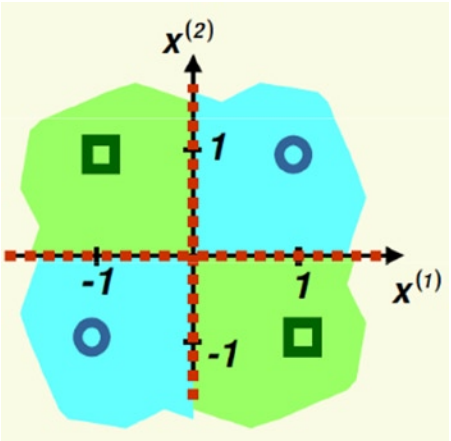


Figure 8-18. *SVM XOR problem*

To solve this problem, SVMs use the concept of *kernels*. A kernel is a function that maps data from one configuration to another, possibly changing the dimensionality. For example, a kernel that could solve the problem in Figure 8-18 would be one that could "fold" the plane about the `x1 = x2` line (Figure 8-18). If you use a kernel, the SVM algorithm finds a hyperplane for correct classification in the feature space transformed by the kernel. Then it transforms each test input by the kernel and uses the trained hyperplane to classify. The hyperplane might not remain linear in the original feature space, but it is linear in the transformed feature space.

In OpenCV, the `CvSVM` class with its `train()` and `predict()` methods can be used to train an SVM using different kernels and predict using the trained SVM, respectively.

Object Categorization

In this section, we will develop an object detector app that uses SURF descriptors of keypoints as features, and SVMs to predict the category of object present in the image. For this app we will use the CMake build system which makes configuring code project and linking it to various libraries a breeze. We will also use the `filesystem` component of the Boost C++ libraries to go through our data folders and the STL data structures `map` and `multimap` to store organize data in an easily accessible manner. Comments in the code are written to make everything easily understandable, but you are encouraged to get some very basic knowledge of CMake and the STL data structures map and remap before going ahead.

Strategy

Here I outline the strategy we will use, which was proposed by Gabriella Csurka et al. in their paper "Visual Categorization with Bags of Keypoints," and implemented wonderfully in the OpenCV `features2d` module.

- Compute (SURF) descriptors at keypoints in all templates and pool all of them together
- Group similar descriptors into an arbitrary number of clusters. Here, similarity is decided by Euclidean distance between the 64-element SURF descriptors and grouping is done by the `cluster()` method of the `BOWKMeansTrainer` class. These clusters are called "bags of keypoints" or "visual words" in the paper and they collectively represent the "vocabulary" of the program. Each cluster has a cluster center, which can be thought of as the representative descriptor of all the descriptors belonging to that cluster

- Now, compute training data for the SVM classifiers. This is done by
  - Computing (SURF) descriptors for each training image
  - Associating each of these descriptors with one of the clusters in the vocabulary by Euclidean distance to cluster center
  - Making a histogram from this association. This histogram has as many bins as there are clusters in the vocabulary. Each bin counts how many descriptors in the training image are associated with the cluster corresponding to that bin. Intuitively, this histogram describes the image in the "visual words" of the "vocabulary," and is called the "bag of visual words descriptor" of the image. This is done by using the `compute()` method of the `BOWImageDescriptorExtractor` class
- After that, train a one-vs-all SVM for each category of object using the training data. Details:
  - For each category, positive data examples are BOW descriptors of its training images, while negative examples are BOW descriptors of training images of all other categories
  - Positive examples are labeled 1, while negative examples are labeled 0
  - Both of these are given to the `train()` method of the `CvSVM` class to train an SVM. We thus have an SVM for every category
  - SVMs are also capable of multiclass classification, but intuitively (and mathematically) it is easier for a classifier to decide whether a data sample belongs to a class or not rather than decide which class a data sample belongs to put of many classes
- Now, classify by using the trained SVMs. Details:
  - Capture an image from the camera and calculate the BOW descriptor, again using the `compute()` method of the `BOWImageDescriptorExtractor` class
  - Give this description to the SVMs to predict upon using the `predict()` method of the `CvSVM` class
  - For each category, the corresponding SVM will tell us whether the image described by the descriptor belongs to the category or not, and also how confident it is in its decision. The smaller this measure, the more confident the SVM is in its decision
  - Pick the category that has the smallest measure as the detected category

Organization

The project folder should be organized as shown in Figure 8-19.

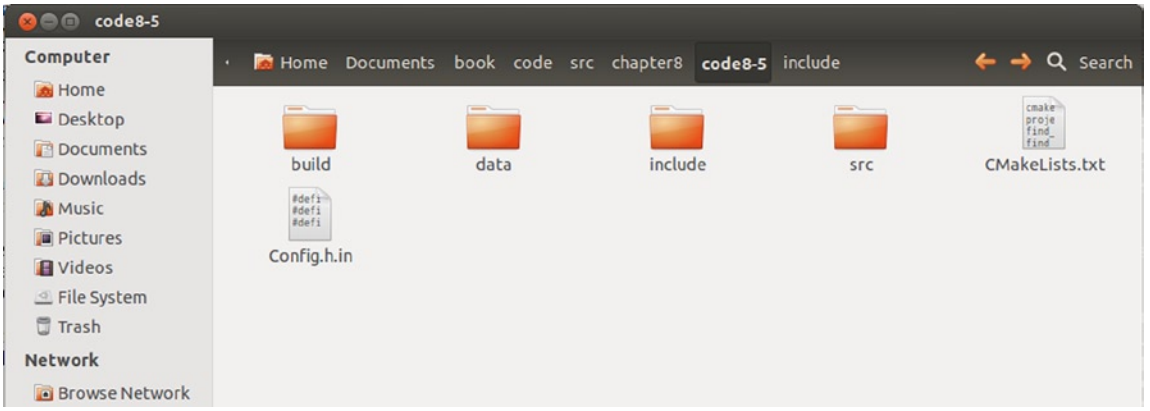


Figure 8-19. *Project root folder organization*

Figure 8-20 shows the organization of the "data" folder. It contains two folders called "templates" and "train_images". The "templates" folder contains images showing the objects that we want to categorize, according to category names. As you can see in Figure 8-20, my categories are called "a," "b," and "c." The "train_images" folder has as many folders as there are templates, named again by object category. Each folder contains images for training SVMs for classification, and the names of the images themselves do not matter.

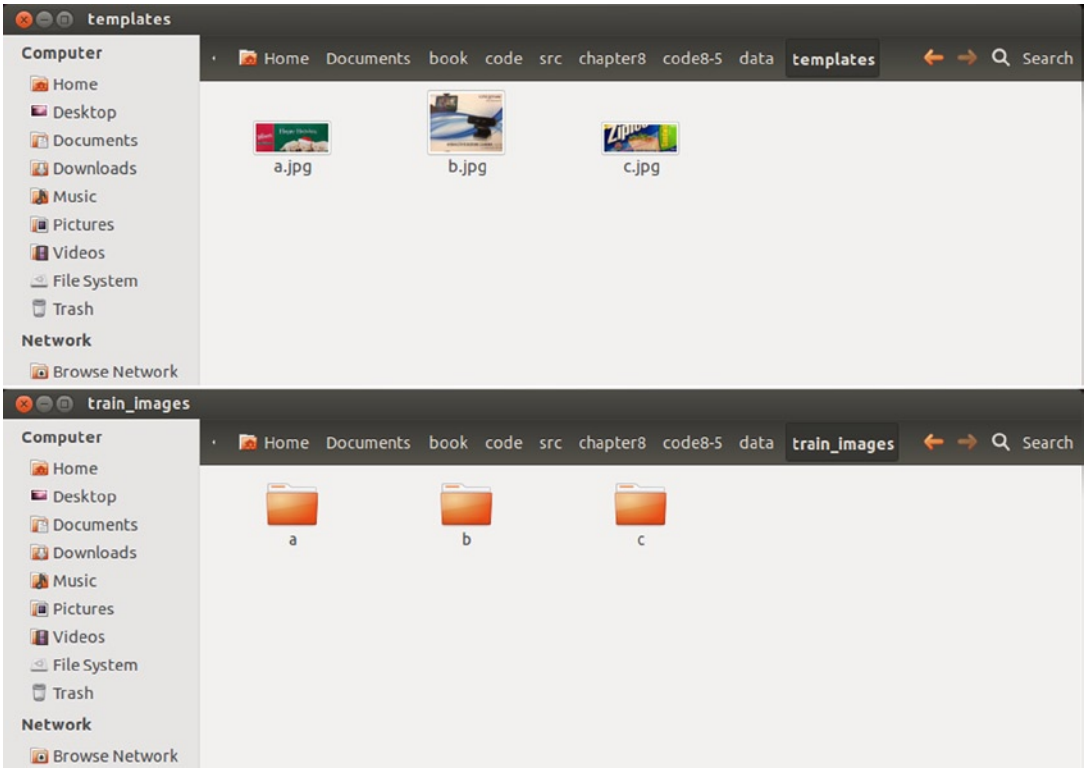


Figure 8-20. *Organization of the two folders in the "data" folder—"templates" and "train_images"*

Figure 8-21 shows my templates and training images for the three categories. Note that the templates are cropped to show just the object and nothing else.

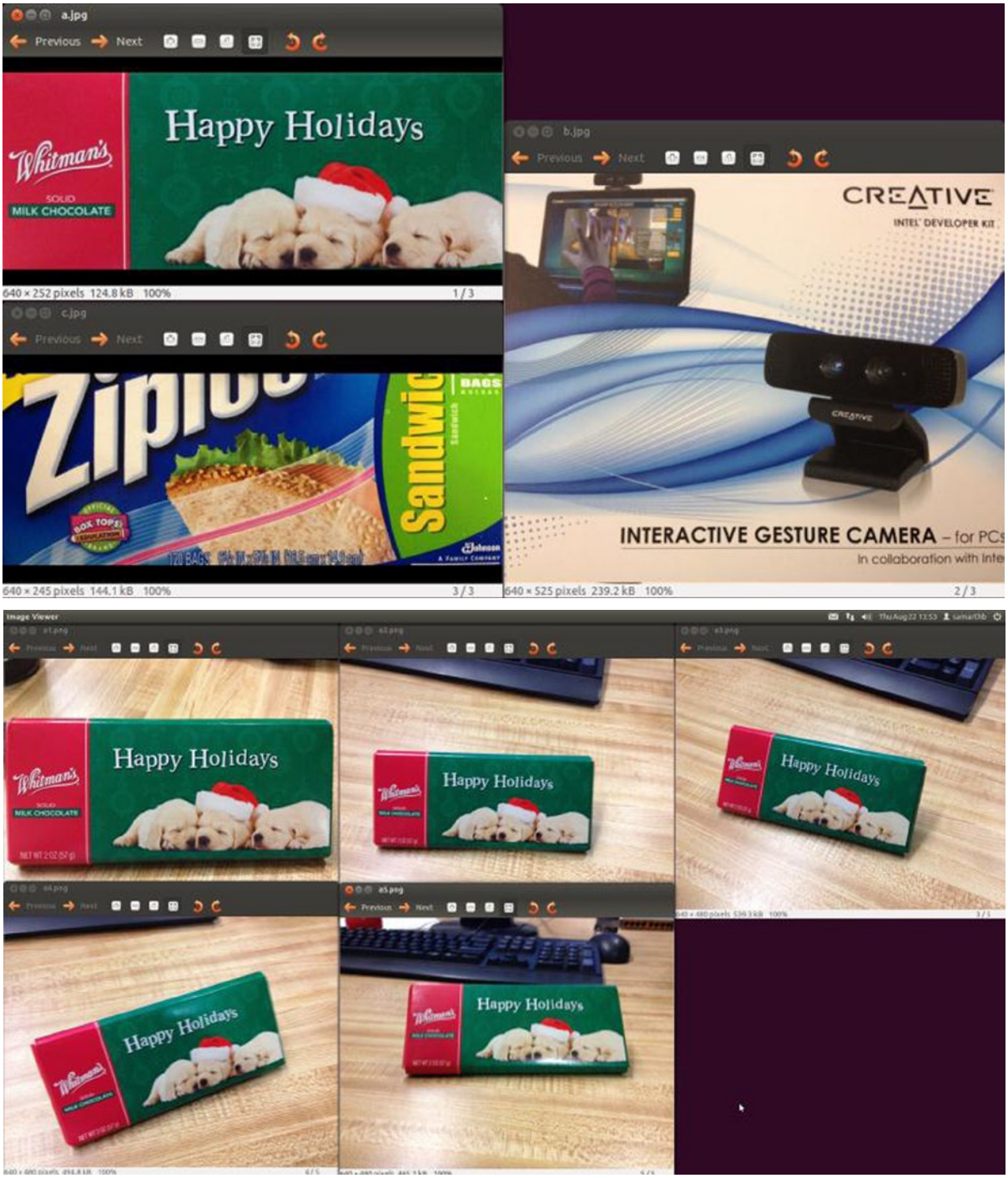


Figure 8-21. (*continued*)

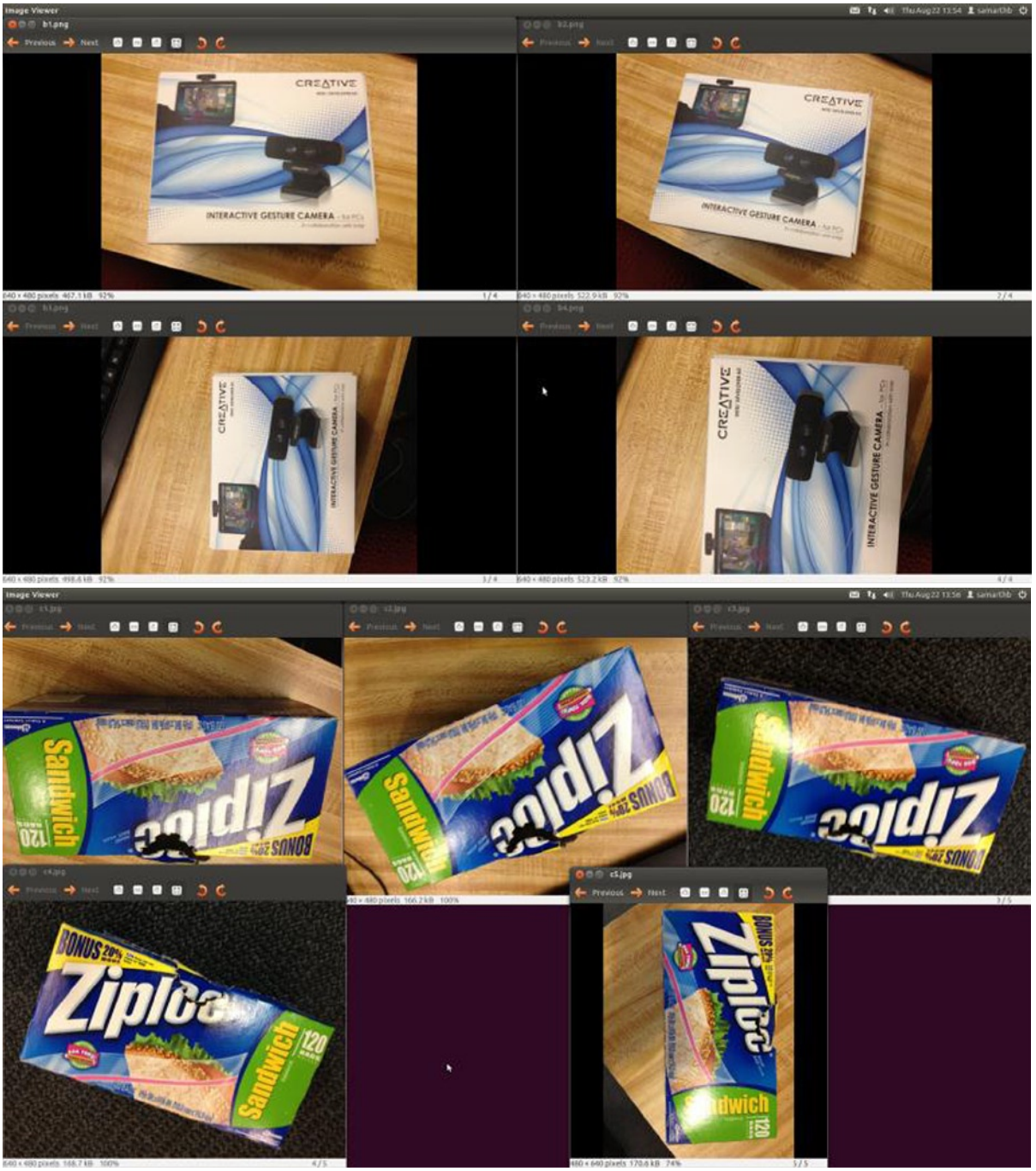


Figure 8-21. *(Top to bottom) Templates and training images for categories "a," "b," and "c"*

Now for the files CmakeLists.txt and Config.h.in, which are shown in Listings 8-5 and 8-6, respectively. CmakeLists.txt is the main file that is used my CMake to set all the properties of the project and locations of the linked libraries (OpenCV and Boost) in our case. Config.h.in set our folder path configurations. This will automatically generate the file Config.h in the "include" folder, which will be includable in our source CPP file and will define paths of folders that store our templates and training images in variables TEMPLATE_FOLDER and TRAIN_FOLDER, respectively.

Listing 8-5: CmakeLists.txt

```
# Minimum required CMake version
cmake_minimum_required(VERSION 2.8)

# Project name
project(object_categorization)

# Find the OpenCV installation
find_package(OpenCV REQUIRED)

# Find the Boost installation, specifically the components 'system' and 'filesystem'
find_package(Boost COMPONENTS system filesystem REQUIRED)

# ${PROJECT_SOURCE_DIR} is the name of the root directory of the project
# TO_NATIVE_PATH converts the path ${PROJECT_SOURCE_DIR}/data/ to a full path and the file()
command stores it in DATA_FOLDER
file(TO_NATIVE_PATH "${PROJECT_SOURCE_DIR}/data/" DATA_FOLDER)
# set TRAIN_FOLDER to DATA_FOLDER/train_images - this is where we will put our templates for
constructing the vocabulary
set(TRAIN_FOLDER "${DATA_FOLDER}train_images/")
# set TEMPLATE_FOLDER to DATA_FOLDER/templates - this is where we will put our traininfg images,
in folders organized by category
set(TEMPLATE_FOLDER "${DATA_FOLDER}templates/")

# set the configuration input file to ${PROJECT_SOURCE_DIR}/Config.h.in and the includable header
file holding configuration information to ${PROJECT_SOURCE_DIR}/include/Config.h
configure_file("${PROJECT_SOURCE_DIR}/Config.h.in" "${PROJECT_SOURCE_DIR}/include/Config.h")

# Other directories where header files for linked libraries can be found
include_directories(${OpenCV_INCLUDE_DIRS} "${PROJECT_SOURCE_DIR}/include" ${Boost_INCLUDE_DIRS})

# executable produced as a result of compilation
add_executable(code8-5 src/code8-5.cpp)
# libraries to be linked with this executable - OpenCV and Boost (system and filesystem components)
target_link_libraries(code8-5 ${OpenCV_LIBS} ${Boost_SYSTEM_LIBRARY} ${Boost_FILESYSTEM_LIBRARY})
```

Code 8-6: Config.h.in

```
// Preprocessor directives to set variables from values in the CMakeLists.txt files
#define DATA_FOLDER "@DATA_FOLDER@"
#define TRAIN_FOLDER "@TRAIN_FOLDER@"
#define TEMPLATE_FOLDER "@TEMPLATE_FOLDER@"
```

This organization of the folders, combined by the configuration files, will allow us to automatically and efficiently manage and read our dataset and make the whole algorithm scalable for any number of object categories, as you will find out when you read through the main code.

The main code, which should be put in the folder called "src" and named according the filename mentioned in the `add_executable()` command in the `CMakeLists.txt` file, is shown in Listing 8-7. It is heavily commented to help you understand what is going on. I will discuss some features of the code afterward but, as always, you are encouraged to look up functions in the online OpenCV documentation!

Listing 8-7. Program to illustrate BOW object categorization

```
// Program to illustrate BOW object categorization
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/features2d/features2d.hpp>
#include <opencv2/nonfree/features2d.hpp>
#include <opencv2/ml/ml.hpp>
#include <boost/filesystem.hpp>
#include "Config.h"

using namespace cv;
using namespace std;
using namespace boost::filesystem3;

class categorizer {
    private:
        map<string, Mat> templates, objects, positive_data, negative_data; //maps from category
names to data
        multimap<string, Mat> train_set; //training images, mapped by category name
        map<string, CvSVM> svms;         //trained SVMs, mapped by category name
        vector<string> category_names;   //names of the categories found in TRAIN_FOLDER
        int categories;                  //number of categories
        int clusters;                    //number of clusters for SURF features to build vocabulary
        Mat vocab;                       //vocabulary

        // Feature detectors and descriptor extractors
        Ptr<FeatureDetector> featureDetector;
        Ptr<DescriptorExtractor> descriptorExtractor;
        Ptr<BOWKMeansTrainer> bowtrainer;
        Ptr<BOWImgDescriptorExtractor> bowDescriptorExtractor;
        Ptr<FlannBasedMatcher> descriptorMatcher;

        void make_train_set();           //function to build the training set multimap
        void make_pos_neg();             //function to extract BOW features from training images and
organize them into positive and negative samples
        string remove_extension(string); //function to remove extension from file name, used for
organizing templates into categories
    public:
        categorizer(int);                //constructor
        void build_vocab();              //function to build the BOW vocabulary
```

```
        void train_classifiers();        //function to train the one-vs-all SVM classifiers for all
categories
        void categorize(VideoCapture);   //function to perform real-time object categorization on
camera frames
};

string categorizer::remove_extension(string full) {
    int last_idx = full.find_last_of(".");
    string name = full.substr(0, last_idx);
    return name;
}

categorizer::categorizer(int _clusters) {
    clusters = _clusters;
    // Initialize pointers to all the feature detectors and descriptor extractors
    featureDetector = (new SurfFeatureDetector());
    descriptorExtractor = (new SurfDescriptorExtractor());
    bowtrainer = (new BOWKMeansTrainer(clusters));
    descriptorMatcher = (new FlannBasedMatcher());
    bowDescriptorExtractor = (new BOWImgDescriptorExtractor(descriptorExtractor,
descriptorMatcher));

    // Organize the object templates by category
    // Boost::filesystem directory iterator
    for(directory_iterator i(TEMPLATE_FOLDER), end_iter; i != end_iter; i++) {
        // Prepend full path to the file name so we can imread() it
        string filename = string(TEMPLATE_FOLDER) + i->path().filename().string();
        // Get category name by removing extension from name of file
        string category = remove_extension(i->path().filename().string());
        Mat im = imread(filename), templ_im;
        objects[category] = im;
        cvtColor(im, templ_im, CV_BGR2GRAY);
        templates[category] = templ_im;
    }
    cout << "Initialized" << endl;

    // Organize training images by category
    make_train_set();
}

void categorizer::make_train_set() {
    string category;
    // Boost::filesystem recursive directory iterator to go through all contents of TRAIN_FOLDER
    for(recursive_directory_iterator i(TRAIN_FOLDER), end_iter; i != end_iter; i++) {
        // Level 0 means a folder, since there are only folders in TRAIN_FOLDER at the zeroth level
        if(i.level() == 0) {
            // Get category name from name of the folder
            category = (i -> path()).filename().string();
            category_names.push_back(category);
        }
```

```
        // Level 1 means a training image, map that by the current category
        else {
            // File name with path
            string filename = string(TRAIN_FOLDER) + category + string("/") +
(i -> path()).filename().string();
            // Make a pair of string and Mat to insert into multimap
            pair<string, Mat> p(category, imread(filename, CV_LOAD_IMAGE_GRAYSCALE));
            train_set.insert(p);
        }
    }
    // Number of categories
    categories = category_names.size();
    cout << "Discovered " << categories << " categories of objects" << endl;
}

void categorizer::make_pos_neg() {
    // Iterate through the whole training set of images
    for(multimap<string, Mat>::iterator i = train_set.begin(); i != train_set.end(); i++) {
        // Category name is the first element of each entry in train_set
        string category = (*i).first;
        // Training image is the second elemnt
        Mat im = (*i).second, feat;

        // Detect keypoints, get the image BOW descriptor
        vector<KeyPoint> kp;
        featureDetector -> detect(im, kp);
        bowDescriptorExtractor -> compute(im, kp, feat);

        // Mats to hold the positive and negative training data for current category
        Mat pos, neg;
        for(int cat_index = 0; cat_index < categories; cat_index++) {
            string check_category = category_names[cat_index];
            // Add BOW feature as positive sample for current category ...
            if(check_category.compare(category) == 0)
                positive_data[check_category].push_back(feat);
            //... and negative sample for all other categories
            else
                negative_data[check_category].push_back(feat);
        }
    }

    // Debug message
    for(int i = 0; i < categories; i++) {
        string category = category_names[i];
        cout << "Category " << category << ": " << positive_data[category].rows << " Positives, "
<< negative_data[category].rows << " Negatives" << endl;
    }
}
```

```
void categorizer::build_vocab() {
    // Mat to hold SURF descriptors for all templates
    Mat vocab_descriptors;
    // For each template, extract SURF descriptors and pool them into vocab_descriptors
    for(map<string, Mat>::iterator i = templates.begin(); i != templates.end(); i++) {
        vector<KeyPoint> kp; Mat templ = (*i).second, desc;
        featureDetector -> detect(templ, kp);
        descriptorExtractor -> compute(templ, kp, desc);
        vocab_descriptors.push_back(desc);
    }

    // Add the descriptors to the BOW trainer to cluster
    bowtrainer -> add(vocab_descriptors);
    // cluster the SURF descriptors
    vocab = bowtrainer->cluster();

    // Save the vocabulary
    FileStorage fs(DATA_FOLDER "vocab.xml", FileStorage::WRITE);
    fs << "vocabulary" << vocab;
    fs.release();

    cout << "Built vocabulary" << endl;
}

void categorizer::train_classifiers() {
    // Set the vocabulary for the BOW descriptor extractor
    bowDescriptorExtractor -> setVocabulary(vocab);
    // Extract BOW descriptors for all training images and organize them into positive and negative
samples for each category
    make_pos_neg();

    for(int i = 0; i < categories; i++) {
        string category = category_names[i];

        // Postive training data has labels 1
        Mat train_data = positive_data[category], train_labels = Mat::ones(train_data.rows, 1, CV_32S);
        // Negative training data has labels 0
        train_data.push_back(negative_data[category]);
        Mat m = Mat::zeros(negative_data[category].rows, 1, CV_32S);
        train_labels.push_back(m);

        // Train SVM!
        svms[category].train(train_data, train_labels);

        // Save SVM to file for possible reuse
        string svm_filename = string(DATA_FOLDER) + category + string("SVM.xml");
        svms[category].save(svm_filename.c_str());

        cout << "Trained and saved SVM for category " << category << endl;
    }

}
```

```
void categorizer::categorize(VideoCapture cap) {
    cout << "Starting to categorize objects" << endl;
    namedWindow("Image");

    while(char(waitKey(1)) != 'q') {
        Mat frame, frame_g;
        cap >> frame;
        imshow("Image", frame);

        cvtColor(frame, frame_g, CV_BGR2GRAY);

        // Extract frame BOW descriptor
        vector<KeyPoint> kp;
        Mat test;
        featureDetector -> detect(frame_g, kp);
        bowDescriptorExtractor -> compute(frame_g, kp, test);

        // Predict using SVMs for all catgories, choose the prediction with the most negative signed
distance measure
        float best_score = 777;
        string predicted_category;
        for(int i = 0; i < categories; i++) {
            string category = category_names[i];
            float prediction = svms[category].predict(test, true);
            //cout << category << " " << prediction << " ";
            if(prediction < best_score) {
                best_score = prediction;
                predicted_category = category;
            }
        }
        //cout << endl;

        // Pull up the object template for the detected category and show it in a separate window
        imshow("Detected object", objects[predicted_category]);
    }
}

int main() {
    // Number of clusters for building BOW vocabulary from SURF features
    int clusters = 1000;
    categorizer c(clusters);
    c.build_vocab();
    c.train_classifiers();

    VideoCapture cap(0);
    namedWindow("Detected object");
    c.categorize(cap);
    return 0;
}
```

The only syntactic concept in the code that I want to emphasize is the use of the STL data structures `map` and `multimap`. They allow you to store data in the form of (key, value) pairs where key and value can be pretty much any data type. You can access the value associated with key by indexing the map with the key. Here we set our keys to be the object category names so that we can access our category-specific templates, training images, and SVMs easily using the category name as index into the map. We have to use `multimap` for organizing the training images, because we can have more than one training image for a single category. This trick allows the program to use any arbitrary number of object categories quite easily!

Summary

This is one of the main chapters of the book for two reasons. First, I hope, it showed you the full power of OpenCV, STL data structures, and object-oriented programming approaches when used together. Second, you now have the tools of the trade to be able to accomplish any modest object detector application. SIFT, SURF, and ORB keypoint-based object detectors are very common as base-level object detectors in a lot of robotics applications. We saw how SIFT is the slowest of the three (but also the most accurate in matching and able to extract the highest number of meaningful keypoints), whereas computing and matching ORB descriptors is very fast (but ORB tends to miss out on some keypoints). SURF falls somewhere in the middle, but tends to favor accuracy more than speed. An interesting fact is that because of the structure of OpenCV's `features2d` module, you can use one kind of keypoint extractor and another kind of descriptor extractor and matcher. For example, you can use SURF keypoints for speed but then compute and match SIFT descriptors at those keypoints for matching accuracy.

I also discussed basic machine learning, which is a skill I think every computer vision scientist must have, because the more automated and intelligent your vision program, the cooler your robot will be!

The CMake building system is also a standard in big projects for easy build management and you now know the basics of using it!

The next chapter discusses how you can use your knowledge of keypoint descriptor matching to find out correspondences between images of an object viewed from different perspectives. You will also see how these projective relations enable you to make beautiful seamless panoramas out of a bunch of images!

CHAPTER 9

Affine and Perspective Transformations and Their Applications to Image Panoramas

In this chapter you will learn about two important classes of geometric image transformations—Affine and Perspective—and how to represent and use them as matrices in your code. This will act as base knowledge for the next chapter, which deals with stereo vision and a lot of 3D image geometry.

Geometric image transformations are just transformations of images that follow geometric rules. The simplest of geometric transformations are rotating and scaling an image. There can be other more complicated geometric transformations, too. Another property of these transformations is that they are all linear and hence can be expressed as matrices and transformation of the image just amounts to matrix multiplication. As you can imagine, given two images (one original and the other transformed), you can recover the transformation matrix if you have enough point correspondences between the two images. You will learn how to recover Affine as well as Perspective transformations by using point correspondences between images clicked by the user. Later, you will also learn how to automate the process of finding correspondences by matching descriptors at keypoints. Oh, and you will be able to put all this knowledge to use by learning how to make beautiful panoramas by stitching together a bunch of images, using OpenCV's excellent stitching module!

Affine Transforms

An affine transform is any linear transform that preserves the "parallelness" of lines after transformation. It also preserves points as points, straight lines as straight lines and the ratio of distances of points along straight lines. It does not preserve angles between straight lines. Affine transforms include all types of rotation, translation, and mirroring of images. Let us now see how affine transforms can be expressed as matrices.

Let `(x, y)` be the coordinates of a point in the original image and `(x', y')` be the coordinates of that point after transformation, in the transformed image. Different transformations are:

- Scaling: `x' = a*x, y' = b*y`
- Flipping both X and Y coordinates: `x' = -x, y' = -y`
- Rotation counterclockwise about the origin by an angle θ: `x' = x*cos(θ)–y*sin(θ), y' = x*sin(θ) + y*cos(θ)`

Because all geometric transforms are linear, we can relate `(x', y')` to `(x, y)` by a matrix multiplication with a 2x2 matrix `M`:

```
(x', y') = M * (x, y)
```

The matrix M takes the following forms for the three kinds of transformations outlined above:

- Scaling: $M = \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}$, where `a` is the scaling factor for X coordinates and `b` is the scaling factor for Y coordinates
- Flipping both X and Y coordinates: $M = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$
- Rotation counterclockwise about the origin by an angle: `θ`: $M = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$

Except for the flipping matrix, the determinant of the 2 x 2 part of all Affine transform matrices must be +1.

Applying Affine Transforms

In OpenCV it is easy to construct an Affine transformation matrix and apply that transformation to an image. Let us first look at the function that applies an affine transform so that we can understand the structure of the OpenCV affine transform matrix better. The function `warpAffine()` takes a source image and a 2x3 matrix `M` and gives a transformed output image. Suppose M is of the form:

$$M = \begin{matrix} M11 & M12 & M13 \\ M21 & M22 & M23 \end{matrix}$$

`warpAffine()` applies the following transform:

$$dst(x, y) = src(M11 * x + M12 * y + M13, M21 * x + M22 * y + M23)$$

A potential source of bugs while hand-constructing the matrix to be given to `warpAffine()` is that OpenCV places the origin at the top-left corner of the image. This fact does not affect scaling transforms, but it does affect flipping and rotation transforms. Specifically, for a successful flip the `M` input to `warpAffine()` must be:

$$M = \begin{matrix} -1 & 0 & im.cols \\ 0 & -1 & im.rows \end{matrix}$$

The OpenCV function `getRotationMatrix2D()` gives out a 2 x 3 Affine transformation matrix that does a rotation. It takes the angle of rotation (measured counter clockwise in degrees from the horizontal axis) and center of rotation as input. For a normal rotation, you would want the center of rotation to be at the center of the image. Listing 9-1 shows how you can use `getRotationMatrix2D()` to get a rotation matrix and use `warpAffine()` to apply it to an image. Figure 9-1 shows the original and Affine transformed images.

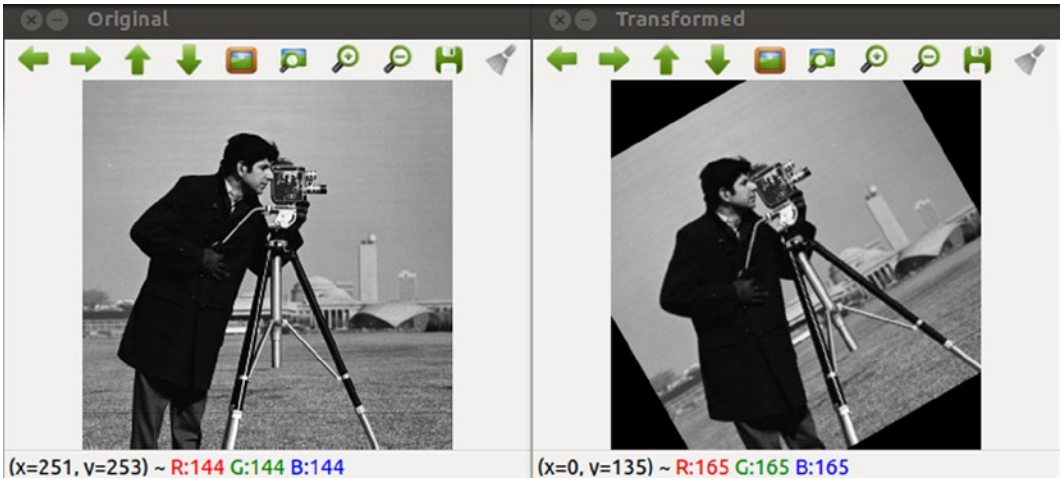


Figure 9-1. *Applying simple Affine transforms*

Listing 9-1. Program to illustrate a simple affine transform

```
//Program to illustrate a simple affine transform
//Author: Samarth Manoj Brahmbhatt, University of Pennsyalvania

#include <opencv2/opencv.hpp>
#include <opencv2/stitching/stitcher.hpp>
#include <opencv2/stitching/warpers.hpp>
#include "Config.h"

using namespace std;
using namespace cv;

int main() {
    Mat im = imread(DATA_FOLDER_1 + string("/image.jpg")), im_transformed;
    imshow("Original", im);

    int rotation_degrees = 30;

    // Construct Affine rotation matrix
    Mat M = getRotationMatrix2D(Point(im.cols/2, im.rows/2), rotation_degrees, 1);
    cout << M << endl;

    // Apply Affine transform
    warpAffine(im, im_transformed, M, im.size(), INTER_LINEAR);

    imshow("Transformed", im_transformed);

    while(char(waitKey(1)) != 'q') {}

    return 0;
}
```

Estimating Affine Transforms

Sometimes you know that one image is related to another by an Affine (or nearly Affine) transform and you want to get the Affine transformation matrix for some other calculation (e.g., to estimate the rotation of the camera). The OpenCV function `getAffineTransform()` is handy for such applications. The idea is that if you have three pairs of corresponding points in the two images you can recover the Affine transform that relates them using simple math. This is because each pair gives you two equations (one relating X coordinates and one relating Y coordinates). Hence you need three such pairs to solve for all six elements of the 2 x 3 Affine transformation matrix. `getAffineTransform()` does the equation-solving for you by taking in two vectors of three Point2f's each—one for original points and one for transformed points. In Listing 9-2, the user is asked to click corresponding points in the two images. These points are used to recover the Affine transform. To verify that the transform recovered is correct, the user is also shown the difference between the original transformed image and the untransformed image transformed by the recovered Affine transform. Figure 9-2 shows that the recovered Affine transform is actually correct (the difference image is almost all zeros—black).

Listing 9-2. Program to illustrate affine transform recovery

```
//Program to illustrate affine transform recovery
//Author: Samarth Manoj Brahmbhatt, University of Pennsyalvania

#include <opencv2/opencv.hpp>
#include <opencv2/stitching/stitcher.hpp>
#include <opencv2/stitching/warpers.hpp>
#include <opencv2/highgui/highgui.hpp>
#include "Config.h"

using namespace std;
using namespace cv;

// Mouse callback function
void on_mouse(int event, int x, int y, int, void* _p) {
    Point2f* p = (Point2f *)_p;
    if (event == CV_EVENT_LBUTTONUP) {
        p->x = x;
        p->y = y;
    }
}

class affine_transformer {
    private:
        Mat im, im_transformed, im_affine_transformed, im_show, im_transformed_show;
        vector<Point2f> points, points_transformed;
        Mat M; // Estimated Affine transformation matrix
        Point2f get_click(string, Mat);
    public:
        affine_transformer(); //constructor
        void estimate_affine();
        void show_diff();
};
```

```
affine_transformer::affine_transformer() {
    im = imread(DATA_FOLDER_2 + string("/image.jpg"));
    im_transformed = imread(DATA_FOLDER_2 + string("/transformed.jpg"));
}

// Function to get location clicked by user on a specific window
Point2f affine_transformer::get_click(string window_name, Mat im) {
    Point2f p(-1, -1);
    setMouseCallback(window_name, on_mouse, (void *)&p);
    while(p.x == -1 && p.y == -1) {
        imshow(window_name, im);
        waitKey(20);
    }
    return p;
}

void affine_transformer::estimate_affine() {
    imshow("Original", im);
    imshow("Transformed", im_transformed);

    cout << "To estimate the Affine transform between the original and transformed images you will
have to click on 3 matching pairs of points" << endl;

    im_show = im.clone();
    im_transformed_show = im_transformed.clone();
    Point2f p;

    // Get 3 pairs of matching points from user
    for(int i = 0; i < 3; i++) {
        cout << "Click on a distinguished point in the ORIGINAL image" << endl;
        p = get_click("Original", im_show);
        cout << p << endl;
        points.push_back(p);
        circle(im_show, p, 2, Scalar(0, 0, 255), -1);
        imshow("Original", im_show);

        cout << "Click on a distinguished point in the TRANSFORMED image" << endl;
        p = get_click("Transformed", im_transformed_show);
        cout << p << endl;
        points_transformed.push_back(p);
        circle(im_transformed_show, p, 2, Scalar(0, 0, 255), -1);
        imshow("Transformed", im_transformed_show);
    }

    // Estimate Affine transform
    M = getAffineTransform(points, points_transformed);
    cout << "Estimated Affine transform = " << M << endl;

    // Apply estimates Affine transfrom to check its correctness
    warpAffine(im, im_affine_transformed, M, im.size());
    imshow("Estimated Affine transform", im_affine_transformed);
}
```

```
void affine_transformer::show_diff() {
    imshow("Difference", im_transformed - im_affine_transformed);
}

int main() {
    affine_transformer a;
    a.estimate_affine();
    cout << "Press 'd' to show difference, 'q' to end" << endl;
    if(char(waitKey(-1)) == 'd') {
        a.show_diff();
        cout << "Press 'q' to end" << endl;
        if(char(waitKey(-1)) == 'q') return 0;
    }
    else
        return 0;
}
```

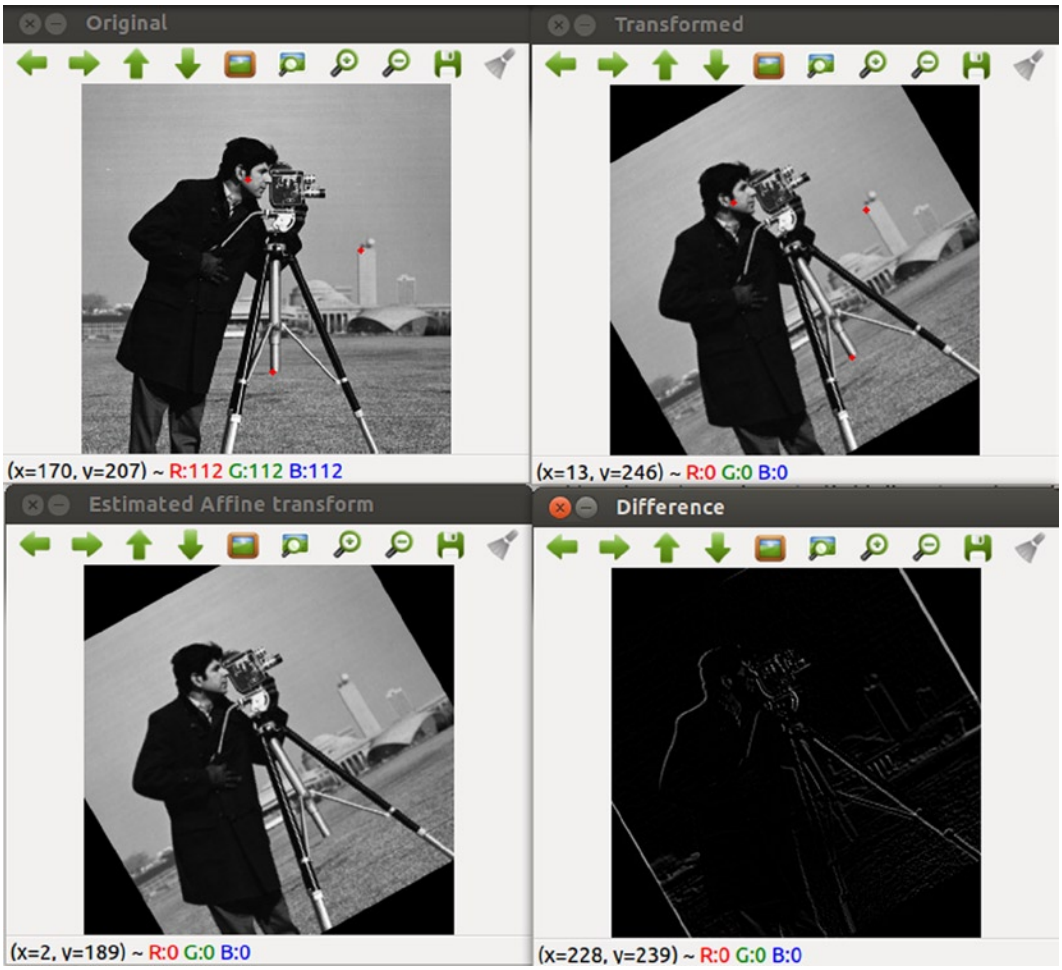


Figure 9-2. *Affine transform recovery using three pairs of matching points*

Perspective Transforms

Perspective transforms are more general than Affine transforms. They do not necessarily preserve the "parallelness" of lines. But because they are more general, they are more practically useful, too—almost all transforms encountered in day-to-day images are perspective transforms. Ever wondered why the two rails seem to meet at a distance? This is because the image plane of your eyes views them at a perspective and perspective transforms do not necessarily keep parallel lines parallel. If you view those rails from above, they will not seem to meet at all.

Given a 3 x 3 perspective transform matrix `M`, `warpPerspective()` applies the following transform:

$$dst(x, y) = src\left(\frac{M11*x+M12*y+M13}{M31*x+M32*y+M33}, \frac{M21*x+M22*y+M23}{M31*x+M32*y+M33}\right)$$

Note that the determinant of the top-left 2 x 2 part of a perspective transform matrix does not need to be +1. Also, because of the division in the transformation shown earlier, multiplying all elements of a perspective transform matrix by a constant does not make any difference in the transform represented. Hence, it is common for perspective transform matrices to be calculated so that M33 = 1. This leaves us with eight free numbers in M, and hence four pairs of corresponding points are enough to recover a perspective transformation between two images. The OpenCV function `findHomography()` does this for you. The interesting fact is that if you specify the flag CV_RANSAC while calling this function (see online documentation), it can even take in more than four points and use the RANSAC algorithm to robustly estimate the transform from all those points. RANSAC makes the transform estimation process immune to noisy "wrong" correspondences. Listing 9-3 reads two images (related by a perspective transform), asks the user to click eight pairs of points, estimates the perspective transform robustly using RANSAC, and shows the difference between the original and new perspectively transformed images to verify the estimated transform. Again, the difference image is mostly black in the relevant area, which means that the estimated transform is correct.

Listing 9-3. Program to illustrate a simple perspective transform recovery and application

```
//Program to illustrate a simple perspective transform recovery and application
//Author: Samarth Manoj Brahmbhatt, University of Pennsyalvania

#include <opencv2/opencv.hpp>
#include <opencv2/stitching/stitcher.hpp>
#include <opencv2/stitching/warpers.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/calib3d/calib3d.hpp>
#include "Config.h"

using namespace std;
using namespace cv;

void on_mouse(int event, int x, int y, int, void* _p) {
    Point2f* p = (Point2f *)_p;
    if (event == CV_EVENT_LBUTTONUP) {
        p->x = x;
        p->y = y;
    }
}

class perspective_transformer {
    private:
        Mat im, im_transformed, im_perspective_transformed, im_show, im_transformed_show;
```

```
        vector<Point2f> points, points_transformed;
        Mat M;
        Point2f get_click(string, Mat);
    public:
        perspective_transformer();
        void estimate_perspective();
        void show_diff();
};

perspective_transformer::perspective_transformer() {
    im = imread(DATA_FOLDER_3 + string("/image.jpg"));
    im_transformed = imread(DATA_FOLDER_3 + string("/transformed.jpg"));
    cout << DATA_FOLDER_3 + string("/transformed.jpg") << endl;
}

Point2f perspective_transformer::get_click(string window_name, Mat im) {
    Point2f p(-1, -1);
    setMouseCallback(window_name, on_mouse, (void *)&p);
    while(p.x == -1 && p.y == -1) {
        imshow(window_name, im);
        waitKey(20);
    }
    return p;
}

void perspective_transformer::estimate_perspective() {
    imshow("Original", im);
    imshow("Transformed", im_transformed);

    cout << "To estimate the Perspective transform between the original and transformed images you
will have to click on 8 matching pairs of points" << endl;

    im_show = im.clone();
    im_transformed_show = im_transformed.clone();
    Point2f p;
    for(int i = 0; i < 8; i++) {
        cout << "POINT " << i << endl;

        cout << "Click on a distinguished point in the ORIGINAL image" << endl;
        p = get_click("Original", im_show);
        cout << p << endl;
        points.push_back(p);
        circle(im_show, p, 2, Scalar(0, 0, 255), -1);
        imshow("Original", im_show);

        cout << "Click on a distinguished point in the TRANSFORMED image" << endl;
        p = get_click("Transformed", im_transformed_show);
        cout << p << endl;
        points_transformed.push_back(p);
        circle(im_transformed_show, p, 2, Scalar(0, 0, 255), -1);
        imshow("Transformed", im_transformed_show);
    }
```

```
    // Estimate perspective transform
    M = findHomography(points, points_transformed, CV_RANSAC, 2);
    cout << "Estimated Perspective transform = " << M << endl;

    // Apply estimated perspecive trasnform
    warpPerspective(im, im_perspective_transformed, M, im.size());
    imshow("Estimated Perspective transform", im_perspective_transformed);
}

void perspective_transformer::show_diff() {
    imshow("Difference", im_transformed - im_perspective_transformed);
}

int main() {
    perspective_transformer a;
    a.estimate_perspective();
    cout << "Press 'd' to show difference, 'q' to end" << endl;
    if(char(waitKey(-1)) == 'd') {
        a.show_diff();
        cout << "Press 'q' to end" << endl;
        if(char(waitKey(-1)) == 'q') return 0;
    }
    else
        return 0;
}
```

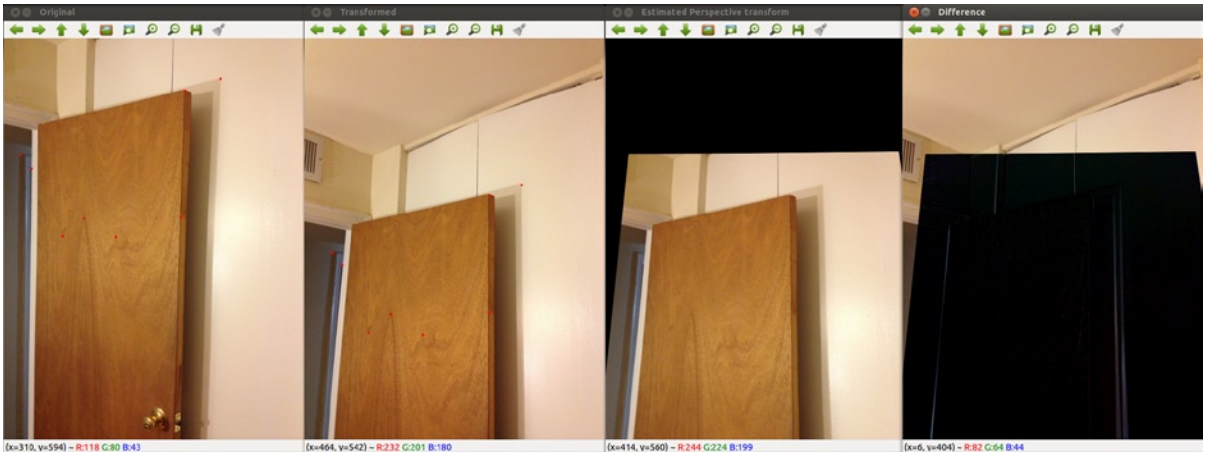


Figure 9-3. *Perspective transform recovery by clicking matching points*

By now, you must have realized that this whole pair-finding process can also be automated by matching image features between the two images with a high distance threshold. This is precisely what Listing 9-4 does. It computes ORB keypoints and descriptors (we learned about this in Chapter 8), matches them, and uses the matches to robustly estimate the perspective transform between the images. Figure 9-4 shows the code in action. Note how the RANSAC makes the transform estimation process robust to the wrong ORB feature match. The difference image is almost black, which means that the estimated transform is correct.

Listing 9-4. Program to illustrate perspective transform recovery by matching ORB features

```
//Program to illustrate perspective transform recovery by matching ORB features
//Author: Samarth Manoj Brahmbhatt, University of Pennsyalvania

#include <opencv2/opencv.hpp>
#include <opencv2/stitching/stitcher.hpp>
#include <opencv2/stitching/warpers.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/calib3d/calib3d.hpp>
#include "Config.h"

using namespace std;
using namespace cv;

class perspective_transformer {
    private:
        Mat im, im_transformed, im_perspective_transformed;
        vector<Point2f> points, points_transformed;
        Mat M;
    public:
        perspective_transformer();
        void estimate_perspective();
        void show_diff();
};

perspective_transformer::perspective_transformer() {
    im = imread(DATA_FOLDER_3 + string("/image.jpg"));
    im_transformed = imread(DATA_FOLDER_3 + string("/transformed.jpg"));
}

void perspective_transformer::estimate_perspective() {
    // Match ORB features to point correspondences between the images
    vector<KeyPoint> kp, t_kp;
    Mat desc, t_desc, im_g, t_im_g;

    cvtColor(im, im_g, CV_BGR2GRAY);
    cvtColor(im_transformed, t_im_g, CV_BGR2GRAY);

    OrbFeatureDetector featureDetector;
    OrbDescriptorExtractor featureExtractor;

    featureDetector.detect(im_g, kp);
    featureDetector.detect(t_im_g, t_kp);

    featureExtractor.compute(im_g, kp, desc);
    featureExtractor.compute(t_im_g, t_kp, t_desc);

    flann::Index flannIndex(desc, flann::LshIndexParams(12, 20, 2), cvflann::FLANN_DIST_HAMMING);
    Mat match_idx(t_desc.rows, 2, CV_32SC1), match_dist(t_desc.rows, 2, CV_32FC1);
    flannIndex.knnSearch(t_desc, match_idx, match_dist, 2, flann::SearchParams());
```

```
    vector<DMatch> good_matches;
    for(int i = 0; i < match_dist.rows; i++) {
        if(match_dist.at<float>(i, 0) < 0.6 * match_dist.at<float>(i, 1)) {
            DMatch dm(i, match_idx.at<int>(i, 0), match_dist.at<float>(i, 0));
            good_matches.push_back(dm);
            points.push_back((kp[dm.trainIdx]).pt);
            points_transformed.push_back((t_kp[dm.queryIdx]).pt);
        }
    }

    Mat im_show;
    drawMatches(im_transformed, t_kp, im, kp, good_matches, im_show);
    imshow("ORB matches", im_show);

    M = findHomography(points, points_transformed, CV_RANSAC, 2);
    cout << "Estimated Perspective transform = " << M << endl;

    warpPerspective(im, im_perspective_transformed, M, im.size());
    imshow("Estimated Perspective transform", im_perspective_transformed);
}

void perspective_transformer::show_diff() {
    imshow("Difference", im_transformed - im_perspective_transformed);
}

int main() {
    perspective_transformer a;
    a.estimate_perspective();
    cout << "Press 'd' to show difference, 'q' to end" << endl;
    if(char(waitKey(-1)) == 'd') {
        a.show_diff();
        cout << "Press 'q' to end" << endl;
        if(char(waitKey(-1)) == 'q') return 0;
    }
    else
        return 0;
}
```

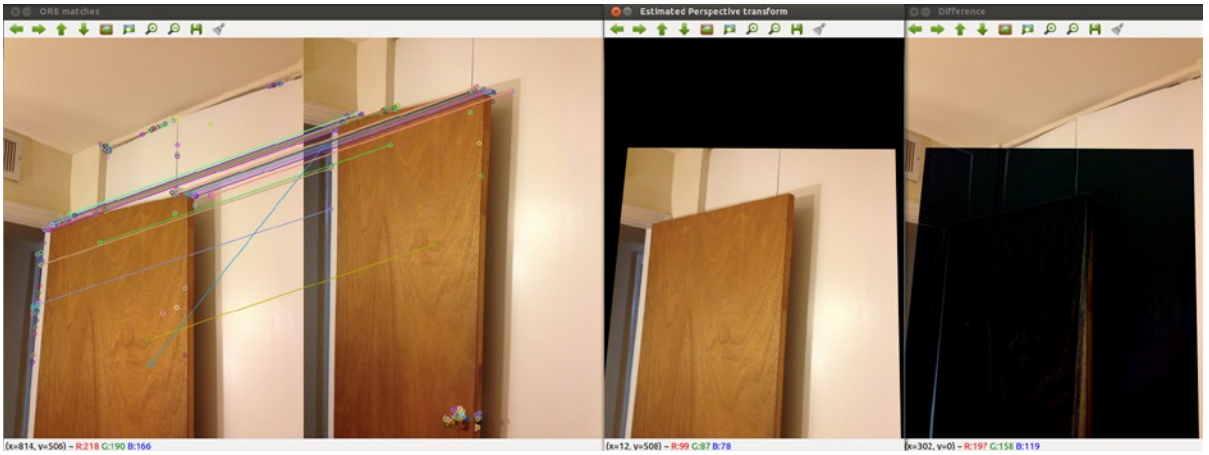


Figure 9-4. *Perspective transform recovery by matching ORB features*

Panoramas

Making panoramas is one of the main applications of automatically recovering perspective transforms. The techniques discussed earlier can be used to estimate the perspective transforms between a set of images captured by a rotating/revolving (but not translating) camera. One can then construct a panorama by "arranging" all these images on a big blank "canvas" image. The arrangement is done according to the estimated perspective transforms. Although this is the high-level algorithm used most commonly to make panoramas, some minor details have to be taken care of in order to make seamless panoramas:

- The estimated perspective transforms are most likely to be not perfect. Hence, if one just goes by the estimated transforms to arrange the images on the canvas, one observes small discontinuities in the regions where two images overlap. Hence after the pair-wise transforms are estimated, a second "global" estimation has to be done, which will perturb the individual transforms to make all the transforms agree well with each other
- Some form of seam blending has to be implemented to remove discontinuities in overlapping regions. Most modern cameras have auto-exposure settings. Hence, different images might have been captures at different exposures and therefore they may be darker or brighter than their neighboring image in the panorama. The difference in exposure must be neutralized across all neighboring images

The OpenCV `stitching` module has all these facilities built in excellently. It uses the high-level algorithm outlined in Figure 9-5 to stitch images into a visually correct panorama.

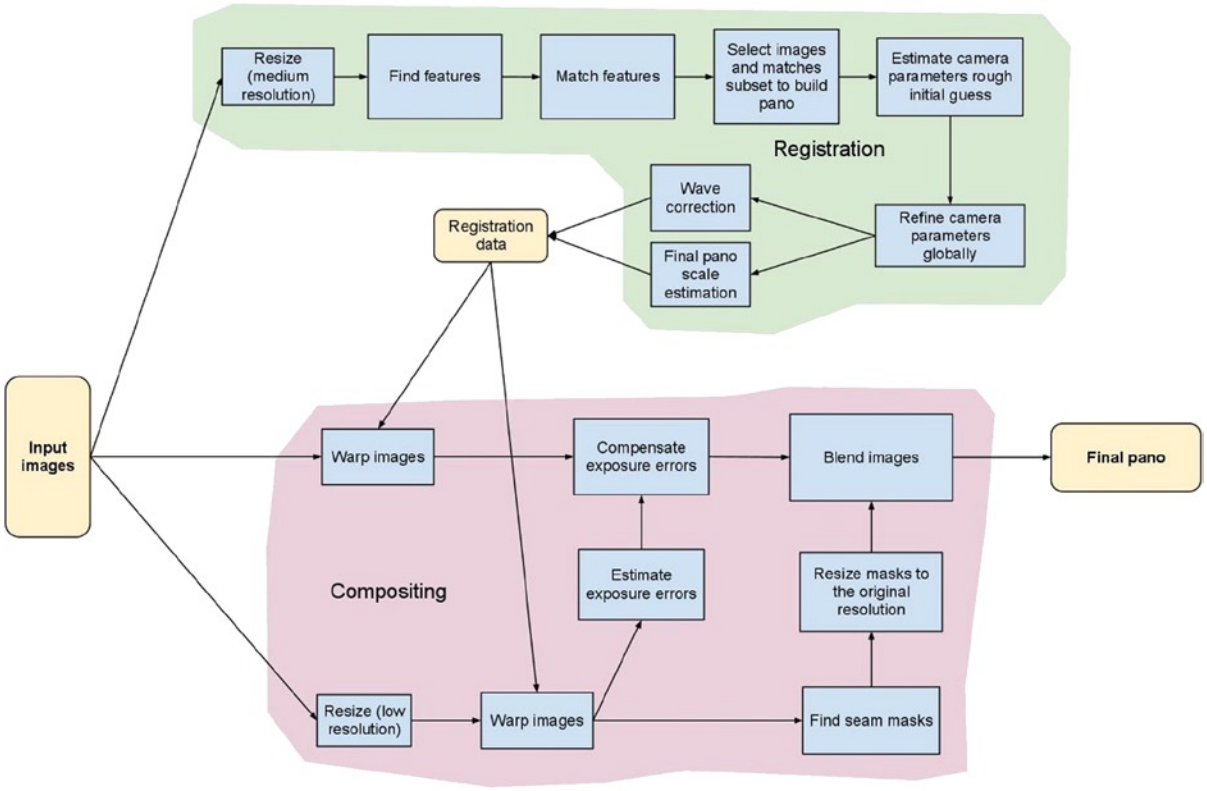


Figure 9-5. *OpenCV image stitching pipeline, taken from OpenCV online documentation*

The `stitching` module is really simple to use in the sense that to make a panorama one just has to create a `stitching` object and pass it a vector of `Mat`'s containing the images you want to stitch. Listing 9-5 shows the simple code used to generate the beautiful panorama out of six images shown in Figure 9-6. Note that this code requires the images to be present in a location called `DATA_FOLDER_1` and defined in the `Config.h` header file. It uses `CMake` to link the executable to the Boost filesystem library. You can use the architecture and `CMake` organization explained towards the end of Chapter 8 to compile the code.

Listing 9-5. Code to create a panorama from a collection of images

```
//Code to create a panorama from a collection of images
//Author: Samarth Manoj Brahmbhatt, University of Pennsyalvania

#include <opencv2/opencv.hpp>
#include <opencv2/stitching/stitcher.hpp>
#include <opencv2/stitching/warpers.hpp>
#include "Config.h"
#include <boost/filesystem.hpp>
```

```
using namespace std;
using namespace cv;
using namespace boost::filesystem;

int main() {
    vector<Mat> images;

    // Read images
    for(directory_iterator i(DATA_FOLDER_5), end_iter; i != end_iter; i++) {
        string im_name = i->path().filename().string();
        string filename = string(DATA_FOLDER_5) + im_name;
        Mat im = imread(filename);
        if(!im.empty())
            images.push_back(im);
    }

    cout << "Read " << images.size() << " images" << endl << "Now making panorama..." << endl;

    Mat panorama;

    Stitcher stitcher = Stitcher::createDefault();
    stitcher.stitch(images, panorama);

    namedWindow("Panorama", CV_WINDOW_NORMAL);
    imshow("Panorama", panorama);

    while(char(waitKey(1)) != 'q') {}

    return 0;
}
```

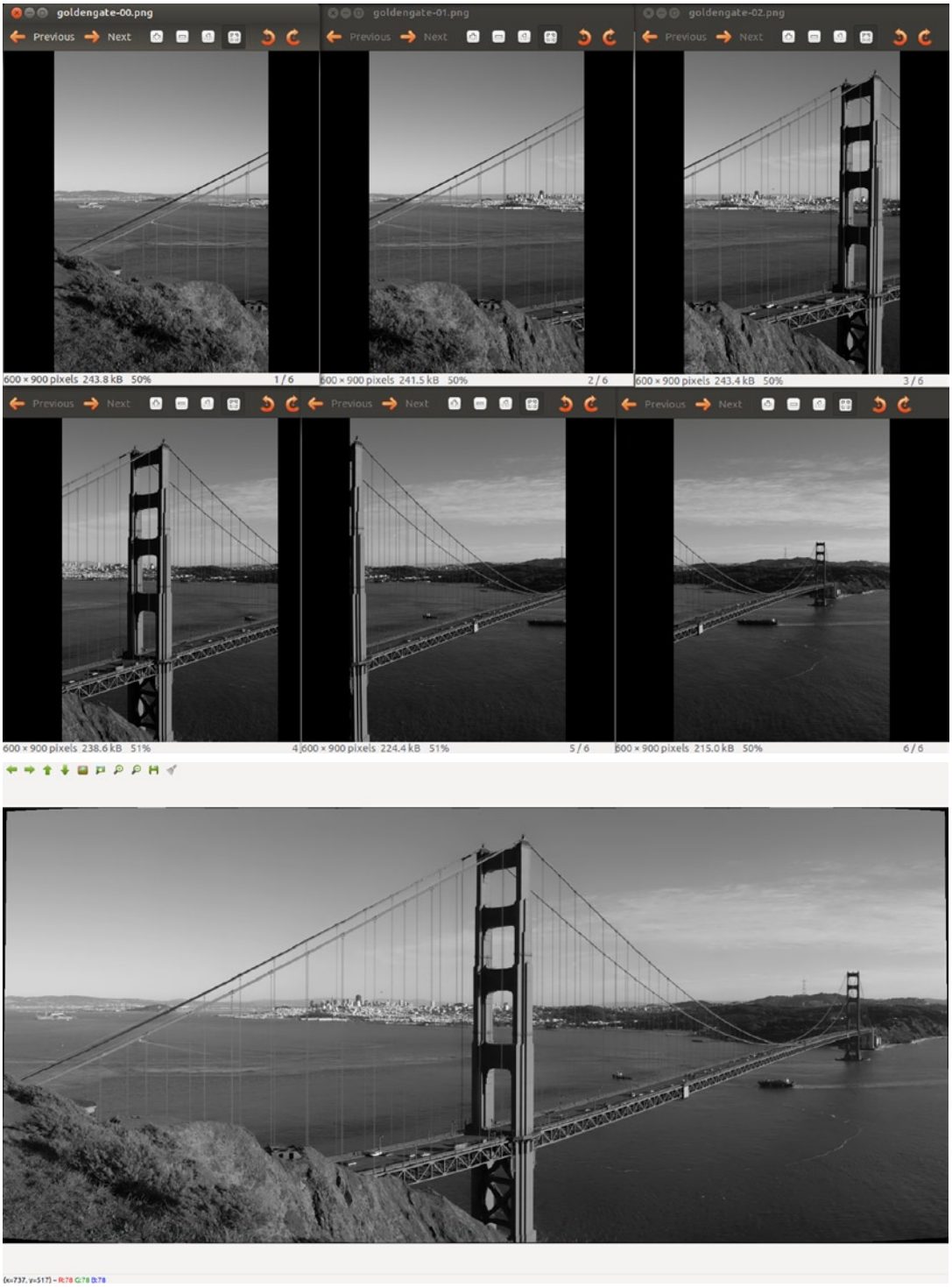


Figure 9-6. *6 images (top) used to generate the Golden Gate panorama (bottom)*

The panorama code scales up quite well too. Figure 9-7 shows the panorama generated from 23 images using the same code.

Figure 9-7. *Panorama made by stitching 23 images*

The online documentation for the stitching module at http://docs.opencv.org/modules/stitching/doc/stitching.html shows that there are a lot of options available for different parts of the pipeline. For example, you can:

- Use SURF or ORB as your choice of image features
- Planar, spherical, or cylindrical as the shape of your panorama (the shape of the canvas on which all the images are arranged)
- Graph-cuts or Voronoi diagrams as a method of finding seam regions that need blending

Listing 9-6 shows how you can plug in and plug out different modules of the pipeline using various "setter" functions of the `stitching` class. It changes the shape of the panorama from the default planar to cylindrical. Figure 9-8 shows the cylindrical panorama thus obtained.

Listing 9-6. Code to create a panorama with cylindrical warping from a collection of images

```
//Code to create a panorama with cylindrical warping from a collection of images
//Author: Samarth Manoj Brahmbhatt, University of Pennsyalvania

#include <opencv2/opencv.hpp>
#include <opencv2/stitching/stitcher.hpp>
#include <opencv2/stitching/warpers.hpp>
#include "Config.h"
#include <boost/filesystem.hpp>
```

```
using namespace std;
using namespace cv;
using namespace boost::filesystem;

int main() {
    vector<Mat> images;

    for(directory_iterator i(DATA_FOLDER_5), end_iter; i != end_iter; i++) {
        string im_name = i->path().filename().string();
        string filename = string(DATA_FOLDER_5) + im_name;
        Mat im = imread(filename);
        if(!im.empty())
            images.push_back(im);
    }

    cout << "Read " << images.size() << " images" << endl << "Now making panorama..." << endl;

    Mat panorama;

    Stitcher stitcher = Stitcher::createDefault();
    CylindricalWarper* warper = new CylindricalWarper();
    stitcher.setWarper(warper);

    // Estimate perspective transforms between images
    Stitcher::Status status = stitcher.estimateTransform(images);
    if (status != Stitcher::OK) {
        cout << "Can't stitch images, error code = " << int(status) << endl;
        return -1;
    }

    // Make panorama
    status = stitcher.composePanorama(panorama);
    if (status != Stitcher::OK) {
        cout << "Can't stitch images, error code = " << int(status) << endl;
        return -1;
    }

    namedWindow("Panorama", CV_WINDOW_NORMAL);
    imshow("Panorama", panorama);

    while(char(waitKey(1)) != 'q') {}

    return 0;
}
```

Figure 9-8. *Cylindrical panorama made from seven images*

Summary

Geometric image transforms are an important part of all computer vision programs that deal with the real world, because there is always a perspective transform between the world and your camera's image plane as well as between the image planes in two positions of the camera. They are also very cool, because they can be used to make panoramas! In this chapter, you learned how to write code implementing perspective transforms and how to recover the transform between two given images, which is a useful skill to have in many practical computer vision projects. The next chapter is on stereo vision. We will use our knowledge of perspective transform matrices to represent the transform between the left and right cameras of a stereo camera, which will be an important step in learning how to calibrate a stereo camera.

CHAPTER 10

3D Geometry and Stereo Vision

This chapter introduces you to the exciting world of the geometry involved behind computer vision using single and multiple cameras. Knowledge of this geometry will enable you to translate image coordinates into actual 3D world coordinates—in short, you will be able to relate position in your images to well-defined physical positions in the world.

The chapter is divided into two main sections—single cameras and stereo (two) cameras. For each section, we will first discuss the associated mathematical concepts and then delve into OpenCV implementations of that math. The section on stereo cameras is especially exciting, since it will enable you to compute full 3D information from a pair of images. So let's explore camera models without further delay!

Single Camera Calibration

As you are probably aware, every sensor needs to be calibrated. Simply put, calibration is the process of relating the sensor's measuring range with the real world quantity that it measures. A camera being a sensor, it needs to be calibrated to give us information in physical units, too. Without calibration, we can only know about objects in their image coordinates, which is not that useful when one wants to use the vision system on a robot that interacts with the real world. For calibrating a camera, it is necessary to first understand the mathematical model of a camera, as shown in Figure 10-1.

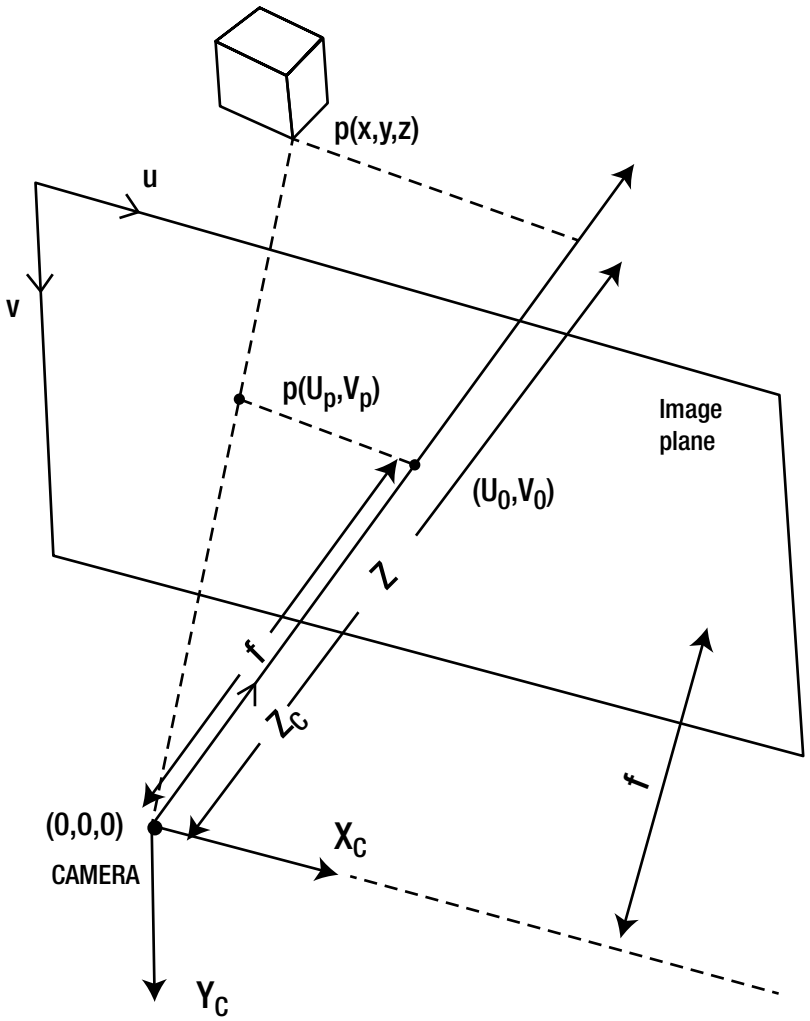


Figure 10-1. *A simple camera model*

It is assumed that rays from real-world objects are focused on the camera point called the projection center using a system of lenses. A 3D coordinate system is shown, with the camera projection center at its origin. This is called the camera coordinate system. Note that all information gained from the camera images will be in this camera coordinate system; if you want to transform this information to any other coordinate frame on your robot, you need to know the rotation and translation from that frame to the camera coordinate frame, and I will not discuss that in this book. The imaging plane of the camera is situated at a distance equal to the focal length *f* along the Z axis. The imaging plane has its own 2D coordinate system (`u, v`) and for the moment let us put its origin at the point (u_0, v_0) where the Z camera axis intersects

the imaging plane. Using the concept of similar triangles it is easy to observe that the relationship between the `(X, Y, Z)` coordinates of an object in the camera coordinate system and the `(u, v)` coordinates of its image is:

$$\frac{u_p}{X} = \frac{f}{Z}$$

$$\frac{v_p}{Y} = \frac{f}{Z}$$

However, in almost all imaging software the origin of an image is placed at the top left corner. Considering this, the relationship becomes:

$$\frac{u_p - u_0}{X} = \frac{f}{Z}$$

$$\frac{v_p - v_0}{Y} = \frac{f}{Z}$$

Calibration of a single camera is the procedure of finding the focal length f and the image (u_0, v_0) of the camera coordinate system origin. From these equations, it should be apparent that even if you know f, u_0, and v_0 by calibrating a camera you cannot determine the Z coordinate of image points and you can only determine the X and Y coordinates up to a scale (because Z is not known). What then, you might argue, is the point of these relations? The point is that these relations (expressed in the form of matrices) allow you to find the camera parameters f, u_0, and v_0 from a bunch of known correspondences between 3D coordinates of an object *in the object's own coordinate frame* and the 2D image coordinates of the points in the object's image. This is done by some ingenious matrix math that I describe briefly later in this chapter.

The camera parameters can be arranged into a camera matrix K as follows:

Suppose `(X, Y, Z)` are the coordinates of the object points in the object's own coordinate system. For calibration,

$$K = \begin{matrix} f & 0 & u_0 \\ 0 & f & v_0 \\ 0 & 0 & 1 \end{matrix}$$

one usually uses a planar checkerboard because it is relatively easy to detect the corners and hence automatically determine the 3D-2D point correspondences. Now, because the choice of the object's coordinate system is in our hands, we will make much life much easier by choosing it such that the board is in the XY plane. Hence, the Z coordinates of all the object points will be 0. If we know the side length of the squares of the checkerboard, X and Y will be just a grid of points with that distance between them. Suppose that R and T are the rotation matrix and translation between the object's coordinate system and the camera coordinate system. This means that once we transform the object coordinates by R and T, we will be able to use the relations we derived from Figure 10-1. R and T are unknown. If we represent the object coordinates by the column vector `[X Y Z 1]`^{T}, the vector `[u1, u2, u3]`^{T} we get from:

$$[u1\,u2\,u3]^T = K\,[RT][X\,Y\,Z\,1]^T$$

can be used to get the image coordinates in pixels. Specifically, pixel coordinates:

$$(u,v) = (\frac{u1}{u3}, \frac{u2}{u3})$$

Now, if you had an algorithm to detect internal corners in the image of a checkerboard, you would have a set of 2D-3D correspondences for each image of the checkerboard. Grouping all these correspondence gives you two equations—one for `u` and one for `v`. The unknowns in these equations are `R, T`, and the camera parameters. If you have a large number of these correspondences, you can solve a huge linear system of equations to get the camera parameters f, u_0, and v_0 and the `R` and `T` for each image.

OpenCV Implementation of Single Camera Calibration

The OpenCV function `findChessboardCorners()` can find the internal corners in images of chessboards.

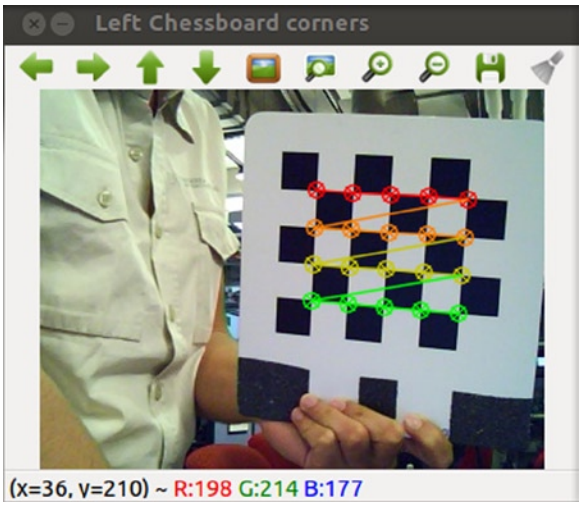


Figure 10-2. *Chessboard corner extraction in OpenCV*

It takes in the image and the size of the pattern (internal corners along width and height) as input and uses a sophisticated algorithm to calculate and return the pixel locations of those corners. These are your 2D image points. You can make a vector of Point3f's for the 3D object points yourself—the `X` and `Y` coordinates are a grid and the `Z` coordinates are all 0, as explained previously. The OpenCV function `calibrateCamera()` takes these object 3D points and corresponding image 2D points and solves the equations to output the camera matrix, `R`, `T` and distortion coefficients. Yes, every camera has a lens distortion, which OpenCV models by using distortion coefficients. Although a discussion on the math behind determining these coefficients is beyond the scope of this book, a good estimation of distortion coefficients can improve the camera calibration a lot. The good news is that if you have a good set of chessboard images with varied views, OpenCV usually does a very good job determining the coefficients.

Listing 10-1 shows an object-oriented approach to read a set of images, find corners, and calibrate a camera. It uses the same `CMake` build system we used in the previous chapter, and a similar folder organization. The images are assumed to be present in a location `IMAGE_FOLDER`. The program also stores the camera matrix and distortion coefficients to an XML file for potential later use.

Listing 10-1. Program to illustrate single camera calibration

```
// Program illustrate single camera calibration
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/calib3d/calib3d.hpp>
#include <boost/filesystem.hpp>
#include "Config.h"

using namespace cv;
using namespace std;
using namespace boost::filesystem3;

class calibrator {
    private:
        string path;                            //path of folder containing chessboard images
        vector<Mat> images;                     //chessboard images
        Mat cameraMatrix, distCoeffs;           //camera matrix and distortion coefficients
        bool show_chess_corners;                //visualize the extracted chessboard corners?
        float side_length;                      //side length of a chessboard square in mm
        int width, height;                      //number of internal corners of the chessboard along
width and height
        vector<vector<Point2f> > image_points;  //2D image points
        vector<vector<Point3f> > object_points; //3D object points

    public:
        calibrator(string, float, int, int);    //constructor, reads in the images
        void calibrate();                       //function to calibrate the camera
        Mat get_cameraMatrix();                 //access the camera matrix
        Mat get_distCoeffs();                   //access the distortion coefficients
        void calc_image_points(bool);           //calculate internal corners of the chessboard image
};

calibrator::calibrator(string _path, float _side_length, int _width, int _height) {
    side_length = _side_length;
    width = _width;
    height = _height;

    path = _path;
    cout << path << endl;

    // Read images
    for(directory_iterator i(path), end_iter; i != end_iter; i++) {
        string filename = path + i->path().filename().string();
        images.push_back(imread(filename));
    }
}
```

```
void calibrator::calc_image_points(bool show) {
    // Calculate the object points in the object co-ordinate system (origin at top left corner)
    vector<Point3f> ob_p;
    for(int i = 0; i < height; i++) {
        for(int j = 0; j < width; j++) {
            ob_p.push_back(Point3f(j * side_length, i * side_length, 0.f));
        }
    }

    if(show) namedWindow("Chessboard corners");

    for(int i = 0; i < images.size(); i++) {
        Mat im = images[i];
        vector<Point2f> im_p;
        //find corners in the chessboard image
        bool pattern_found = findChessboardCorners(im, Size(width, height), im_p,
CALIB_CB_ADAPTIVE_THRESH + CALIB_CB_NORMALIZE_IMAGE+ CALIB_CB_FAST_CHECK);
        if(pattern_found) {
            object_points.push_back(ob_p);
            Mat gray;
            cvtColor(im, gray, CV_BGR2GRAY);
            cornerSubPix(gray, im_p, Size(5, 5), Size(-1, -1), TermCriteria(CV_TERMCRIT_EPS +
CV_TERMCRIT_ITER, 30, 0.1));
            image_points.push_back(im_p);
            if(show) {
                Mat im_show = im.clone();
                drawChessboardCorners(im_show, Size(width, height), im_p, true);
                imshow("Chessboard corners", im_show);
                while(char(waitKey(1)) != ' ') {}
            }
        }
        //if a valid pattern was not found, delete the entry from vector of images
        else images.erase(images.begin() + i);
    }
}

void calibrator::calibrate() {
    vector<Mat> rvecs, tvecs;
    float rms_error = calibrateCamera(object_points, image_points, images[0].size(), cameraMatrix,
distCoeffs, rvecs, tvecs);
    cout << "RMS reprojection error " << rms_error << endl;
}

Mat calibrator::get_cameraMatrix() {
    return cameraMatrix;
}

Mat calibrator::get_distCoeffs() {
    return distCoeffs;
}
```

```
int main() {
    calibrator calib(IMAGE_FOLDER, 25.f, 5, 4);

    calib.calc_image_points(true);
    cout << "Calibrating camera…" << endl;
    calib.calibrate();
    //save the calibration for future use
    string filename = DATA_FOLDER + string("cam_calib.xml");
    FileStorage fs(filename, FileStorage::WRITE);
    fs << "cameraMatrix" << calib.get_cameraMatrix();
    fs << "distCoeffs" << calib.get_distCoeffs();
    fs.release();
    cout << "Saved calibration matrices to " << filename << endl;

    return 0;
}
```

`calibrateCamera()` returns the RMS reprojection error, which should be less than 0.5 pixels for a good calibration. The reprojection error for the camera and set of images I used was 0.0547194. Keep in mind that you must have a set of at least 10 images with the chessboard at various positions and angles but always fully visible.

Stereo Vision

A stereo camera is like your eyes—two cameras separated horizontally by a fixed distance called the baseline. A stereo camera setup allows you to compute even the physical depth of an image point using the concept of disparity. To understand disparity, imagine you are viewing a 3D point through the two cameras of a stereo rig. If the two cameras are not pointing towards each other (and they are usually not), the image of the point in the right image will have a lower horizontal coordinate than the image of the point in the left image. This apparent shift in the image of the point in the two cameras is called disparity. Extending this logic also tells us that disparity is inversely proportional to the depth of the point.

Triangulation

Let us now discuss the relation between depth and disparity in a more mathematical setting, considering the stereo camera model shown in Figure 10-3.

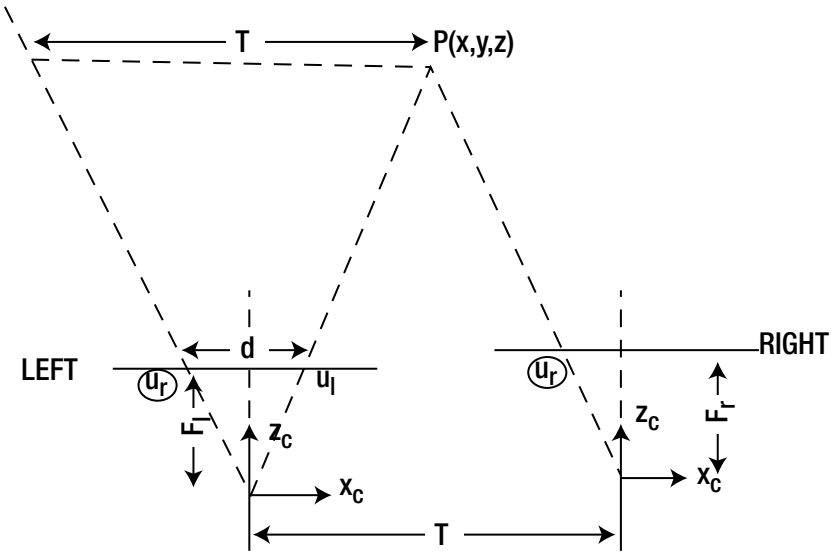


Figure 10-3. *A stereo camera model*

Observe the positions of the image of the point P in the two images. The disparity in this case is:

$$d = ul - ur$$

Using the concept of similar triangles, the depth of the point (Z coordinate of P in the left camera coordinate system) is governed by the expression:

$$\frac{d}{T} = \frac{f}{Z}$$

and hence depth of the point:

$$Z = T * \frac{f}{d}$$

Once you know Z you can calculate the X and Y coordinates of the point exactly by using the equations for single camera model mentioned in the previous section. This whole process is termed "stereo triangulation."

Calibration

To calibrate a stereo camera, you first have to calibrate the cameras individually. Calibration of the stereo rig entails finding out the baseline T. Also notice that I made a strong assumption while drawing Figure 10-3—that the two image planes are exactly aligned vertically and they are parallel to each other. This is usually not the case as a result of small manufacturing defects and there is a definite rotation matrix R that aligns the two imaging planes. Calibration also computes R. T is also not a single number, but a vector representing the translation from the left camera origin to

the right camera origin. The OpenCV function `stereoCalibrate()` takes in calculates `R, T`, and a couple of other matrices called `E` (essential matrix) and `F` (fundamental matrix) by taking the following inputs:

- 3D Object points (same as in the single camera calibration case)
- Left and right 2D image points (computed by using `findCameraCorners()` in the left and right images)
- Left and right camera matrices and distortion coefficients (optional)

Note that the left and right camera calibration information is optional and the function tries to compute it if not provided. But it is highly recommended you provide it so that the function has to optimize less parameters.

If you have a stereo camera attached to your USB port, the port will usually not have enough bandwidth to transfer both left and right 640 x 480 color frames at 30 fps. You can reduce the size of the frames by changing the properties of the `VideoCapture` object associated with the camera as shown in Listing 10-2. Figure 10-4 shows 320 x 240 frames from my stereo camera. For stereo applications, the left and right images must be captured at the same instant. This is possible with hardware synchronization but if your stereo camera does not have that feature, you can first grab the raw frames quickly using the `grab()` method of the `VideoCapture` class and then do the heavier demosaicing, decoding, and `Mat` storage tasks afterward using the `retrieve()` method. This ensures that both frames are captured at almost the same time.

Listing 10-2. Program to illustrate frame capture from a USB stereo camera

```
// Program to illustrate frame capture from a USB stereo camera
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>

using namespace cv;
using namespace std;

int main() {
    VideoCapture capr(1), capl(2);
    //reduce frame size
    capl.set(CV_CAP_PROP_FRAME_HEIGHT, 240);
    capl.set(CV_CAP_PROP_FRAME_WIDTH, 320);
    capr.set(CV_CAP_PROP_FRAME_HEIGHT, 240);
    capr.set(CV_CAP_PROP_FRAME_WIDTH, 320);

    namedWindow("Left");
    namedWindow("Right");

    while(char(waitKey(1)) != 'q') {
        //grab raw frames first
        capl.grab();
        capr.grab();
        //decode later so the grabbed frames are less apart in time
        Mat framel, framer;
        capl.retrieve(framel);
        capr.retrieve(framer);
```

```
        if(framel.empty() || framer.empty()) break;

        imshow("Left", framel);
        imshow("Right", framer);
    }
    capl.release();
    capr.release();
    return 0;
}
```

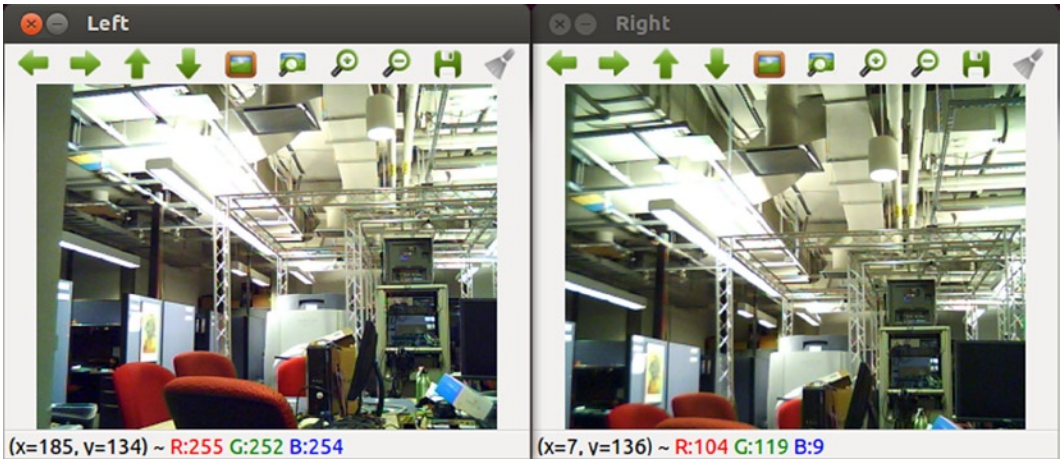


Figure 10-4. *Frames from a stereo camera*

Listing 10-3 is an app that you can use to capture a set of checkerboard images to calibrate your stereo camera. It saves a pair of images when you press "c" into separate folders (called `LEFT_FOLDER` and `RIGHT_FOLDER`). It uses the `CMake` build system and a configuration file similar to the one we have been using in our `CMake` projects.

Listing 10-3. Program to collect stereo snapshots for calibration

```
// Program to collect stereo snapshots for calibration
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include "Config.h"
#include <iomanip>

using namespace cv;
using namespace std;

int main() {
    VideoCapture capr(1), capl(2);
    //reduce frame size
    capl.set(CV_CAP_PROP_FRAME_HEIGHT, 240);
    capl.set(CV_CAP_PROP_FRAME_WIDTH, 320);
```

```
    capr.set(CV_CAP_PROP_FRAME_HEIGHT, 240);
    capr.set(CV_CAP_PROP_FRAME_WIDTH, 320);

    namedWindow("Left");
    namedWindow("Right");

    cout << "Press 'c' to capture ..." << endl;

    char choice = 'z';
    int count = 0;
    while(choice != 'q') {
        //grab frames quickly in succession
        capl.grab();
        capr.grab();
        //execute the heavier decoding operations
        Mat framel, framer;
        capl.retrieve(framel);
        capr.retrieve(framer);

        if(framel.empty() || framer.empty()) break;

        imshow("Left", framel);
        imshow("Right", framer);
        if(choice == 'c') {
            //save files at proper locations if user presses 'c'
            stringstream l_name, r_name;
            l_name << "left" << setw(4) << setfill('0') << count << ".jpg";
            r_name << "right" << setw(4) << setfill('0') << count << ".jpg";
            imwrite(string(LEFT_FOLDER) + l_name.str(), framel);
            imwrite(string(RIGHT_FOLDER) + r_name.str(), framer);
            cout << "Saved set " << count << endl;
            count++;
        }
        choice = char(waitKey(1));
    }
    capl.release();
    capr.release();
    return 0;
}
```

Listing 10-4 is an app that calibrates a stereo camera by reading in previously captured stereo images stored in LEFT_FOLDER and RIGHT_FOLDER, and previously saved individual camera calibration information in DATA_FOLDER. My camera and set of images gave me a RMS reprojection error of 0.377848 pixels.

Listing 10-4. Program to illustrate stereo camera calibration

```
// Program illustrate stereo camera calibration
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/calib3d/calib3d.hpp>
```

```
#include <boost/filesystem.hpp>
#include "Config.h"

using namespace cv;
using namespace std;
using namespace boost::filesystem3;

class calibrator {
    private:
        string l_path, r_path; //path for folders containing left and right checkerboard images
        vector<Mat> l_images, r_images; //left and right checkerboard images
        Mat l_cameraMatrix, l_distCoeffs, r_cameraMatrix, r_distCoeffs; //Mats for holding
individual camera calibration information
        bool show_chess_corners; //visualize checkerboard corner detections?
        float side_length; //side length of checkerboard squares
        int width, height; //number of internal corners in checkerboard along width and height
        vector<vector<Point2f> > l_image_points, r_image_points; //left and right image points
        vector<vector<Point3f> > object_points; //object points (grid)
        Mat R, T, E, F; //stereo calibration information

    public:
        calibrator(string, string, float, int, int); //constructor
        bool calibrate(); //function to calibrate stereo camera
        void calc_image_points(bool); //function to calculae image points by detecting checkerboard
corners
};

calibrator::calibrator(string _l_path, string _r_path, float _side_length, int _width, int _height)
{
    side_length = _side_length;
    width = _width;
    height = _height;

    l_path = _l_path;
    r_path = _r_path;

    // Read images
    for(directory_iterator i(l_path), end_iter; i != end_iter; i++) {
        string im_name = i->path().filename().string();
        string l_filename = l_path + im_name;
        im_name.replace(im_name.begin(), im_name.begin() + 4, string("right"));
        string r_filename = r_path + im_name;
        Mat lim = imread(l_filename), rim = imread(r_filename);
        if(!lim.empty() && !rim.empty()) {
            l_images.push_back(lim);
            r_images.push_back(rim);
        }
    }
}
```

```
void calibrator::calc_image_points(bool show) {
    // Calculate the object points in the object co-ordinate system (origin at top left corner)
    vector<Point3f> ob_p;
    for(int i = 0; i < height; i++) {
        for(int j = 0; j < width; j++) {
            ob_p.push_back(Point3f(j * side_length, i * side_length, 0.f));
        }
    }

    if(show) {
        namedWindow("Left Chessboard corners");
        namedWindow("Right Chessboard corners");
    }

    for(int i = 0; i < l_images.size(); i++) {
        Mat lim = l_images[i], rim = r_images[i];
        vector<Point2f> l_im_p, r_im_p;
        bool l_pattern_found = findChessboardCorners(lim, Size(width, height), l_im_p,
CALIB_CB_ADAPTIVE_THRESH + CALIB_CB_NORMALIZE_IMAGE+ CALIB_CB_FAST_CHECK);
        bool r_pattern_found = findChessboardCorners(rim, Size(width, height), r_im_p,
CALIB_CB_ADAPTIVE_THRESH + CALIB_CB_NORMALIZE_IMAGE+ CALIB_CB_FAST_CHECK);
        if(l_pattern_found && r_pattern_found) {
            object_points.push_back(ob_p);
            Mat gray;
            cvtColor(lim, gray, CV_BGR2GRAY);
            cornerSubPix(gray, l_im_p, Size(5, 5), Size(-1, -1), TermCriteria(CV_TERMCRIT_EPS +
CV_TERMCRIT_ITER, 30, 0.1));
            cvtColor(rim, gray, CV_BGR2GRAY);
            cornerSubPix(gray, r_im_p, Size(5, 5), Size(-1, -1), TermCriteria(CV_TERMCRIT_EPS +
CV_TERMCRIT_ITER, 30, 0.1));
            l_image_points.push_back(l_im_p);
            r_image_points.push_back(r_im_p);
            if(show) {
                Mat im_show = lim.clone();
                drawChessboardCorners(im_show, Size(width, height), l_im_p, true);
                imshow("Left Chessboard corners", im_show);
                im_show = rim.clone();
                drawChessboardCorners(im_show, Size(width, height), r_im_p, true);
                imshow("Right Chessboard corners", im_show);
                while(char(waitKey(1)) != ' ') {}
            }
        }
        else {
            l_images.erase(l_images.begin() + i);
            r_images.erase(r_images.begin() + i);
        }
    }
}
```

```
bool calibrator::calibrate() {
    string filename = DATA_FOLDER + string("left_cam_calib.xml");
    FileStorage fs(filename, FileStorage::READ);
    fs["cameraMatrix"] >> l_cameraMatrix;
    fs["distCoeffs"] >> l_distCoeffs;
    fs.release();

    filename = DATA_FOLDER + string("right_cam_calib.xml");
    fs.open(filename, FileStorage::READ);
    fs["cameraMatrix"] >> r_cameraMatrix;
    fs["distCoeffs"] >> r_distCoeffs;
    fs.release();

    if(!l_cameraMatrix.empty() && !l_distCoeffs.empty() && !r_cameraMatrix.empty() &&
!r_distCoeffs.empty()) {
        double rms = stereoCalibrate(object_points, l_image_points, r_image_points,
l_cameraMatrix, l_distCoeffs, r_cameraMatrix, r_distCoeffs, l_images[0].size(), R, T, E, F);
        cout << "Calibrated stereo camera with a RMS error of " << rms << endl;
        filename = DATA_FOLDER + string("stereo_calib.xml");
        fs.open(filename, FileStorage::WRITE);
        fs << "l_cameraMatrix" << l_cameraMatrix;
        fs << "r_cameraMatrix" << r_cameraMatrix;
        fs << "l_distCoeffs" << l_distCoeffs;
        fs << "r_distCoeffs" << r_distCoeffs;
        fs << "R" << R;
        fs << "T" << T;
        fs << "E" << E;
        fs << "F" << F;
        cout << "Calibration parameters saved to " << filename << endl;
        return true;
    }
    else return false;
}

int main() {
    calibrator calib(LEFT_FOLDER, RIGHT_FOLDER, 1.f, 5, 4);
    calib.calc_image_points(true);
    bool done = calib.calibrate();

    if(!done) cout << "Stereo Calibration not successful because individial calibration matrices
could not be read" << endl;

    return 0;
}
```

Rectification and Disparity by Matching

Recall from the discussion on stereo triangulation that you can determine the disparity of a pixel from a pair of stereo images by finding out which pixel in the right image matches a pixel in the left image, and then taking the difference of their horizontal coordinates. But it is prohibitively difficult to search for pixel matches in the entire right image. How could one optimize the search for the matching pixel?

Theoretically, the matching right pixel for a left pixel will be at the same vertical coordinate as the left pixel (but shifted along the horizontal coordinate inversely proportional to depth). This is because the theoretical stereo camera has its individual cameras offset only horizontally. This greatly simplifies the search—you only have to search the right image in the same row as the row of the left pixel! Practically the two cameras in a stereo rig are not vertically aligned exactly due to manufacturing defects. Calibration solves this problem for us—it computes the rotation R and translation T from left camera to right camera. If we transform our right image by R and T, the two images can be exactly aligned and the single-line search becomes valid.

In OpenCV, the process of aligning the two images (called stereo rectification) is accomplished by functions—`stereoRectify()`, `initUndistortRectifyMap()`, and `remap()`. `stereoRectify()` takes in the calibration information of the individual cameras as well as the stereo rig and gives the following outputs (names of matrices match the online documentation):

- `R1`—Rotation matrix to be applied to left camera image to align it
- `R2`—Rotation matrix to be applied to right camera image to align it
- `P1`—The apparent camera matrix of the aligned left camera appended with a column for the translation to origin of the calibrated coordinate system (coordinate system of the left camera). For the left camera, this is [0 0 0]T
- `P2`—The apparent camera matrix of the aligned right camera appended with a column for the translation to origin of the calibrated coordinate system (coordinate system of the left camera). For the right camera, this is approximately [-T 0 0]T where T is the distance between two camera origins
- `Q`—disparity-to-depth mapping matrix (see documentation of function reprojectImageTo3d() for formula)

Once you get these transformations you have to apply them to the corresponding image. An efficient way to do this is pixel map. Here is how a pixel map works—imagine that the rectified image is a blank canvas of the appropriate size that we want to fill with appropriate pixels from the unrectified image. The pixel map is a matrix of the same size as the rectified image and acts as a lookup table from position in rectified image to position in unrectified image. To know which pixel in the unrectified image one must take to fill in a blank pixel in the rectified image, one just looks up the corresponding entry in the pixel map. Because matrix lookups are very efficient compared to floating point multiplications, pixel maps make the process of aligning images very fast. The OpenCV function `initUndistortRectifyMap()` computes pixels maps by taking the matrices output by `stereoRectify()` as input. The function `remap()` applies the pixel map to an unrectified image to rectify it.

Listing 10-5 shows usage examples of all these functions; you are also encouraged to look up their online documentation to explore various options. Figure 10-5 shows stereo rectification in action—notice how visually corresponding pixels in the two images are now at the same horizontal level.

Listing 10-5. Program to illustrate stereo camera rectification

```
// Program illustrate stereo camera rectification
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/calib3d/calib3d.hpp>
#include <boost/filesystem.hpp>
#include "Config.h"

using namespace cv;
using namespace std;
using namespace boost::filesystem3;
```

```
class calibrator {
    private:
        string l_path, r_path;
        vector<Mat> l_images, r_images;
        Mat l_cameraMatrix, l_distCoeffs, r_cameraMatrix, r_distCoeffs;
        bool show_chess_corners;
        float side_length;
        int width, height;
        vector<vector<Point2f> > l_image_points, r_image_points;
        vector<vector<Point3f> > object_points;
        Mat R, T, E, F;

    public:
        calibrator(string, string, float, int, int);
        void calc_image_points(bool);
        bool calibrate();
        void save_info(string);
        Size get_image_size();
};

class rectifier {
    private:
        Mat map_l1, map_l2, map_r1, map_r2; //pixel maps for rectification
        string path;
    public:
        rectifier(string, Size); //constructor
        void show_rectified(Size); //function to show live rectified feed from stereo camera
};

calibrator::calibrator(string _l_path, string _r_path, float _side_length, int _width, int _height)
{
    side_length = _side_length;
    width = _width;
    height = _height;

    l_path = _l_path;
    r_path = _r_path;

    // Read images
    for(directory_iterator i(l_path), end_iter; i != end_iter; i++) {
        string im_name = i->path().filename().string();
        string l_filename = l_path + im_name;
        im_name.replace(im_name.begin(), im_name.begin() + 4, string("right"));
        string r_filename = r_path + im_name;
        Mat lim = imread(l_filename), rim = imread(r_filename);
        if(!lim.empty() && !rim.empty()) {
            l_images.push_back(lim);
            r_images.push_back(rim);
        }
    }
}
```

```
void calibrator::calc_image_points(bool show) {
    // Calculate the object points in the object co-ordinate system (origin at top left corner)
    vector<Point3f> ob_p;
    for(int i = 0; i < height; i++) {
        for(int j = 0; j < width; j++) {
            ob_p.push_back(Point3f(j * side_length, i * side_length, 0.f));
        }
    }

    if(show) {
        namedWindow("Left Chessboard corners");
        namedWindow("Right Chessboard corners");
    }

    for(int i = 0; i < l_images.size(); i++) {
        Mat lim = l_images[i], rim = r_images[i];
        vector<Point2f> l_im_p, r_im_p;
        bool l_pattern_found = findChessboardCorners(lim, Size(width, height), l_im_p,
CALIB_CB_ADAPTIVE_THRESH + CALIB_CB_NORMALIZE_IMAGE+ CALIB_CB_FAST_CHECK);
        bool r_pattern_found = findChessboardCorners(rim, Size(width, height), r_im_p,
CALIB_CB_ADAPTIVE_THRESH + CALIB_CB_NORMALIZE_IMAGE+ CALIB_CB_FAST_CHECK);
        if(l_pattern_found && r_pattern_found) {
            object_points.push_back(ob_p);
            Mat gray;
            cvtColor(lim, gray, CV_BGR2GRAY);
            cornerSubPix(gray, l_im_p, Size(5, 5), Size(-1, -1), TermCriteria(CV_TERMCRIT_EPS +
CV_TERMCRIT_ITER, 30, 0.1));
            cvtColor(rim, gray, CV_BGR2GRAY);
            cornerSubPix(gray, r_im_p, Size(5, 5), Size(-1, -1), TermCriteria(CV_TERMCRIT_EPS +
CV_TERMCRIT_ITER, 30, 0.1));
            l_image_points.push_back(l_im_p);
            r_image_points.push_back(r_im_p);
            if(show) {
                Mat im_show = lim.clone();
                drawChessboardCorners(im_show, Size(width, height), l_im_p, true);
                imshow("Left Chessboard corners", im_show);
                im_show = rim.clone();
                drawChessboardCorners(im_show, Size(width, height), r_im_p, true);
                imshow("Right Chessboard corners", im_show);
                while(char(waitKey(1)) != ' ') {}
            }
        }
        else {
            l_images.erase(l_images.begin() + i);
            r_images.erase(r_images.begin() + i);
        }
    }
}
```

```
bool calibrator::calibrate() {
    string filename = DATA_FOLDER + string("left_cam_calib.xml");
    FileStorage fs(filename, FileStorage::READ);
    fs["cameraMatrix"] >> l_cameraMatrix;
    fs["distCoeffs"] >> l_distCoeffs;
    fs.release();

    filename = DATA_FOLDER + string("right_cam_calib.xml");
    fs.open(filename, FileStorage::READ);
    fs["cameraMatrix"] >> r_cameraMatrix;
    fs["distCoeffs"] >> r_distCoeffs;
    fs.release();

    if(!l_cameraMatrix.empty() && !l_distCoeffs.empty() && !r_cameraMatrix.empty() &&
!r_distCoeffs.empty()) {
        double rms = stereoCalibrate(object_points, l_image_points, r_image_points, l_cameraMatrix,
l_distCoeffs, r_cameraMatrix, r_distCoeffs, l_images[0].size(), R, T, E, F);
        cout << "Calibrated stereo camera with a RMS error of " << rms << endl;
        return true;
    }
    else return false;
}

void calibrator::save_info(string filename) {
    FileStorage fs(filename, FileStorage::WRITE);
    fs << "l_cameraMatrix" << l_cameraMatrix;
    fs << "r_cameraMatrix" << r_cameraMatrix;
    fs << "l_distCoeffs" << l_distCoeffs;
    fs << "r_distCoeffs" << r_distCoeffs;
    fs << "R" << R;
    fs << "T" << T;
    fs << "E" << E;
    fs << "F" << F;
    fs.release();
    cout << "Calibration parameters saved to " << filename << endl;
}

Size calibrator::get_image_size() {
    return l_images[0].size();
}

rectifier::rectifier(string filename, Size image_size) {
    // Read individual camera calibration information from saved XML file
    Mat l_cameraMatrix, l_distCoeffs, r_cameraMatrix, r_distCoeffs, R, T;
    FileStorage fs(filename, FileStorage::READ);
    fs["l_cameraMatrix"] >> l_cameraMatrix;
    fs["l_distCoeffs"] >> l_distCoeffs;
    fs["r_cameraMatrix"] >> r_cameraMatrix;
    fs["r_distCoeffs"] >> r_distCoeffs;
    fs["R"] >> R;
    fs["T"] >> T;
```

```
    fs.release();

    if(l_cameraMatrix.empty() || r_cameraMatrix.empty() || l_distCoeffs.empty() ||
r_distCoeffs.empty() || R.empty() || T.empty())
        cout << "Rectifier: Loading of files not successful" << endl;

    // Calculate transforms for rectifying images
    Mat Rl, Rr, Pl, Pr, Q;
    stereoRectify(l_cameraMatrix, l_distCoeffs, r_cameraMatrix, r_distCoeffs, image_size, R, T, Rl,
Rr, Pl, Pr, Q);

    // Calculate pixel maps for efficient rectification of images via lookup tables
    initUndistortRectifyMap(l_cameraMatrix, l_distCoeffs, Rl, Pl, image_size, CV_16SC2, map_l1,
map_l2);
    initUndistortRectifyMap(r_cameraMatrix, r_distCoeffs, Rr, Pr, image_size, CV_16SC2, map_r1,
map_r2);

    fs.open(filename, FileStorage::APPEND);
    fs << "Rl" << Rl;
    fs << "Rr" << Rr;
    fs << "Pl" << Pl;
    fs << "Pr" << Pr;
    fs << "Q" << Q;
    fs << "map_l1" << map_l1;
    fs << "map_l2" << map_l2;
    fs << "map_r1" << map_r1;
    fs << "map_r2" << map_r2;
    fs.release();
}

void rectifier::show_rectified(Size image_size) {
    VideoCapture capr(1), capl(2);
    //reduce frame size
    capl.set(CV_CAP_PROP_FRAME_HEIGHT, image_size.height);
    capl.set(CV_CAP_PROP_FRAME_WIDTH, image_size.width);
    capr.set(CV_CAP_PROP_FRAME_HEIGHT, image_size.height);
    capr.set(CV_CAP_PROP_FRAME_WIDTH, image_size.width);

    destroyAllWindows();
    namedWindow("Combo");
    while(char(waitKey(1)) != 'q') {
        //grab raw frames first
        capl.grab();
        capr.grab();
        //decode later so the grabbed frames are less apart in time
        Mat framel, framel_rect, framer, framer_rect;
        capl.retrieve(framel);
        capr.retrieve(framer);

        if(framel.empty() || framer.empty()) break;
```

```
        // Remap images by pixel maps to rectify
        remap(framel, framel_rect, map_l1, map_l2, INTER_LINEAR);
        remap(framer, framer_rect, map_r1, map_r2, INTER_LINEAR);

        // Make a larger image containing the left and right rectified images side-by-side
        Mat combo(image_size.height, 2 * image_size.width, CV_8UC3);
        framel_rect.copyTo(combo(Range::all(), Range(0, image_size.width)));
        framer_rect.copyTo(combo(Range::all(), Range(image_size.width, 2*image_size.width)));

        // Draw horizontal red lines in the combo image to make comparison easier
        for(int y = 0; y < combo.rows; y += 20)
            line(combo, Point(0, y), Point(combo.cols, y), Scalar(0, 0, 255));

        imshow("Combo", combo);
    }
    capl.release();
    capr.release();
}

int main() {
    string filename = DATA_FOLDER + string("stereo_calib.xml");
    /*
    calibrator calib(LEFT_FOLDER, RIGHT_FOLDER, 25.f, 5, 4);
    calib.calc_image_points(true);
    bool done = calib.calibrate();
    if(!done) cout << "Stereo Calibration not successful because individial calibration matrices
could not be read" << endl;
    calib.save_info(filename);
    Size image_size = calib.get_image_size();
    */

    Size image_size(320, 240);
    rectifier rec(filename, image_size);
    rec.show_rectified(image_size);

    return 0;
}
```

Figure 10-5. *Stereo rectification*

Now it becomes quite easy to get matching pixels in the two images and calculate the disparity. OpenCV implements the Semi-Global Block Matching (SGBM) algorithm in the `StereoSGBM` class. This algorithm uses a technique called dynamic programming to make the matching more robust. The algorithm has a bunch of parameters associated with it, chiefly:

- `SADWindowSize`: The size of block to match (must be an odd number)
- `P1` and `P2`: Parameters controlling the smoothness of the calculated disparity. They are 'penalties' incurred by the dynamic programming matching algorithm if disparity changes between two neighboring pixels by +/- 1 or more than 1 respectively
- `speckleWindowSize` and `speckleRange`: Disparities often have speckles. These parameters control the disparity speckle filter. `speckleWindowSize` indicates the maximum size of a smooth disparity region for it to be considered as a speckle, while `speckleRange` indicates
- `minDisparity` and `numberOfDisparities`: Together, these parameters control the 'range' of your stereo setup. As mentioned before, the location of a matching pixel in the right image is to the left of the location of the corresponding pixel in the left image. If `minDisparity` is 0 the algorithm starts to search in the right image for matching pixels from location of the pixel in the left image and proceeds left. If `minDisparity` is positive, the algorithm starts searching in the right image from left of the location in the left image (by `minDisparity` pixels) and then proceeds left. You would want this when the two cameras are pointing away from each other. You can also set `minDisparity` to be negative for when the two cameras are pointing towards each other. Thus the start location of the search is determined by `minDisparity`. However, the search always proceeds left in the right image from its starting position. How far does it go? This is decided by `numberOfDisparities`. Note that `numberOfDisparities` must be a multiple of 16 for OpenCV implementation

Listing 10-6 shows how to calculate disparities from two rectified images using the `StereoSGBM` class. It also allows you to experiment with the values of `minDisparity` and `numberOfDisparities` using sliders. Note the conversion of disparity computed by `StereoSGBM` to a visible form, since the original disparity output by `StereoSGBM` is scaled by 16 (read online documentation of `StereoSGBM`). The program uses values of some of the parameters of the stereo algorithm from the OpenCV `stereo_match` demo code. Figure 10-6 shows disparity for a particular scene.

Listing 10-6. Program to illustrate disparity calculation from a calibrated stereo camera

```
// Program illustrate disparity calculation from a calibrated stereo camera
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/calib3d/calib3d.hpp>
#include "Config.h"

using namespace cv;
using namespace std;

class disparity {
    private:
        Mat map_l1, map_l2, map_r1, map_r2; // rectification pixel maps
        StereoSGBM stereo;                  // stereo matching object for disparity computation
        int min_disp, num_disp;             // parameters of StereoSGBM
    public:
        disparity(string, Size);            //constructor
        void set_minDisp(int minDisp) { stereo.minDisparity = minDisp; }
        void set_numDisp(int numDisp) { stereo.numberOfDisparities = numDisp; }
        void show_disparity(Size);          // show live disparity by processing stereo camera feed
};

// Callback functions for minDisparity and numberOfDisparities trackbars
void on_minDisp(int min_disp, void * _disp_obj) {
    disparity * disp_obj = (disparity *) _disp_obj;
    disp_obj -> set_minDisp(min_disp - 30);
}

void on_numDisp(int num_disp, void * _disp_obj) {
    disparity * disp_obj = (disparity *) _disp_obj;
    num_disp = (num_disp / 16) * 16;
    setTrackbarPos("numDisparity", "Disparity", num_disp);
    disp_obj -> set_numDisp(num_disp);
}

disparity::disparity(string filename, Size image_size) {
    // Read pixel maps from XML file
    FileStorage fs(filename, FileStorage::READ);
    fs["map_l1"] >> map_l1;
    fs["map_l2"] >> map_l2;
    fs["map_r1"] >> map_r1;
    fs["map_r2"] >> map_r2;

    if(map_l1.empty() || map_l2.empty() || map_r1.empty() || map_r2.empty())
        cout << "WARNING: Loading of mapping matrices not successful" << endl;

    // Set SGBM parameters (from OpenCV stereo_match.cpp demo)
    stereo.preFilterCap = 63;
    stereo.SADWindowSize = 3;
    stereo.P1 = 8 * 3 * stereo.SADWindowSize * stereo.SADWindowSize;
    stereo.P2 = 32 * 3 * stereo.SADWindowSize * stereo.SADWindowSize;
```

```
    stereo.uniquenessRatio = 10;
    stereo.speckleWindowSize = 100;
    stereo.speckleRange = 32;
    stereo.disp12MaxDiff = 1;
    stereo.fullDP = true;
}

void disparity::show_disparity(Size image_size) {
    VideoCapture capr(1), capl(2);
    //reduce frame size
    capl.set(CV_CAP_PROP_FRAME_HEIGHT, image_size.height);
    capl.set(CV_CAP_PROP_FRAME_WIDTH, image_size.width);
    capr.set(CV_CAP_PROP_FRAME_HEIGHT, image_size.height);
    capr.set(CV_CAP_PROP_FRAME_WIDTH, image_size.width);

    min_disp = 30;
    num_disp = ((image_size.width / 8) + 15) & -16;

    namedWindow("Disparity", CV_WINDOW_NORMAL);
    namedWindow("Left", CV_WINDOW_NORMAL);
    createTrackbar("minDisparity + 30", "Disparity", &min_disp, 60, on_minDisp, (void *)this);
    createTrackbar("numDisparity", "Disparity", &num_disp, 150, on_numDisp, (void *)this);

    on_minDisp(min_disp, this);
    on_numDisp(num_disp, this);

    while(char(waitKey(1)) != 'q') {
        //grab raw frames first
        capl.grab();
        capr.grab();
        //decode later so the grabbed frames are less apart in time
        Mat framel, framel_rect, framer, framer_rect;
        capl.retrieve(framel);
        capr.retrieve(framer);

        if(framel.empty() || framer.empty()) break;

        remap(framel, framel_rect, map_l1, map_l2, INTER_LINEAR);
        remap(framer, framer_rect, map_r1, map_r2, INTER_LINEAR);

        // Calculate disparity
        Mat disp, disp_show;
        stereo(framel_rect, framer_rect, disp);
        // Convert disparity to a form easy for visualization
        disp.convertTo(disp_show, CV_8U, 255/(stereo.numberOfDisparities * 16.));
        imshow("Disparity", disp_show);
        imshow("Left", framel);
    }
    capl.release();
    capr.release();
}
```

```
int main() {
    string filename = DATA_FOLDER + string("stereo_calib.xml");

    Size image_size(320, 240);
    disparity disp(filename, image_size);
    disp.show_disparity(image_size);

    return 0;
}
```

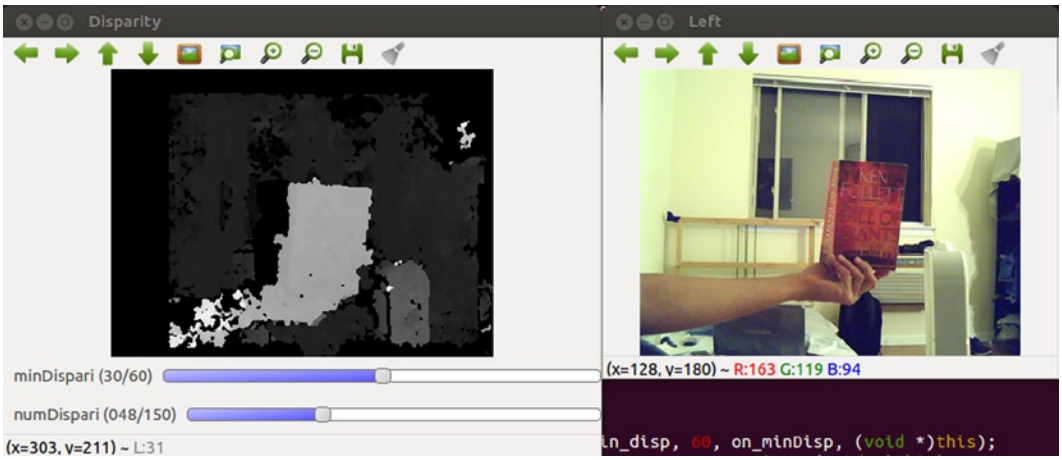


Figure 10-6. *Stereo disparity*

How do you get depth from disparity? The function `reprojectImageTo3d()` does this by taking the disparity and the disparity-to-depth mapping matrix `Q` generated by `stereoRectify()`. Listing 10-7 shows a simple proof-of-concept app that calculates 3D coordinates from disparity and prints out the mean of the depths of points inside a rectangle at the center of the image. As you can see from Figure 10-7, the computed distance jumps around a bit, but is quite near the correct value of 400 mm most of the times. Recall that the disparity output by `StereoSGBM` is scaled by 16, so we must divide it by 16 to get the true disparity.

Listing 10-7. Program to illustrate distance measurement using a stereo camera

```
// Program illustrate distance measurement using a stereo camera
// Author: Samarth Manoj Brahmbhatt, University of Pennsylvania

#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/calib3d/calib3d.hpp>
#include "Config.h"

using namespace cv;
using namespace std;

class disparity {
    private:
        Mat map_l1, map_l2, map_r1, map_r2, Q;
        StereoSGBM stereo;
        int min_disp, num_disp;
```

```
    public:
        disparity(string, Size);
        void set_minDisp(int minDisp) { stereo.minDisparity = minDisp; }
        void set_numDisp(int numDisp) { stereo.numberOfDisparities = numDisp; }
        void show_disparity(Size);
};

void on_minDisp(int min_disp, void * _disp_obj) {
    disparity * disp_obj = (disparity *) _disp_obj;
    disp_obj -> set_minDisp(min_disp - 30);
}

void on_numDisp(int num_disp, void * _disp_obj) {
    disparity * disp_obj = (disparity *) _disp_obj;
    num_disp = (num_disp / 16) * 16;
    setTrackbarPos("numDisparity", "Disparity", num_disp);
    disp_obj -> set_numDisp(num_disp);
}

disparity::disparity(string filename, Size image_size) {
    FileStorage fs(filename, FileStorage::READ);
    fs["map_l1"] >> map_l1;
    fs["map_l2"] >> map_l2;
    fs["map_r1"] >> map_r1;
    fs["map_r2"] >> map_r2;
    fs["Q"] >> Q;

    if(map_l1.empty() || map_l2.empty() || map_r1.empty() || map_r2.empty() || Q.empty())
        cout << "WARNING: Loading of mapping matrices not successful" << endl;

    stereo.preFilterCap = 63;
    stereo.SADWindowSize = 3;
    stereo.P1 = 8 * 3 * stereo.SADWindowSize * stereo.SADWindowSize;
    stereo.P2 = 32 * 3 * stereo.SADWindowSize * stereo.SADWindowSize;
    stereo.uniquenessRatio = 10;
    stereo.speckleWindowSize = 100;
    stereo.speckleRange = 32;
    stereo.disp12MaxDiff = 1;
    stereo.fullDP = true;
}

void disparity::show_disparity(Size image_size) {
    VideoCapture capr(1), capl(2);
    //reduce frame size
    capl.set(CV_CAP_PROP_FRAME_HEIGHT, image_size.height);
    capl.set(CV_CAP_PROP_FRAME_WIDTH, image_size.width);
    capr.set(CV_CAP_PROP_FRAME_HEIGHT, image_size.height);
    capr.set(CV_CAP_PROP_FRAME_WIDTH, image_size.width);

    min_disp = 30;
    num_disp = ((image_size.width / 8) + 15) & -16;

    namedWindow("Disparity", CV_WINDOW_NORMAL);
    namedWindow("Left", CV_WINDOW_NORMAL);
```

```
    createTrackbar("minDisparity + 30", "Disparity", &min_disp, 60, on_minDisp, (void *)this);
    createTrackbar("numDisparity", "Disparity", &num_disp, 150, on_numDisp, (void *)this);

    on_minDisp(min_disp, this);
    on_numDisp(num_disp, this);

    while(char(waitKey(1)) != 'q') {
        //grab raw frames first
        capl.grab();
        capr.grab();
        //decode later so the grabbed frames are less apart in time
        Mat framel, framel_rect, framer, framer_rect;
        capl.retrieve(framel);
        capr.retrieve(framer);

        if(framel.empty() || framer.empty()) break;

        remap(framel, framel_rect, map_l1, map_l2, INTER_LINEAR);
        remap(framer, framer_rect, map_r1, map_r2, INTER_LINEAR);

        Mat disp, disp_show, disp_compute, pointcloud;
        stereo(framel_rect, framer_rect, disp);
        disp.convertTo(disp_show, CV_8U, 255/(stereo.numberOfDisparities * 16.));
        disp.convertTo(disp_compute, CV_32F, 1.f/16.f);

        // Calculate 3D co-ordinates from disparity image
        reprojectImageTo3D(disp_compute, pointcloud, Q, true);

        // Draw red rectangle around 40 px wide square area im image
        int xmin = framel.cols/2 - 20, xmax = framel.cols/2 + 20, ymin = framel.rows/2 - 20,
ymax = framel.rows/2 + 20;
        rectangle(framel_rect, Point(xmin, ymin), Point(xmax, ymax), Scalar(0, 0, 255));

        // Extract depth of 40 px rectangle and print out their mean
        pointcloud = pointcloud(Range(ymin, ymax), Range(xmin, xmax));
        Mat z_roi(pointcloud.size(), CV_32FC1);
        int from_to[] = {2, 0};
        mixChannels(&pointcloud, 1, &z_roi, 1, from_to, 1);

        cout << "Depth: " << mean(z_roi) << " mm" << endl;

        imshow("Disparity", disp_show);
        imshow("Left", framel_rect);
    }
    capl.release();
    capr.release();
}

int main() {
    string filename = DATA_FOLDER + string("stereo_calib.xml");

    Size image_size(320, 240);
    disparity disp(filename, image_size);
    disp.show_disparity(image_size);

    return 0;
}
```

```
Depth: [383.094, 0, 0, 0] mm
Depth: [419.09, 0, 0, 0] mm
Depth: [389.231, 0, 0, 0] mm
Depth: [383.326, 0, 0, 0] mm
Depth: [382.844, 0, 0, 0] mm
Depth: [406.998, 0, 0, 0] mm
Depth: [406.908, 0, 0, 0] mm
Depth: [383.047, 0, 0, 0] mm
Depth: [389.008, 0, 0, 0] mm
Depth: [382.6, 0, 0, 0] mm
Depth: [383.25, 0, 0, 0] mm
Depth: [418.716, 0, 0, 0] mm
Depth: [382.889, 0, 0, 0] mm
Depth: [509.128, 0, 0, 0] mm
Depth: [425.331, 0, 0, 0] mm
Depth: [395.165, 0, 0, 0] mm
Depth: [382.728, 0, 0, 0] mm
Depth: [382.83, 0, 0, 0] mm
Depth: [437.278, 0, 0, 0] mm
Depth: [419.165, 0, 0, 0] mm
Depth: [382.978, 0, 0, 0] mm
Depth: [383.007, 0, 0, 0] mm
Depth: [436.999, 0, 0, 0] mm
Depth: [400.938, 0, 0, 0] mm
Depth: [382.981, 0, 0, 0] mm
Depth: [382.826, 0, 0, 0] mm
Depth: [382.804, 0, 0, 0] mm
Depth: [401.1, 0, 0, 0] mm
Depth: [401.267, 0, 0, 0] mm
Depth: [382.796, 0, 0, 0] mm
Depth: [635.448, 0, 0, 0] mm
Depth: [383.131, 0, 0, 0] mm
Depth: [419.259, 0, 0, 0] mm
Depth: [382.978, 0, 0, 0] mm
Depth: [407.482, 0, 0, 0] mm
Depth: [382.702, 0, 0, 0] mm
```

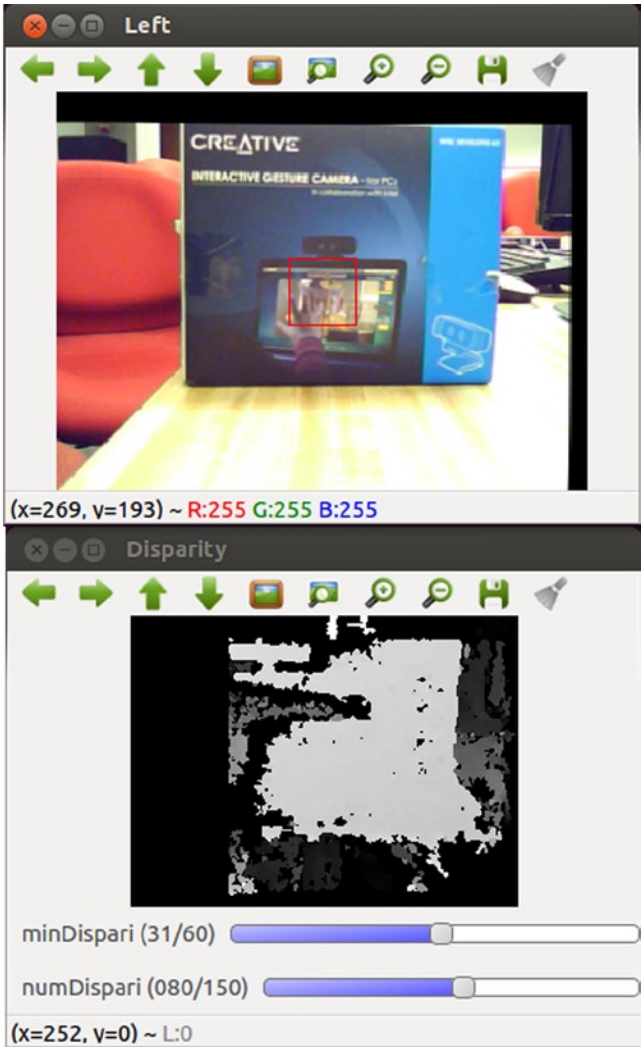


Figure 10-7. *Depth measurement using stereo vision*

Summary

Isn't 3D geometry exciting? If, like me, you are fascinated by how a bunch of matrices can empower you to measure actual distances in physical images from a pair of photographs you should read *Multiple View Geometry in Computer Vision* by Richard Hartley and Andrew Zisserman (Cambridge University Press, 2004), a classic on this subject. Stereo vision opens up a lot of possibilities for you—and most important, it allows you to establish a direct relationship between the output of your other computer vision algorithms and the real world! For example, you could combine stereo vision with the object detector app to reliably know the distance in physical units to the detected object.

The next chapter will take you one step closer to deploying your computer vision algorithms on actual mobile robots—it discusses details of running OpenCV apps on Raspberry Pi, which is a versatile little (but powerful) microcontroller that is becoming increasingly famous for embedded applications.

CHAPTER 11

Embedded Computer Vision: Running OpenCV Programs on the Raspberry Pi

Embedded computer vision is a very practical branch of computer vision that concerns itself with developing or modifying vision algorithms to run on embedded systems—small mobile computers like smartphone processors or hobby boards. The two main considerations in an embedded computer vision system are judicious use of processing power and low battery consumption. As an example of an embedded computing system, we will consider the Raspberry Pi (Figure 11-1), an ARM processor–based small open-source computer that is rapidly gaining popularity among hobbyists and researchers alike for its ease of use, versatile capabilities, and surprisingly low cost in spite of good build and support quality.

Figure 11-1. *The Raspberry Pi board*

Raspberry Pi

The Raspberry Pi is a small computer that can run a Linux-based operating system from a SD memory card. It has a 700 MHz ARM processor and a small Broadcom VideoCore IV 250 MHz GPU. The CPU and GPU share 512 MB of SDRAM memory, and you can change the sharing of memory between each according to your use pattern. As shown in Figure 11-1, the Pi has one Ethernet, one HDMI, two USB 2.0 ports, 8 general-purpose input/output pins, and a UART to interact with other devices. A 5 MP camera board has also been recently released to promote the use of Pi in small-scale computer vision applications. This camera board has its own special parallel connector to the board; hence, it is expected to support higher frame-rates than a web camera attached to one of the USB ports. The Pi is quite power-efficient, too—it can run off a powered USB port of your computer or your iPhone's USB wall charger!

Setting Up Your New Raspberry Pi

Because the Pi is just a processor with various communication ports and pins on it, if you are planning to buy one make sure you order it with all the required accessories. These mainly include connector cables:

- Ethernet cable for connection to the Internet
- USB-A to USB-B converter cable for power supply from a powered USB port
- HDMI cable for connection to a monitor
- Camera module (optional)
- USB expander hub if you plan to connect more than two USB devices to the Pi
- SD memory card with a capacity of at least 4 GB for installing the OS and other programs

Although several Linux-based operating systems can be installed on the Pi, Raspbian (latest version "wheezy" as of writing), which is a flavor of Debian optimized for the Pi, is recommended for beginners. The rest of this chapter will assume that you installed Raspbian on your Pi, because it works nicely out-of-the-box and has a large amount community support. Once you install the OS and connect a monitor, keyboard, and mouse to the Pi, it becomes a fully functional computer! A typical GUI-based setup is shown in Figure 11-2.

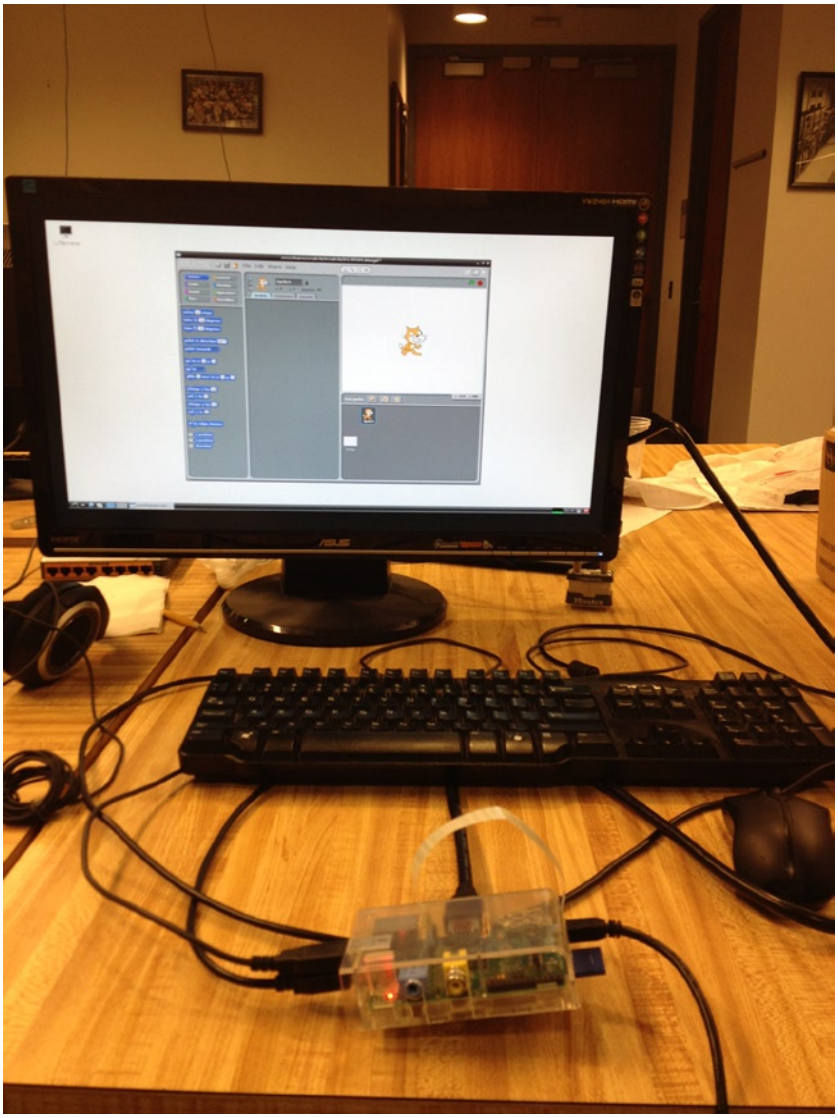


Figure 11-2. *Raspberry Pi connected to keyboard, mouse, and monitor—a complete computer!*

Installing Raspbian on the Pi

You will need a blank SD card with a capacity of at least 4 GB and access to a computer for this straightforward process. If you are a first-time user, it is recommended that you use the specially packaged New Out Of Box Software (NOOBS) at `www.raspberrypi.org/downloads`. Detailed instructions for using it can be found at the quick-start guide at `www.raspberrypi.org/wp-content/uploads/2012/04/quick-start-guide-v2_1.pdf`. In essence, you:

- Download the NOOBS software to your computer
- Unzip it on the SD card

- Insert the SD card into the Pi and boot it up with a keyboard and monitor attached to USB and HDMI, respectively
- Follow the simple on-screen instructions, making sure that you choose Raspbian as the operating system that you want to install

Note that this creates a user account "pi" with the password "raspberry." This account also has sudo access with the same password.

Initial Settings

After you install Raspbian, reboot and hold "Shift" to enter the `raspi-config` settings screen, which is shown in Figure 11-3.

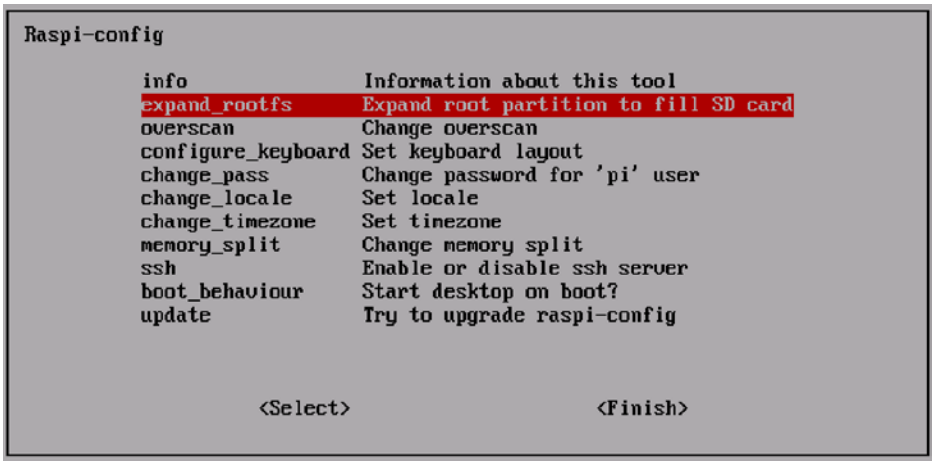


Figure 11-3. *The `raspi-config` screen*

- First, you should use the `'expand_rootfs'` option to enable the Pi to use your entire memory card.
- You can also change the RAM memory sharing settings using the `'memory_split'` option. My recommendation is to allocate as much as possible to the CPU, because OpenCV does not support the VideoCore GPU yet, so we will end up using the ARM CPU for all our computations.
- You can also use the `'boot_behavior'` option in `raspi-config` specify whether the Pi boots up into the GUI or a command line. Obviously, maintaining the GUI takes up processing power, so I recommend that you get comfortable with the Linux terminal and switch off the GUI. The typical method of access to a Pi that has been set to boot into the command line is by SSH-ing into it with X-forwarding. This essentially means that you connect to the Pi using Ethernet and have access to the Pi's terminal from a terminal on your computer. The X-forwarding means that the Pi will your computer's screen to render any windows (for example, OpenCV `imshow()` windows). Adafruit has a great guide for first-time SSH users at `http://learn.adafruit.com/downloads/pdf/adafruits-raspberry-pi-lesson-6-using-ssh.pdf`.

Installing OpenCV

Because Raspbian is a Linux flavor with Debian as its package-management system, the process to install OpenCV remains exactly the same as that for installing on 64-bit systems, as outlined in Chapter 2. Once you install OpenCV you should be able to run the demo programs just as you were able to on your computer. The demos that require live feed from a camera will also work if you attach a USB web camera to the Pi.

Figure 11-4 shows the OpenCV video homography estimation demo (`cpp-example-video_homography`) running from a USB camera on the Pi. As you might recall, this function tracks corners in frames and then finds a transformation (shown by the green lines) with reference to a frame that the user can choose. If you will run this demo yourself, you will observe that the 700 MHz processor in the Pi is quite inadequate to process 640x480 frames in such demos—the frame rate is very low.

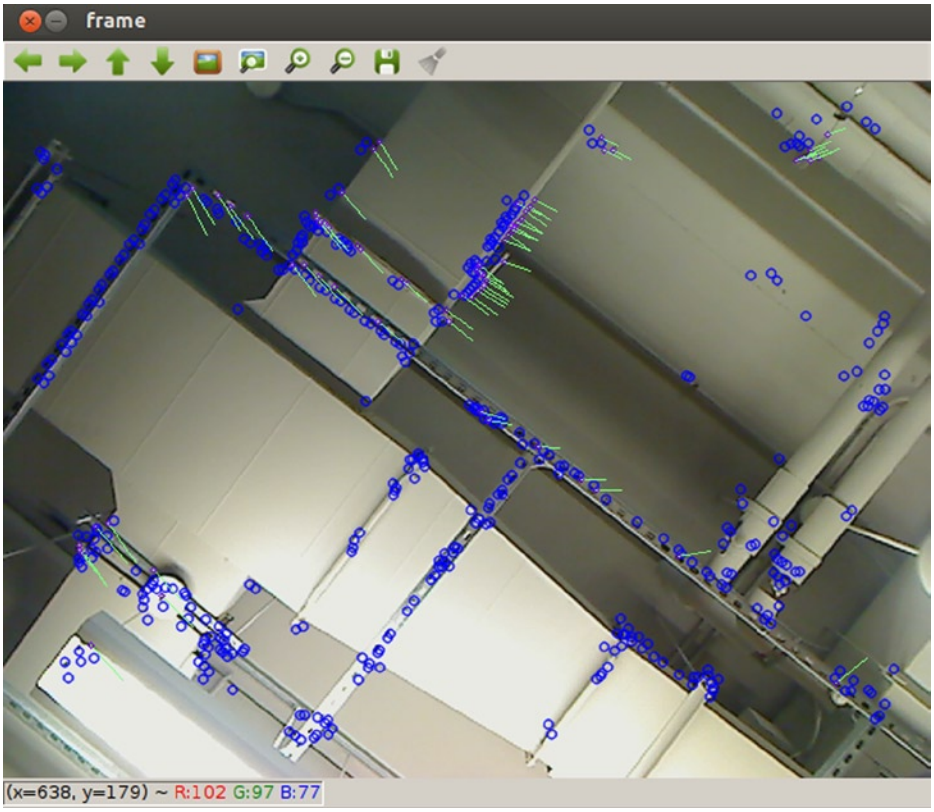


Figure 11-4. *OpenCV built-in video homography demo being run on the Raspberry Pi*

Figure 11-5 shows the OpenCV convex hull demo (`cpp-example-convexhull`) that chooses a random set of points and computes the smallest convex hull around them being executed on the Pi.

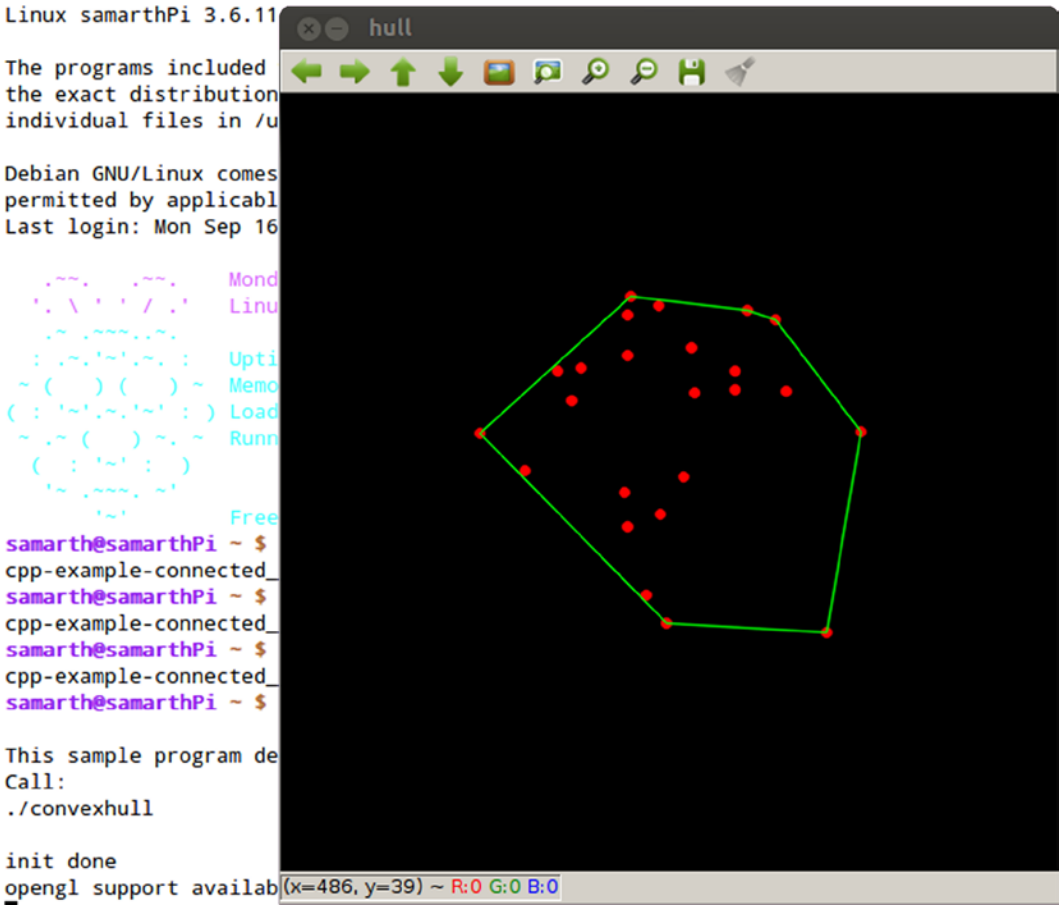


Figure 11-5. *OpenCV convex hull demo being run on the Pi*

Camera board

The makers of the Raspberry Pi also released a custom camera board recently to encourage image processing and computer vision applications of the Pi. The board is small and weighs just 3 grams, but it boasts a 5 Mega pixel CMOS sensor. It connects to the Pi using a ribbon cable and looks like Figure 11-6.

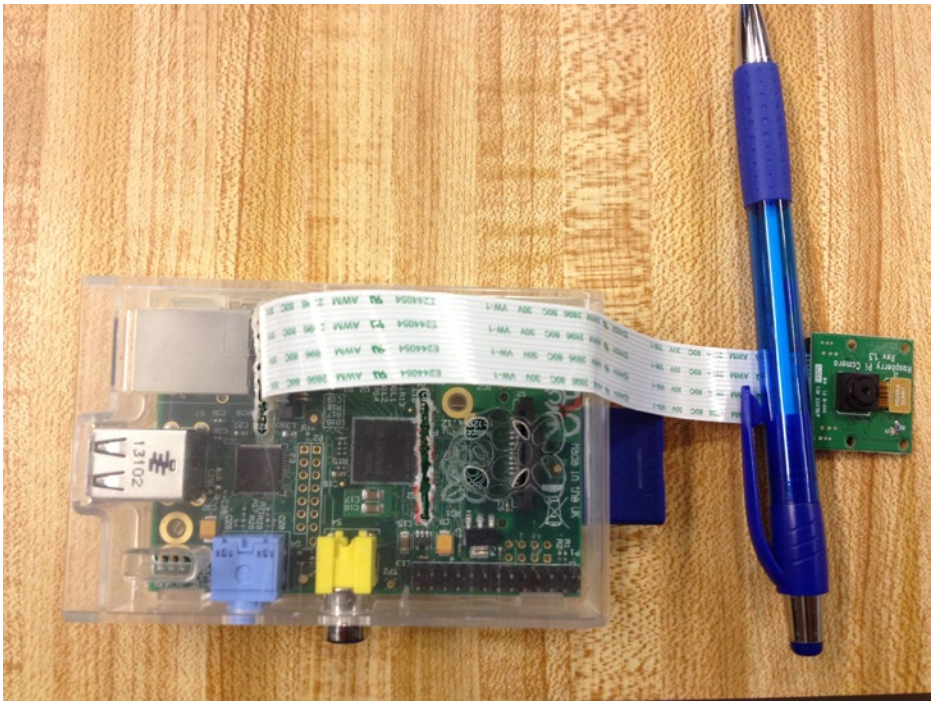


Figure 11-6. *The Raspberry Pi with the camera board attached to it*

Once you connect it by following the instructional video at `www.raspberrypi.org/camera`, you need to enable it from the `raspi-config` screen (which can be brought up by pressing and holding "Shift" at boot-up or typing `sudo raspi-config` at a terminal). The preinstalled utilities `raspistill` and `raspivid` can be used to capture still images and videos, respectively. They offer a lot of capture options that you can tweak. You can learn more about them by going through the documentation at `https://github.com/raspberrypi/userland/blob/master/host_applications/linux/apps/raspicam/RaspiCamDocs.odt`. The source code for these applications is open source, so we will see how to use it to get the board to interact with OpenCV.

Camera Board vs. USB Camera

You might ask, why use a camera board when you can use a simple USB camera like the ones you use on desktop computers? Two reasons:

1. The Raspberry Pi camera board connects to the Pi using a special connector that is designed to transfer data in parallel. This makes is faster than a USB camera
2. The camera board has a better sensor than a lot of other USB cameras that cost the same (or even more)

The only concern is that because the board is not a USB device, OpenCV does not recognize it out-of-the-box. I made a small C++ wrapper program that grabs the bytes from the camera buffer and puts them into an OpenCV `Mat` by reusing some C code that I found on Raspberry Pi forums. This code makes use of the open-sourced code released by developers of the camera drivers. This program allows you to specify the size of the frame that you want to capture and a boolean flag that indicates whether the captured frame should be colored or grayscale. The camera returns

image information in the form of the Y channel and subsampled U and V channels of the (YUV color space). To get a grayscale image, the wrapper program just has to set the Y channel as the data source of an OpenCV `Mat`. Things get complicated (and slow) if you want color. To get a color image, the following steps have to be executed by the wrapper program:

- Copy the Y, U, and V channels into OpenCV Mats
- Resize the U and V channels because the U and V returned by the camera are subsampled
- Perform a YUV to RGB conversion

These operations are time-consuming, and hence frame rates drop if you want to stream color images from the camera board.

By contrast, a USB camera captures in color by default, and you will have to spend extra time converting it to grayscale. Hence, unless you know how to set the capture mode of the camera to grayscale, grayscale images from USB cameras are costlier than color images.

As we have seen so far, most computer vision applications operate on intensity and, hence, just require a grayscale image. This is one more reason the use the camera board than a USB camera. Let us see how to use the wrapper code to grab frames from the Pi camera board. Note that this process requires OpenCV to be installed on the Pi.

- If you enabled your camera from `raspi-config`, you should already have the source code and libraries for interfacing with the camera board in `/opt/vc/`. Confirm this by seeing if the commands `raspistill` and `raspivid` work as expected.
- Clone the official MMAL Github repository to get the required header files by navigating to any directory in your home folder in a terminal and running the following command. Note that this directory will be referenced as USERLAND_DIR in the CMakeLists.txt file in Listing 11-1, so make sure that you replace USERLAND_DIR in that file with the full path to this directory. If you do not have Git installed on your Raspberry Pi you can install it by typing `sudo apt-get install git` in a terminal.)

  ```
  git clone https://github.com/raspberrypi/userland.git
  ```

- The wrapper program will work with the CMake build environment. In a separate folder (called DIR further on) make folders called “src” and “include.” Put Picam.cpp and cap.h files from code 10-1 into the “src” and “include” folders, respectively. These constitute the wrapper code
- To use the wrapper code, make a simple file like main.cpp shown in code 10-1 to grab frames from the camera board, show them in a window and measure the frame-rate. Put the file in the “src” folder
- The idea is to make an executable that uses functions and classes from both the main.cpp and PiCapture.cpp files. This can be done by using a CMakeLists.txt file like the one shown in code 10-1 and saving it in DIR
- To compile and build the executable, run in DIR

  ```
  mkdir build
  cd build
  cmake ..
  make
  ```

A little explanation about the wrapper code follows. As you might have realized, the wrapper code makes a class called `PiCapture` with a constructor that takes in the width, height, and boolean flag (true means color images are grabbed). The class also defines a method called `grab()` that returns an OpenCV `Mat` containing the grabbed image of the appropriate size and type.

Listing 11-1. Simple CMake project illustrating the wrapper code to capture frames from the Raspberry Pi camera board

main.cpp:

```
//Code to check the OpenCV installation on Raspberry Pi and measure frame rate
//Author: Samarth Manoj Brahmbhatt, University of Pennsyalvania

#include "cap.h"
#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>

using namespace cv;
using namespace std;

int main() {
    namedWindow("Hello");

    PiCapture cap(320, 240, false);

    Mat im;
    double time = 0;
    unsigned int frames = 0;
    while(char(waitKey(1)) != 'q') {
        double t0 = getTickCount();
        im = cap.grab();
        frames++;
        if(!im.empty()) imshow("Hello", im);
        else cout << "Frame dropped" << endl;

        time += (getTickCount() - t0) / getTickFrequency();
        cout << frames / time << " fps" << endl;
    }

    return 0;
}
```

cap.h:

```
#include <opencv2/opencv.hpp>
#include "interface/mmal/mmal.h"
#include "interface/mmal/util/mmal_default_components.h"
#include "interface/mmal/util/mmal_connection.h"
#include "interface/mmal/util/mmal_util.h"
#include "interface/mmal/util/mmal_util_params.h"

class PiCapture {
    private:
        MMAL_COMPONENT_T *camera;
        MMAL_COMPONENT_T *preview;
        MMAL_ES_FORMAT_T *format;
        MMAL_STATUS_T status;
        MMAL_PORT_T *camera_preview_port, *camera_video_port, *camera_still_port;
        MMAL_PORT_T *preview_input_port;
```

```
        MMAL_CONNECTION_T *camera_preview_connection;
        bool color;
    public:
        static cv::Mat image;
        static int width, height;
        static MMAL_POOL_T *camera_video_port_pool;
        static void set_image(cv::Mat _image) {image = _image;}
        PiCapture(int, int, bool);
        cv::Mat grab() {return image;}
};

static void color_callback(MMAL_PORT_T *, MMAL_BUFFER_HEADER_T *);
static void gray_callback(MMAL_PORT_T *, MMAL_BUFFER_HEADER_T *);
```

PiCapture.cpp:

```
/*
 * File:   opencv_demo.c
 * Author: Tasanakorn
 *
 * Created on May 22, 2013, 1:52 PM
 */

// OpenCV 2.x C++ wrapper written by Samarth Manoj Brahmbhatt, University of Pennsylvania

#include <stdio.h>
#include <stdlib.h>

#include <opencv2/opencv.hpp>

#include "bcm_host.h"

#include "interface/mmal/mmal.h"
#include "interface/mmal/util/mmal_default_components.h"
#include "interface/mmal/util/mmal_connection.h"
#include "interface/mmal/util/mmal_util.h"
#include "interface/mmal/util/mmal_util_params.h"

#include "cap.h"

#define MMAL_CAMERA_PREVIEW_PORT 0
#define MMAL_CAMERA_VIDEO_PORT 1
#define MMAL_CAMERA_CAPTURE_PORT 2

using namespace cv;
using namespace std;

int PiCapture::width = 0;
int PiCapture::height = 0;
MMAL_POOL_T * PiCapture::camera_video_port_pool = NULL;
Mat PiCapture::image = Mat();
```

```
static void color_callback(MMAL_PORT_T *port, MMAL_BUFFER_HEADER_T *buffer) {
    MMAL_BUFFER_HEADER_T *new_buffer;

    mmal_buffer_header_mem_lock(buffer);
    unsigned char* pointer = (unsigned char *)(buffer -> data);
    int w = PiCapture::width, h = PiCapture::height;
    Mat y(h, w, CV_8UC1, pointer);
    pointer = pointer + (h*w);
    Mat u(h/2, w/2, CV_8UC1, pointer);
    pointer = pointer + (h*w/4);
    Mat v(h/2, w/2, CV_8UC1, pointer);
    mmal_buffer_header_mem_unlock(buffer);
    mmal_buffer_header_release(buffer);

    if (port->is_enabled) {
        MMAL_STATUS_T status;

        new_buffer = mmal_queue_get(PiCapture::camera_video_port_pool->queue);

        if (new_buffer)
            status = mmal_port_send_buffer(port, new_buffer);

        if (!new_buffer || status != MMAL_SUCCESS)
            printf("Unable to return a buffer to the video port\n");
    }

    Mat image(h, w, CV_8UC3);

    resize(u, u, Size(), 2, 2, INTER_LINEAR);
    resize(v, v, Size(), 2, 2, INTER_LINEAR);
    int from_to[] = {0, 0};
    mixChannels(&y, 1, &image, 1, from_to, 1);
    from_to[1] = 1;
    mixChannels(&v, 1, &image, 1, from_to, 1);
    from_to[1] = 2;
    mixChannels(&u, 1, &image, 1, from_to, 1);
    cvtColor(image, image, CV_YCrCb2BGR);

    PiCapture::set_image(image);
}

static void gray_callback(MMAL_PORT_T *port, MMAL_BUFFER_HEADER_T *buffer) {
    MMAL_BUFFER_HEADER_T *new_buffer;

    mmal_buffer_header_mem_lock(buffer);
    unsigned char* pointer = (unsigned char *)(buffer -> data);
    PiCapture::set_image(Mat(PiCapture::height, PiCapture::width, CV_8UC1, pointer));
    mmal_buffer_header_release(buffer);
```

```
    if (port->is_enabled) {
        MMAL_STATUS_T status;

        new_buffer = mmal_queue_get(PiCapture::camera_video_port_pool->queue);

        if (new_buffer)
            status = mmal_port_send_buffer(port, new_buffer);

        if (!new_buffer || status != MMAL_SUCCESS)
            printf("Unable to return a buffer to the video port\n");
    }
}

PiCapture::PiCapture(int _w, int _h, bool _color) {
    color = _color;
    width = _w;
    height = _h;

    camera = 0;
    preview = 0;
    camera_preview_port = NULL;
    camera_video_port = NULL;
    camera_still_port = NULL;
    preview_input_port = NULL;
    camera_preview_connection = 0;

    bcm_host_init();

    status = mmal_component_create(MMAL_COMPONENT_DEFAULT_CAMERA, &camera);
    if (status != MMAL_SUCCESS) {
        printf("Error: create camera %x\n", status);
    }

    camera_preview_port = camera->output[MMAL_CAMERA_PREVIEW_PORT];
    camera_video_port = camera->output[MMAL_CAMERA_VIDEO_PORT];
    camera_still_port = camera->output[MMAL_CAMERA_CAPTURE_PORT];

    {
        MMAL_PARAMETER_CAMERA_CONFIG_T cam_config = {
            { MMAL_PARAMETER_CAMERA_CONFIG, sizeof (cam_config)}, width, height, 0, 0,
width, height, 3, 0, 1, MMAL_PARAM_TIMESTAMP_MODE_RESET_STC };
        mmal_port_parameter_set(camera->control, &cam_config.hdr);
    }

    format = camera_video_port->format;

    format->encoding = MMAL_ENCODING_I420;
    format->encoding_variant = MMAL_ENCODING_I420;

    format->es->video.width = width;
    format->es->video.height = height;
    format->es->video.crop.x = 0;
```

```
    format->es->video.crop.y = 0;
    format->es->video.crop.width = width;
    format->es->video.crop.height = height;
    format->es->video.frame_rate.num = 30;
    format->es->video.frame_rate.den = 1;

    camera_video_port->buffer_size = width * height * 3 / 2;
    camera_video_port->buffer_num = 1;

    status = mmal_port_format_commit(camera_video_port);

    if (status != MMAL_SUCCESS) {
        printf("Error: unable to commit camera video port format (%u)\n", status);
    }

    // create pool form camera video port
    camera_video_port_pool = (MMAL_POOL_T *) mmal_port_pool_create(camera_video_port,
camera_video_port->buffer_num, camera_video_port->buffer_size);

    if(color) {
        status = mmal_port_enable(camera_video_port, color_callback);
        if (status != MMAL_SUCCESS)
            printf("Error: unable to enable camera video port (%u)\n", status);
        else
            cout << "Attached color callback" << endl;
    }
    else {
        status = mmal_port_enable(camera_video_port, gray_callback);
        if (status != MMAL_SUCCESS)
            printf("Error: unable to enable camera video port (%u)\n", status);
        else
            cout << "Attached gray callback" << endl;
    }

    status = mmal_component_enable(camera);

    // Send all the buffers to the camera video port
    int num = mmal_queue_length(camera_video_port_pool->queue);
    int q;

    for (q = 0; q < num; q++) {
        MMAL_BUFFER_HEADER_T *buffer = mmal_queue_get(camera_video_port_pool->queue);

        if (!buffer) {
            printf("Unable to get a required buffer %d from pool queue\n", q);
        }

        if (mmal_port_send_buffer(camera_video_port, buffer) != MMAL_SUCCESS) {
            printf("Unable to send a buffer to encoder output port (%d)\n", q);
        }
    }
```

```
    if (mmal_port_parameter_set_boolean(camera_video_port, MMAL_PARAMETER_CAPTURE, 1) !=
MMAL_SUCCESS) {
        printf("%s: Failed to start capture\n", __func__);
    }

    cout << "Capture started" << endl;
}
```

CMakeLists.txt:

```
cmake_minimum_required(VERSION 2.8)

project(PiCapture)

SET(COMPILE_DEFINITIONS -Werror)

find_package( OpenCV REQUIRED )

include_directories(/opt/vc/include)
include_directories(/opt/vc/include/interface/vcos/pthreads)
include_directories(/opt/vc/include/interface/vmcs_host)
include_directories(/opt/vc/include/interface/vmcs_host/linux)
include_directories(USERLAND_DIR)
include_directories("${PROJECT_SOURCE_DIR}/include")

link_directories(/opt/vc/lib)
link_directories(/opt/vc/src/hello_pi/libs/vgfont)

add_executable(main src/main.cpp src/PiCapture.cpp)

target_link_libraries(main mmal_core mmal_util mmal_vc_client bcm_host ${OpenCV_LIBS})
```

From now on you can use the same strategy to grab frames from the camera board—include `cam.h` in your source file, and make an executable that uses both your source file and `PiCapture.cpp`.

Frame-Rate Comparisons

Figure 11-7 shows frame-rate comparisons for a USB camera versus the camera board. The programs used are simple; they just grab frames and show them using `imshow()` in a loop. In case of the camera board, the boolean flag in the argument list to the `PiCapture` constructor is used to switch from color to grayscale. For the USB camera, grayscale images are obtained by using `cvtColor()` on the color images.

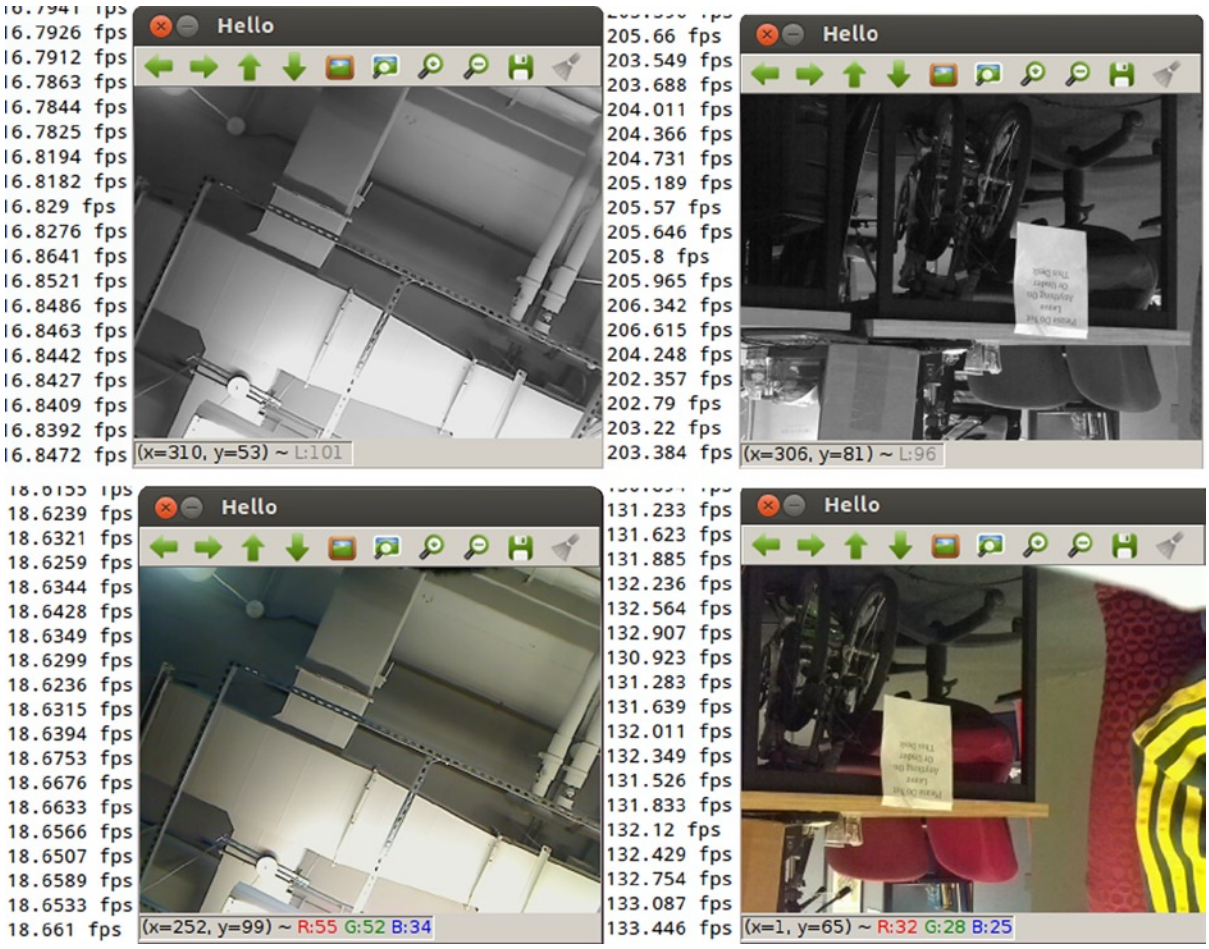


Figure 11-7. *Frame-rate tests. Clockwise from top left: Grayscale from USB camera, grayscale from camera board, color from camera board, and color from USB camera*

The frame-rate calculations are done as before using the OpenCV functions `getTickCount()` and `getTickFrequency()`. However, sometimes the parallel processes going on in `PiCapture.cpp` seem to be messing up the tick counts, and the frame rates measured for the camera board are reported to be quite higher than they actually are. When you run the programs yourself, you will find that both approaches give almost the same frame rate (visually) for color images, but grabbing from the camera board is much faster than using a USB camera for grayscale images.

Usage Examples

Once you set everything up everything on your Pi, it acts pretty much like your normal computer and you should be able to run all the OpenCV code that you ran on your normal computer, except that it will run slower. As usage examples, we will check frame rates of the two types of object detectors we have developed in this book, color-based and ORB keypoint-based.

Color-based Object Detector

Remember the color-based object detector that we perfected in Chapter 5 (Listing 5-7)? Let us see how it runs for USB camera versus color images grabbed using the wrapper code. Figure 11-8 shows that for frames of size 320 x 240, the detector runs about 50 percent faster if we grab frames using the wrapper code!

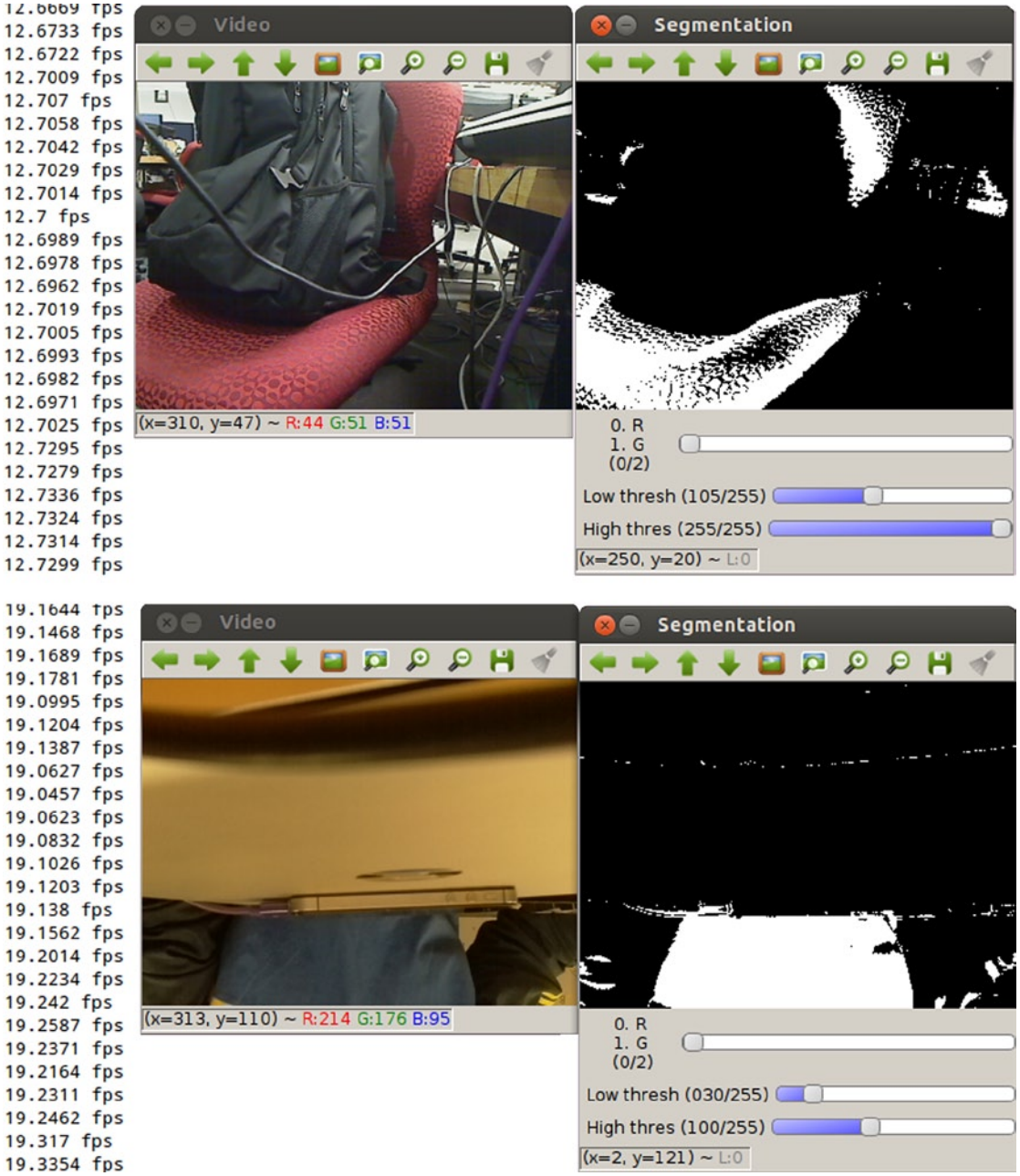


Figure 11-8. *Color-based object detector using color frames from USB camera (top) and camera board (bottom)*

ORB Keypoint-based Object Detector

The ORB keypoint-based object detector that we developed in Chapter 8 (Listing 8-4) requires a lot more computation than computing histogram back-projections, and so it is a good idea to see how the two methods perform when the bottleneck is not frame-grabbing speed but frame-processing speed. Figure 11-9 shows that grabbing 320 x 240 grayscale frames (remember, ORB keypoint detection and description requires just a grayscale image) using the wrapper code runs about 14 percent faster than grabbing color frames from the USB camera and manually converting them to grayscale images.

Figure 11-9. *ORB-based object detector using frames grabbed from a USB camera (top) and camera board (bottom)*

Summary

With this chapter, I conclude our introduction to the fascinating field of embedded computer vision. This chapter gave you a detailed insight into running your own vision algorithms on the ubiquitous Raspberry Pi, including the use of its swanky new camera board. Feel free to let your mind wander and come up with some awesome use-cases! You will find that knowing how to make a Raspberry Pi understand what it sees is a very powerful tool, and its applications know no bounds.

There are lots of embedded systems other than the Pi out there in the market that can run OpenCV. My advice is to pick one that best suits your application and budget, and get really good at using it effectively rather than knowing something about every embedded vision product out there. At the end of the day, these platforms are just a means to an end; it is the algorithm and idea that makes the difference.

Index

A

Affine transforms, 155
  application to image transformation, 156
  estimation, 158
  flipping X, Y coordinates, 155
  getAffineTransform() function, 158
  getRotationMatrix2D() function, 156–157
  matrix multiplication, 156
  recovery, 158–160
  rotation transformation, 155
  scaling transformation, 155
  warpAffine() function, 156

B

Bag of visual words descriptor, 143
Bags of keypoints, 142
Blurring images, 45
  GaussianBlur() function, 46
  Gaussian kernel, 46–47
  pyrDown() function, 48
  resize() function, 48
  setTrackbarPos() function, 48
Box filters, 132

C

Calibration, stereo camera, 180
Camera board
  features, 206
  Frame-rate tests, 215
  imshow() function, 214
  PiCapture constructor, 214
  raspistill, raspivid, 207
  vs. USB Camera, 207
Camera coordinate system, 174
Cameras
  single camera calibration, 173
    camera coordinate system, 174
    camera matrix, 175
    mathematical model, 173–174
    OpenCV implementation, 176
    planar checkerboard, 175–176
    projection center, 174
  stereo camera, 179
  stereo camera calibration, 180
  stereo camera triangulation model, 179
Camshift tracking algorithm, 13
Classification problem, 140
Color-based object detector, 119
  using color frames, 216
  wrapper code, 216
Computer vision, 3
  in OpenCV
    alpha version, 4
    definition, 4
    modules, 5
    objectives, 4
Contours
  drawContours() function, 69
  findContours() function, 67
  hierarchy levels, 69
  OpenCV's contour extraction, 67–68
  pointPolygonTest(), 70

D

Demos
  camshift tracking algorithm, 13
  hough transform, 18
    circle detection, 18
    line detection, 19
  Image painting, 21

Demos (*cont.*)
meanshift algorithm, image segmentation, 19–20
minarea demo, 21
stereo_matching demo, 16
video_homography demo, 16
Disparity, 179, 194
stereo, 193–194

E

Embedded computer vision
Raspberry Pi "/t" (*see* Raspberry Pi, 201)

F

Feature matching, 120
Features, 121
floodFill() function, 100

G

Gaussian kernel, 46–47
Geometric image transformations, 155
3D geometry, 173
single camera calibration, 173
camera coordinate system, 174
camera matrix, 175
mathematical model, 173–174
OpenCV implementation, 176
planar checkerboard, 175–176
projection center, 174
getStructuralElement() function, 51
GrabCut segmentation, 110
GUI windows, 23
callback functions, 27
color-space converter, 27, 29
global variable declaration, 28
track-bar function, 28

H

Histograms, 111
backprojection, 113
equalizeHist() function, 111
MeanShift() and CamShift() functions, 116
Hough transform, 74
circle detection, 77
lines detection, 74

I, J

Image Filters, 41
blurring images, 45
GaussianBlur() function, 46
Gaussian kernel, 46–47
pyrDown() function, 48
resize() function, 48–49
setTrackbarPos() function, 48
corners, 57
circle() function, 58
goodFeaturesToTrack() function, 57
STL library, 57
Dilating image, 49
Edges, 51
Canny edges, 56
thresholded Scharr operator, image, 52
Erosion image, 49
filter2D() function, 43, 45
horizontal and vertical edges, 43
kernel matrix, 43
morphologyEX() function, 63
object detector program, 60
split() function, 45
temperature fluctuation, 42
Image panoramas. *See* Panoramas
Images, 23
cv\:\:Mat's at() attribute, 33
cv\:\:mat structure, 24
Access elements, 25
creation, 25
definition, 24
expressions, 25
cvtColor() function, 26
highgui module's, 23
imread() function, 24
RGB triplets, 26
ROIS (*see* Regions of Interest(ROIs))
videos (*see* Videos)
waitKey().function, 24
Image segmentation
definition, 95
floodFill() function, 100
foreground extraction and counting objects, 95
GrbCut, 110
histograms, 111
threshold() function, 96
color-based object detection, 96
watershed algorithm, 103
challenges, 105
drawContours(), 107
working principle, 104
imshow() function, 214
Integrated Development Environment(IDE), 12

K, L

Kernel function, 142
Keypoint-based object detection method, 120
feature matching, 120
keypoint descriptors, 120

keypoints, 120
ORB, 136
BRIEF descriptors, 137
oriented FAST keypoints, 137
SIFT, 121
descriptor matching, 125
higher scale, 121
keypoint descriptors, 125
keypoint detection and orientation estimation, 121
lower scale, 121
scale and rotation invariance, 121
SURF, 131
descriptor, 134
keypoint detection, 131
Keypoint descriptors, 120, 125
gradient orientation histograms, 125
illumination invariance, 125
rotation invariance, 125
Keypoint detection, 121
Difference of Gaussians Pyramid, 122–123
gradient orientation computation, 124
maxima and minima selection, 123–124
orientation histogram, 125
scale pyramid, 121–122
Keypoint orientation, 133
Keypoints, 120

M, N

Machine learning (ML), 140
classification, 140
features, 140
labels, 141
regression, 140
SVM, 141
classification algorithm, 141
classifier hyperplane, 141
XOR problem, 141–142
Mac OSX, 12

O

Object categorization, 142
organization, 143
BOW object categorization, 148–152
CmakeLists.txt, 147
Config.h.in, 147
data folder, 144
project root folder, 144
templates and training images, 146–147
strategy, 142
bag of visual words descriptor, 143
bags of keypoints/visual words, 142
multiclass classification, 143
Object detection. *See* Keypoint-based object detection method
OpenCV, computer, 7
demos (*see* Demos)
Mac OSX, 12
Ubuntu operating systems, 7
flags, 10
hello_opencv.cpp, 11
IDE, 12
installation, 7
without superuser privileges, 11–12
windows, 12
OpenCV, single camera calibration, 176
ORB Keypoint-based Object Detector, 217
Orientation estimation, 121
gradient orientation computation, 124
orientation histogram, 125
Oriented FAST and Rotated BRIEF (ORB), 136
BRIEF descriptors, 137
binary pixel intensity tests, 137
extraction and matching, FLANN-LSH, 138–139
Hamming distance, 137
machine learning algorithm, 137
ORB keypoint based object detecor, 140
pair-picking pattern, 138
oriented FAST keypoints, 137

P, Q

Panoramas, 166
creation of, 167
cylindrical panorama, 170
cylindrical warping, 170, 172
global estimation, 166
Golden Gate panorama, 169
image stitching, 170
OpenCV stitching module, 166–167
seam blending, 166
Perspective transforms, 161
clicking matching points, 163
findHomography() function, 161
matrix, 161
recovery and application, 161–163
recovery by matching ORB features, 164–166
Pixel map, 187
Projection center, 174

R

RANdom Sample Consensus (RANSAC), 80
Raspberry Pi
camera board
features, 206
Frame-rate tests, 215
imshow() function, 214

Raspberry Pi (*cont.*)
PiCapture constructor, 214
raspistill, raspivid, 207
vs. USB Camera, 207
color-based object detector
using color frames, 216
wrapper code, 216
features, 202
installation, 203
ORB keypoint-based object detector, 217
power-efficient, 202
Raspberry Pi board, 201
Setting Up
'boot_behavior' option, 204
built-in video homography demo, 205
convex hull demo, 206
Ethernet cable, 202
HDMI cable, 202
'memory_split' option, 204
raspi-config screen, 204
USB-A to USB-B converter cable, 202
USB expander hub, 202
Regions of Interest (ROIs)
cropping, rectangular image, 30
mouse_callback function, 32
point structure, 32
rect structure, 32
Regression problem, 140

S

Scale, 121
higher scale, 121
lower scale, 121
Scale Invariant Feature Transform (SIFT), 121
descriptor matching, 125
brute force approach, 126–128
DescriptorExtractor class, 126
DescriptorMatcher class, 126
drawMatches() function, 128
FeatureDetector class, 125
FLANN based matcher, 128–130
nearest neighbor searches, 125
OpenCV, 125
higher scale, 121
keypoint descriptors, 125
gradient orientation histograms, 125
illumination invariance, 125
rotation invariance, 125
keypoint detection and orientation estimation, 121
Difference of Gaussians Pyramid, 122–123
gradient orientation computation, 124
maxima and minima selection, 123–124
orientation histogram, 125
scale pyramid, 121–122
lower scale, 121
scale and rotation invariance, 121
Semi-Global Block Matching (SGBM)
algorithm, 193
setTrackbarPos() function, 48
Shapes, 67
bounding boxes and circles, 91
contours, 67
convexHull() function, 92
generalized Hough transform, 80
Hough transform, 74
RANSAC, 80
debug() function, 81
findEllipse, 82
object oriented strategy, 81
parameters, 81
Single camera calibration, 173
camera coordinate system, 174
camera matrix, 175
mathematical model, 173–174
OpenCV implementation, 176
planar checkerboard, 175–176
projection center, 174
Speeded Up Robust Features (SURF), 131
descriptor, 134
extraction and matching, FLANN, 134–135
FLANN matching, 136
frame rates, 134
oriented square patches, 134
keypoint detection, 131
box-filter approximations, 131
box filters, 132
Hessian matrix, 131
Hessian matrix determinant, 131
integral image, 132
keypoint orientation, 133
orientation assignment, 133
scale-space pyramid, 133
second-order Gaussian derivatives, 131
siding orientation windows, 133
split() function, 45
Stereo camera, 179
Stereo camera calibration, 180
Stereo_matching demo, 16
Stereo rectification, 187, 189–193
Stereo vision, 173, 179
disparity, 179, 186, 193–194
pixel map, 187
pixel matching, 187
Semi-Global Block Matching (SGBM)
algorithm, 193
stereo camera, 179
stereo camera calibration, 180
stereo rectification, 187, 189–193
triangulation, 179

Support vector machine (SVM), 141
classification algorithm, 141
classifier hyperplane, 141
kernel, 142
XOR problem, 141

T

Triangulation, stereo camera model, 179

U

Ubuntu operating systems, 7
flags, 10
hello_opencv.cpp, 11
installation, 7
without superuser privileges, 11–12
USB Camera/File, 34

V

Video_homography demo, 16
Videos
get() function, 36
USB Camera/File, 34
VideoWriter object, 36
Visual words, 142

W, X, Y, Z

waitKey().function, 24
Watershed segmentation, 103
Windows, 12
Wrapper code, 216

Practical OpenCV

Samarth Brahmbhatt

Apress®

Practical OpenCV

ISBN-13 (pbk): 978-1-4302-6079-0

ISBN-13 (electronic): 978-1-4302-6080-6

President and Publisher: Paul Manning
Lead Editor: Michelle Lowman
Developmental Editor: Tom Welsh
Technical Reviewer: Steven Hickson
Editorial Board: Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Louise Corrigan, James DeWolf, Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing, Matt Wade, Steve Weiss, Tom Welsh
Coordinating Editor: Jill Balzano
Copy Editor: Laura Lawrie
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover Designer: Anna Ishchenko
Cover Image Designer: Rakshit Kothari

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

To my family, who convinced me into this adventure.

And to hope, and the power of dreams.

Contents

About the Author xiii

About the Technical Reviewer xv

Acknowledgments xvii

Part 1: Getting Comfortable 1

Chapter 1: Introduction to Computer Vision and OpenCV 3

Why Was This Book Written? 3

OpenCV 4

History of OpenCV 4

Built-in Modules 4

Summary 5

Chapter 2: Setting up OpenCV on Your Computer 7

Operating Systems 7

Ubuntu 7

Windows 12

Mac OSX 12

Summary 12

Chapter 3: CV Bling—OpenCV Inbuilt Demos 13

Camshift 13

Stereo Matching 16

Homography Estimation in Video 16

Circle and Line Detection 18

Image Segmentation 19

Bounding Box and Circle 21
Image Inpainting 21
Summary 22

Chapter 4: Basic Operations on Images and GUI Windows 23

Displaying Images from Disk in a Window 23
The cv::Mat Structure 24
Creating a cv::Mat 25
Accessing elements of a cv::Mat 25
Expressions with cv::Mat 25
Converting Between Color-spaces 26
GUI Track-Bars and Callback Functions 27
Callback Functions 27
ROIs: Cropping a Rectangular Portion out of an Image 30
Region of Interest in an Image 30
Accessing Individual Pixels of an Image 33
Exercise 33
Videos 34
Displaying the Feed from Your Webcam or USB Camera/File 34
Writing Videos to Disk 36
Summary 37

Part 2: Advanced Computer Vision Problems and Coding Them in OpenCV 39

Chapter 5: Image Filtering 41

Image Filters 41
Blurring Images 45
Resizing Images—Up and Down 48
Eroding and Dilating Images 49
Detecting Edges and Corners Efficiently in Images 51

Edges 51
Canny Edges 56
Corners 57
Object Detector App 60
Morphological Opening and Closing of Images to Remove Noise 63
Summary 65
Chapter 6: Shapes in Images 67
Contours 67
Point Polygon Test 70
Hough Transform 74
Detecting Lines with Hough Transform 74
Detecting Circles with Hough Transform 77
Generalized Hough Transform 80
RANdom Sample Consensus (RANSAC) 80
Bounding Boxes and Circles 91
Convex Hulls 92
Summary 93
Chapter 7: Image Segmentation and Histograms 95
Image Segmentation 95
Simple Segmentation by Thresholding 96
Floodfill 100
Watershed Segmentation 103
GrabCut Segmentation 110
Histograms 111
Equalizing Histograms 111
Histogram Backprojections 113
Meanshift and Camshift 116
Summary 117

Chapter 8: Basic Machine Learning and Object Detection Based on Keypoints 119

Keypoints and Keypoint Descriptors: Introduction and Terminology 119

General Terms 120

How Does the Keypoint-Based Method Work? 120

SIFT Keypoints and Descriptors 121

Keypoint Detection and Orientation Estimation 121

SIFT Keypoint Descriptors 125

Matching SIFT Descriptors 125

SURF Keypoints and Descriptors 131

SURF Keypoint Detection 131

SURF Descriptor 134

ORB (Oriented FAST and Rotated BRIEF) 136

Oriented FAST Keypoints 137

BRIEF Descriptors 137

Basic Machine Learning 140

SVMs 141

Object Categorization 142

Strategy 142

Organization 143

Summary 153

Chapter 9: Affine and Perspective Transformations and Their Applications to Image Panoramas 155

Affine Transforms 155

Applying Affine Transforms 156

Estimating Affine Transforms 158

Perspective Transforms 161

Panoramas 166

Summary 172

Chapter 10: 3D Geometry and Stereo Vision 173

Single Camera Calibration 173

OpenCV Implementation of Single Camera Calibration 176

Stereo Vision 179

Triangulation 179

Calibration 180

Rectification and Disparity by Matching 186

Summary 200

Chapter 11: Embedded Computer Vision: Running OpenCV Programs on the Raspberry Pi 201

Raspberry Pi 202

Setting Up Your New Raspberry Pi 202

Installing Raspbian on the Pi 203

Initial Settings 204

Installing OpenCV 205

Camera board 206

Camera Board vs. USB Camera 207

Frame-Rate Comparisons 214

Usage Examples 215

Color-based Object Detector 216

ORB Keypoint-based Object Detector 217

Summary 218

Index 219

About the Author

Originally from the quiet city of Gandhinagar in India, **Samarth Brahmbhatt** is at present a graduate student at the University of Pennsylvania in Philadelphia, USA. He loves making and programming all kinds of robots, although he has a soft spot for ones that can see well. Samarth hopes to do doctoral research on developing vision algorithms for robots by drawing inspiration from how humans perceive their surroundings. When he is not learning new things about computer vision, Samarth likes to read Tolkien and Le Carré, travel to new places, and cook delicious Indian food.

About the Technical Reviewer

Steven Hickson is an avid technical blogger and current graduate student at Georgia Institute of Technology. He graduated magna cum laude with a degree in Computer Engineering from Clemson University before moving on to the Department of Defense. After consulting and working at the DoD, Steven decided to pursue his PhD with a focus in computer vision, robotics, and embedded systems. His open-source libraries are used the world over and they have been featured in places such as Linux User and Developer Magazine, raspberrypi.org, Hackaday, and Lifehacker. In his free time, Steven likes to rock climb, program random bits of code, and play Magic: The Gathering.

Acknowledgments

The author would like to acknowledge the excellent job done by Tom Welsh, Jill Balzano, and Michelle Lowman at Apress in editing and managing the workflow of this book. He would also like to acknowledge the beautiful design work by Rakshit Kothari for the cover and the excellent technical reviewing by Steven Hickson.