OpenCV 4 Computer Vision Application Programming Cookbook

Fourth Edition

Build complex computer vision applications with OpenCV and C++

Packt

www.packt.com

David Millán Escrivá and Robert Laganiere

OpenCV 4 Computer Vision Application Programming Cookbook

Fourth Edition

Build complex computer vision applications with OpenCV and C++

David Millán Escrivá
Robert Laganiere

Packt

BIRMINGHAM - MUMBAI

OpenCV 4 Computer Vision Application Programming Cookbook
Fourth Edition

Copyright © 2019 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Pavan Ramchandani
Acquisition Editor: Sandeep Mishra
Content Development Editor: Zeeyan Pinheiro
Technical Editor: Gaurav Gala
Copy Editor: Safis Editing
Project Coordinator: Vaidehi Sawant
Proofreader: Safis Editing
Indexer: Priyanka Dhadke
Graphics: Alishon Mendonsa
Production Coordinator: Nilesh Mohite

First published: May 2011
Second edition: August 2014
Third edition: February 2017
Fourth edition: May 2019

Production reference: 1020519

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78934-072-3

`www.packtpub.com`

mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

Packt.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the authors

David Millán Escrivá was 8 years old when he wrote his first program on an 8086 PC in Basic, which enabled the 2D plotting of basic equations. In 2005, he finished his studies in IT with honors, through the Universitat Politécnica de Valencia, in human-computer interaction supported by computer vision with OpenCV (v0.96). He has worked with Blender, an open source, 3D software project, and on its first commercial movie, *Plumiferos*, as a computer graphics software developer. David has more than 10 years' experience in IT, with experience in computer vision, computer graphics, pattern recognition, and machine learning, working on different projects, and at different start-ups, and companies. He currently works as a researcher in computer vision.

Robert Laganiere is a professor at the University of Ottawa, Canada. He is also a faculty member of the VIVA research lab and is the coauthor of several scientific publications and patents in content-based video analysis, visual surveillance, driver-assistance, object detection, and tracking. He cofounded Visual Cortek, a video analytics start-up, which was later acquired by iWatchLife. He is also a consultant in computer vision and has assumed the role of chief scientist in a number of start-ups companies, including Cognivue Corp, iWatchLife, and Tempo Analytics. Robert has a Bachelor of Electrical Engineering degree from Ecole Polytechnique in Montreal (1987), and M.Sc. and Ph.D. degrees from INRS-Telecommunications, Montreal (1996).

About the reviewers

Nibedit Dey is a software engineer turned entrepreneur with over 8 years' experience of building complex software-based products. Before starting his entrepreneurial journey, he worked for L&T and Tektronix in different R&D roles. He has reviewed books including *The Modern C++ Challenge*, *Hands-On GUI Programming with C++ and Qt5*, *Getting Started with Qt5* and *Hands-On High Performance Programming with Qt 5* for Packt Publishing.

I would like to thank the online programming communities, bloggers, and my peers from earlier organizations, from whom I have learned a lot over the years.

Christian Stehno studied computer science and got his diploma from Oldenburg University in 2000. Since then, he has worked on different topics in computer science, first as a researcher in theoretical computer science at university. Later on, he switched to embedded system design at a research institute. In 2010, he started his own company, CoSynth, which develops embedded systems and intelligent cameras for industrial automation. In addition to this, he has been a long-time member of the Irrlicht 3D engine developer team.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface 1

Chapter 1: Playing with Images 8
Installing the OpenCV library 8
Getting ready 9
How to do it... 9
How it works... 12
There's more... 13
Using Qt for OpenCV developments 13
The OpenCV developer site 15
See also 15
Loading, displaying, and saving images 15
Getting ready 15
How to do it... 16
How it works... 18
There's more... 20
Clicking on images 20
Drawing on images 21
Running the example with Qt 22
See also 23
Exploring the cv::Mat data structure 23
How to do it... 23
How it works... 26
There's more... 29
The input and output arrays 30
See also 30
Defining regions of interest 30
Getting ready 31
How to do it... 31
How it works... 32
There's more... 33
Using image masks 33
See also 34

Chapter 2: Manipulating the Pixels 35
Accessing pixel values 36
Getting ready 37
How to do it... 37
How it works... 39
There's more... 40
The cv::Mat_ template class 41

See also 41
Scanning an image with pointers 41
Getting ready 42
How to do it... 42
How it works... 44
There's more... 45
Other color reduction formulas 45
Having input and output arguments 46
Efficient scanning of continuous images 48
Low-level pointer arithmetics 49
See also 50
Scanning an image with iterators 50
Getting ready 50
How to do it... 51
How it works... 52
There's more... 53
See also 53
Writing efficient image-scanning loops 53
How to do it... 54
How it works... 54
There's more... 56
See also 56
Scanning an image with neighbor access 57
Getting ready 57
How to do it... 57
How it works... 59
There's more... 60
See also 61
Performing simple image arithmetic 62
Getting ready 62
How to do it... 62
How it works... 63
There's more... 64
Overloaded image operators 64
Splitting the image channels 65
Remapping an image 66
How to do it... 66
How it works... 68
See also 69
Chapter 3: Processing Color Images with Classes 70
Comparing colors using the strategy design pattern 71
How to do it... 71
How it works... 73
There's more... 77
Computing the distance between two color vectors 77

Using OpenCV functions 78
The functor or function object 80
The OpenCV base class for algorithms 81
See also 81
Segmenting an image with the GrabCut algorithm 82
How to do it... 82
How it works... 85
See also 86
Converting color representations 86
Getting ready 87
How to do it... 87
How it works... 89
See also 90
Representing colors with hue, saturation, and brightness 90
How to do it... 91
How it works... 93
There's more... 95
Using colors for detection – skin tone detection 95
Chapter 4: Counting the Pixels with Histograms 99
Computing the image histogram 99
Getting started 100
How to do it... 101
How it works... 105
There's more... 106
Computing histograms of color images 106
See also 108
Applying lookup tables to modify the image's appearance 109
How to do it... 109
How it works... 110
There's more... 110
Stretching a histogram to improve the image contrast 111
Applying a lookup table on color images 113
Equalizing the image histogram 114
How to do it... 114
How it works... 116
Backprojecting a histogram to detect specific image content 116
How to do it... 117
How it works... 119
There's more... 119
Backprojecting color histograms 120
Using the mean shift algorithm to find an object 123
How to do it... 124
How it works... 127
See also 128
Retrieving similar images using histogram comparison 128

How to do it... 129
How it works... 131
See also 131
Counting pixels with integral images 132
How to do it... 132
How it works... 134
There's more... 135
Adaptive thresholding 135
Visual tracking using histograms 139
See also 143
Chapter 5: Transforming Images with Morphological Operations 144
Eroding and dilating images using morphological filters 145
Getting ready 145
How to do it... 146
How it works... 147
There's more... 149
See also 149
Opening and closing images using morphological filters 150
How to do it... 150
How it works... 152
See also 153
Detecting edges and corners using morphological filters 153
Getting ready 153
How to do it... 154
How it works... 156
See also 158
Segmenting images using watersheds 158
How to do it... 158
How it works... 163
There's more... 164
See also 166
Extracting distinctive regions using MSER 166
How to do it... 166
How it works... 169
See also 172
Extracting foreground objects with the GrabCut algorithm 173
How to do it... 173
How it works... 176
See also 176
Chapter 6: Filtering the Images 177
Filtering images using low-pass filters 178
How to do it... 178
How it works... 180
See also 183

Downsampling an image 183
How to do it... 183
How it works... 185
There's more... 187
Interpolating pixel values 187
See also 189
Filtering images using a median filter 189
How to do it... 190
How it works... 190
Applying directional filters to detect edges 191
How to do it... 192
How it works... 195
There's more... 198
Gradient operators 199
Gaussian derivatives 200
See also 201
Computing the Laplacian of an image 201
How to do it... 201
How it works... 203
There's more... 207
Enhancing the contrast of an image using the Laplacian 207
Difference of Gaussians 207
See also 208
Chapter 7: Extracting Lines, Contours, and Components 209
Detecting image contours with the Canny operator 209
How to do it... 210
How it works... 211
See also 213
Detecting lines in images with the Hough transform 213
Getting ready 213
How to do it... 214
How it works... 219
There's more... 222
Detecting circles 222
See also 225
Fitting a line to a set of points 225
How to do it... 225
How it works... 228
There's more... 228
Extracting the components' contours 229
How to do it... 229
How it works... 231
There's more... 232
Computing components' shape descriptors 234
How to do it... 234

How it works... 236
There's more... 237
Quadrilateral detection 238
Chapter 8: Detecting Interest Points 240
Detecting corners in an image 241
How to do it... 241
How it works... 247
There's more... 249
Good features to track 250
The feature detector's common interface 251
See also 252
Detecting features quickly 253
How to do it... 253
How it works... 254
There's more... 256
Adapted feature detection 256
See also 259
Detecting scale-invariant features 259
How to do it... 259
How it works... 261
There's more... 263
The SIFT feature-detection algorithm 263
See also 265
Detecting FAST features at multiple scales 265
How to do it... 266
How it works... 266
There's more... 268
The ORB feature-detection algorithm 268
See also 269
Chapter 9: Describing and Matching Interest Points 270
Matching local templates 271
How to do it... 271
How it works... 274
There's more... 276
Template matching 276
See also 277
Describing local intensity patterns 278
How to do it... 278
How it works... 281
There's more... 283
Cross-checking matches 283
The ratio test 284
Distance thresholding 285
See also 286
Describing keypoints with binary features 287

How to do it... 287
How it works... 289
There's more... 290
FREAK 290
See also 292
Chapter 10: Estimating Projective Relations in Images 293
Computing the fundamental matrix of an image pair 296
Getting ready 296
How to do it... 298
How it works... 301
See also 302
Matching images using a random sample consensus 302
How to do it... 303
How it works... 306
There's more... 308
Refining the fundamental matrix 308
Refining the matches 309
Computing a homography between two images 309
Getting ready 310
How to do it... 311
How it works... 313
There's more... 314
Detecting planar targets in an image 314
How to do it... 314
See also 317
Chapter 11: Reconstructing 3D Scenes 318
Digital image formation 319
Calibrating a camera 320
Getting ready 321
How to do it... 322
How it works... 326
There's more... 328
Calibration with known intrinsic parameters 329
Using a grid of circles for calibration 329
See also 329
Recovering the camera pose 330
How to do it... 330
How it works... 333
There's more... 334
cv::Viz – a 3D visualizer module 334
See also 336
Reconstructing a 3D scene from calibrated cameras 336
How to do it... 337
How it works... 343

There's more... 345
Decomposing a homography 345
Bundle adjustment 346
See also 346
Computing depth from a stereo image 346
Getting ready 347
How to do it... 348
How it works... 350
See also 351

Chapter 12: Processing Video Sequences 352
Reading video sequences 352
How to do it... 353
How it works... 355
There's more... 356
See also 357
Processing video frames 357
How to do it... 357
How it works... 359
There's more... 363
Processing a sequence of images 363
Using a frame processor class 364
See also 366
Writing video sequences 366
How to do it... 366
How it works... 367
There's more... 370
The codec four-character code 371
See also 372
Extracting the foreground objects in a video 372
How to do it... 374
How it works... 376
There's more... 377
The mixture of Gaussian method 377
See also 379

Chapter 13: Tracking Visual Motion 380
Tracing feature points in a video 381
How to do it... 381
How it works... 386
See also 387
Estimating the optical flow 388
Getting ready 389
How to do it... 390
How it works... 392
See also 394

Tracking an object in a video 394
How to do it... 395
How it works... 398
See also 402
Chapter 14: Learning from Examples 403
Recognizing faces using the nearest neighbors of local binary patterns 404
How to do it... 404
How it works... 407
See also 410
Finding objects and faces with a cascade of Haar features 411
Getting ready 411
How to do it... 413
How it works... 417
There's more... 420
Face detection with a Haar cascade 420
See also 421
Detecting objects and people using SVMs and histograms of oriented gradients 422
Getting ready 422
How to do it... 423
How it works... 427
There's more... 429
HOG visualization 430
People detection 432
Deep learning and convolutional neural networks (CNNs) 434
See also 435
Chapter 15: OpenCV Advanced Features 436
Face detection using deep learning 436
How to do it... 437
How it works... 441
See also 443
Object detection with YOLOv3 443
How to do it... 443
How it works... 447
See also 448
Enabling Halide to improve efficiency 449
How to do it... 449
How it works... 451
See also 451
OpenCV.js introduction 451
How to do it... 452
How it works... 454

Other Books You May Enjoy 455
Index 458

Preface

Augmented reality, driving assistance, video monitoring; more and more applications are now using computer vision and image analysis technologies, and yet we are still in the infancy of the development of new computerized systems capable of understanding our world through the sense of vision. And with the advent of powerful and affordable computing devices and visual sensors, it has never been easier to create sophisticated imaging applications.

A multitude of software tools and libraries manipulating images and videos are available, but for anyone who wishes to develop smart vision-based applications, the OpenCV library is the tool to use. OpenCV is an open source library containing more than 500 optimized algorithms for image and video analysis. Since its introduction in 1999, it has been largely adopted as the primary development tool by the community of researchers and developers in computer vision.

OpenCV was originally developed at Intel by a team led by Gary Bradski as an initiative to advance research in vision and promote the development of rich vision-based, CPU-intensive applications. After a series of beta releases, version 1.0 was launched in 2006. A second major release occurred in 2009 with the launch of OpenCV 2, which proposed important changes, especially the new C++ interface, which we use in this book. In 2012, OpenCV reshaped itself as a non-profit foundation (`https://opencv.org/`) relying on crowdfunding for its future development.

OpenCV 3 was introduced in 2013; changes were made mainly to improve the usability of the library. Its structure has been revised to remove the unnecessary dependencies, large modules have been split into smaller ones, and the API has been refined. This book is the fourth edition of *OpenCV Computer Vision Application Programming Cookbook*, and the first one that covers OpenCV 4. All the programming recipes of the previous editions have been reviewed and updated. We have also added new content and new chapters to provide readers with even better coverage of the essential functionalities of the library.

This book covers many of the library's features and explains how to use them to accomplish specific tasks. Our objective is not to provide detailed coverage of every option offered by the OpenCV functions and classes, but rather to give you the elements you need to build your applications from the ground up. We also explore fundamental concepts in image analysis, and describe some of the important algorithms in computer vision.

This book is an opportunity for you to get introduced to the world of image and video analysis—but this is just the beginning. The good news is that OpenCV continues to evolve and expand. Just consult the OpenCV online documentation at `https://opencv.org/` to stay updated about what the library can do for you. You can also visit the author's website at `http://www.laganiere.name/` for updated information about this cookbook.

Who this book is for

This cookbook is appropriate for novice C++ programmers who want to learn how to use the OpenCV library to build computer vision applications. It is also suitable for professional software developers who want to be introduced to the concepts of computer vision programming. It can be used as a companion book for university-level computer vision courses. It constitutes an excellent reference for graduate students and researchers in image processing and computer vision.

What this book covers

`Chapter 1`, *Playing with Images*, introduces the OpenCV library and shows you how to build simple applications that can read and display images. It also introduces basic OpenCV data structures.

`Chapter 2`, *Manipulating the Pixels*, explains how an image can be read. It describes different methods for scanning an image in order to perform an operation on each of its pixels.

`Chapter 3`, *Processing Color Images with Classes*, consists of recipes presenting various object-oriented design patterns that can help you to build better computer vision applications. It also discusses the concept of colors in images.

`Chapter 4`, *Counting the Pixels with Histograms*, shows you how to compute image histograms and how they can be used to modify an image. Different applications based on histograms are presented that achieve image segmentation, object detection, and image retrieval.

`Chapter 5`, *Transforming Images with Morphological Operations*, explores the concept of mathematical morphology. It presents different operators and how they can be used to detect edges, corners, and segments in images.

`Chapter 6`, *Filtering the Images*, teaches you the principles of frequency analysis and image filtering. It shows how low-pass and high-pass filters can be applied to images, and presents the concept of derivative operators.

`Chapter 7`, *Extracting Lines, Contours, and Components*, focuses on the detection of geometric image features. It explains how to extract contours, lines, and connected components in an image.

`Chapter 8`, *Detecting Interest Points*, describes various feature point detectors in images.

`Chapter 9`, *Describing and Matching Interest Points*, explains how descriptors of interest points can be computed and used to match points between images.

`Chapter 10`, *Estimating Projective Relations in Images*, explores the projective relations that exist between two images in the same scene. It also describes how to detect specific targets in an image.

`Chapter 11`, *Reconstructing 3D Scenes*, allows you to reconstruct the 3D elements of a scene from multiple images and recover the camera pose. It also includes a description of the camera calibration process.

`Chapter 12`, *Processing Video Sequences*, provides a framework to read and write a video sequence and to process its frames. It also shows you how it is possible to extract foreground objects moving in front of a camera.

`Chapter 13`, *Tracking Visual Motion*, addresses the visual tracking problem. It also shows you how to compute apparent motion in videos, and explains how to track moving objects in an image sequence.

`Chapter 14`, *Learning from Examples*, introduces basic concepts in machine learning. It shows how object classifiers can be built from image samples.

`Chapter 15`, *OpenCV Advanced Features*, covers the most advanced and newest features of OpenCV. This chapter introduces the reader to state-of-the-art deep learning models in artificial intelligence and machine learning. Deep learning is applied to object detection, autonomous cars, and facial recognition. This chapter will introduce you to OpenCV.js, a new binding that ports web technology directly from OpenCV.

To get the most out of this book

This cookbook is based on the C++ API of the OpenCV library. It is therefore assumed that you have some experience with the C++ language. In order to run the examples presented in the recipes and experiment with them, you need a good C++ development environment. Microsoft Visual Studio and Qt are two popular choices.

Download the example code files

You can download the example code files for this book from your account at www.packtpub.com.If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packtpub.com.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/OpenCV-4-Computer-Vision-Application-Programming-Cookbook-Fourth-Edition. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://www.packtpub.com/sites/default/files/downloads/9781789340723_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The `cv::Mat` variable's `image` refers to the input image, while `result` refers to the binary output image."

A block of code is set as follows:

```
// compute distance from target color
if (getDistanceToTargetColor(*it)<=maxDist) {
        *itout= 255;
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
Mat scores = outs[i].row(j).colRange(5, outs[i].cols);
Point classIdPoint;
double confidence;  // Get the value and location of the maximum score
```

Any command-line input or output is written as follows:

```
cd llvm_root
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "You then click on **Build Solution** in Visual Studio."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Sections

In this book, you will find several headings that appear frequently (*Getting ready*, *How to do it...*, *How it works...*, *There's more...*, and *See also*).

To give clear instructions on how to complete a recipe, use these sections as follows:

Getting ready

This section tells you what to expect in the recipe and describes how to set up any software or any preliminary settings required for the recipe.

How to do it…

This section contains the steps required to follow the recipe.

How it works…

This section usually consists of a detailed explanation of what happened in the previous section.

There's more…

This section consists of additional information about the recipe in order to make you more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packtpub.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packtpub.com.

1
Playing with Images

This chapter will teach you the basic elements of OpenCV and will show you how to accomplish the most fundamental image-processing tasks—reading, displaying, and saving images. However, before you can start with OpenCV, you need to install the library. This is a simple process that is explained in the first recipe of this chapter.

All your computer vision applications will involve the processing of images. This is why the most fundamental tool that OpenCV offers you is a data structure to handle images and matrices. It is a powerful data structure, with many useful attributes and methods. It also incorporates an advanced memory management model that greatly facilitates the development of applications. The last two recipes of this chapter will teach you how to use this important OpenCV data structure.

In this chapter, we will get you started with the OpenCV library. You will learn how to perform the following tasks:

- Installing the OpenCV library
- Loading, displaying, and saving images
- Exploring the `cv::Mat` data structure
- Defining regions of interest

Installing the OpenCV library

OpenCV is an open source library for developing computer vision applications that run on Windows, Linux, Android, and macOS. It can be used in both academic and commercial applications under a BSD license that allows you to use, distribute, and adapt it freely. This recipe will show you how to install the library on your machine.

Getting ready

When you visit the OpenCV official website at `https://opencv.org/`, you will find the latest release of the library, the online documentation, and many other useful resources concerning OpenCV.

How to do it...

The following steps will help take us through the installation, as follows:

1. From the OpenCV website, go to the downloads page that corresponds to the platform of your choice (Unix/Windows or Android). From there, you will be able to download the OpenCV package.
2. You will then uncompress it, normally under a directory with a name that corresponds to the library version (for example, in Windows, you can save the uncompressed directory under `C:\OpenCV4.0.0`).

   Once this is done, you will find a collection of files and directories that constitute the library at the chosen location. Notably, you will find the `modules` directory here, which contains all the source files. (Yes, it is open source!)

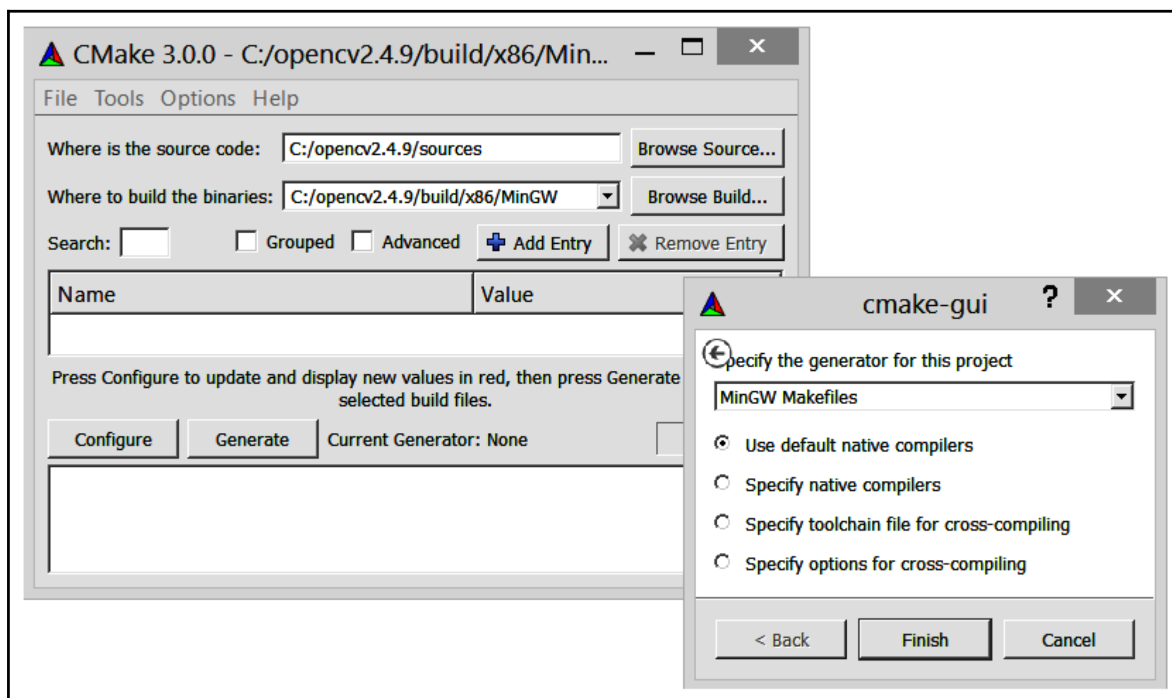
3. However, in order to complete the installation of the library and have it ready for use, you need to undertake an additional step—generating the binary files of the library for the environment of your choice. This is indeed the point where you have to make a decision on the target platform that you will use to create your OpenCV applications. Which operating system should you use? Windows or Linux? Which compiler should you use? Microsoft Visual Studio 2013 or MinGW? 32-bit or 64-bit? The **integrated development environment** (**IDE**) that you will use in your project development will also guide you to make these choices.

Note that if you are working under Windows with Visual Studio, the executable installation package will, most probably, not only install the library sources, but also install all of the precompiled binaries needed to build your applications. Check for the `build` directory; it should contain the `x64` and `x86` subdirectories (corresponding to the 64-bit and 32-bit versions). Within these subdirectories, you should find directories such as `vc14` and `vc15`; these contain the binaries for the different versions of Microsoft Visual Studio. In that case, you are ready to start using OpenCV. Therefore, you can skip the compilation step described in this recipe, unless you want a customized build with specific options.

4. To complete the installation process and build the OpenCV binaries, you need to use the **CMake** tool, available at `https://cmake.org/`.

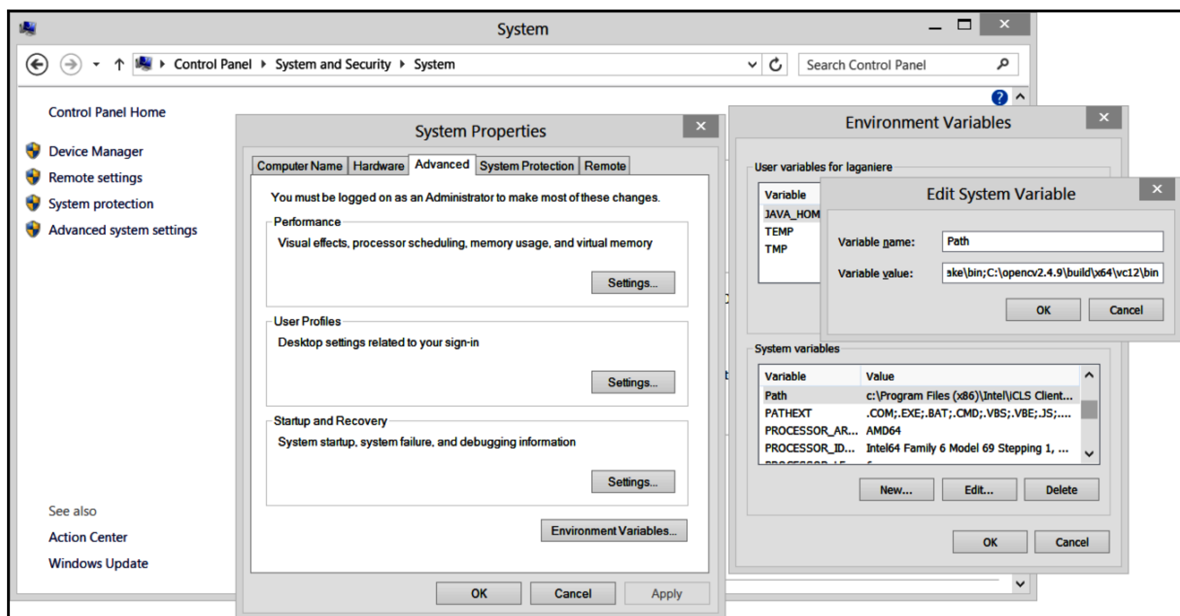
   CMake is another open source software tool designed to control the compilation process of a software system using platform-independent configuration files. It generates the required **makefiles** or **workspaces** needed for compiling a software library in your environment. Therefore, you need to download and install CMake.

5. Then, run it using the command line. Thereafter, it is easier to use CMake with its GUI (`cmake-gui`).
6. Specify the folder containing the OpenCV library source and the one that will contain the binaries. You need to click on **Configure** in order to select the compiler of your choice, and then click on **Configure** again as shown in the following screenshot:



7. You are now ready to generate your project files by clicking on the **Generate** button. These files will allow you to compile the library.

8. This is the last step of the installation process, which will make the library ready to be used under your development environment:
    1. If you have selected Visual Studio, then all you need to do is to open the top-level solution file that CMake has created for you (most probably, the `OpenCV.sln` file).
    2. You then click on **Build Solution** in Visual Studio.
    3. To get both a **Release** and a **Debug** build, you will have to repeat the compilation process twice, one for each configuration.
       The `bin` directory that is created contains the dynamic library files that your executable will call at runtime.
    4. Make sure to set your system `PATH` environment variable from the **Control Panel** such that your operating system can find the `dll` files when you run your applications:



9. In Linux environments, you will use the generated makefiles by running your `make` utility command. To complete the installation of all the directories, you also have to run a **Build INSTALL** or `sudo make INSTALL` command.

If you wish to use Qt as your IDE, the *There's more...* section of this recipe describes an alternative way to compile the OpenCV project.

How it works...

Since Version 2.2, the OpenCV library has been divided into several modules. These modules are built-in library files located in the `lib` directory. Some of the commonly used modules are as follows:

- The `opencv_core` module that contains the core functionalities of the library, in particular, basic data structures and arithmetic functions
- The `opencv_imgproc` module that contains the main image-processing functions
- The `opencv_highgui` module that contains the image and video reading and writing functions along with some user interface functions
- The `opencv_features2d` module that contains the feature point detectors and descriptors and the feature point matching framework
- The `opencv_calib3d` module that contains the camera calibration, two-view geometry estimation, and stereo functions
- The `opencv_video` module that contains the motion estimation, feature tracking, and foreground extraction functions and classes
- The `opencv_objdetect` module that contains the object detection functions such as the face and people detectors

The library also includes other utility modules that contain machine learning functions (`opencv_ml`), computational geometry algorithms (`opencv_flann`), contributed code (`opencv_contrib`), and many more. You will also find other specialized libraries that implement higher level functions, such as `opencv_photo` for computational photography and `opencv_stitching` for image-stitching algorithms. There is also a new branch that contains other library modules, which include non-free algorithms, non-stable modules, or experimental modules. This branch is on the `opencv-contrib` GitHub branch. When you compile your application, you will have to link your program with the libraries that contain the OpenCV functions you are using, linking it with the `opencv-contrib` folder.

All these modules have a header file associated with them (located in the `include` directory). A typical OpenCV C++ code will, therefore, start by including the required modules. For example (and this is the suggested declaration style), it will look like the following code:

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

Downloading the example code
You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files emailed directly to you.

You might see an OpenCV code starting with the following command:

```
#include "cv.h"
```

This is because it used the old style before the library was restructured into modules and became compatible with older definitions.

There's more...

The OpenCV website at `https://opencv.org/` contains detailed instructions on how to install the library. It also contains complete online documentation that includes several tutorials on the different components of the library.

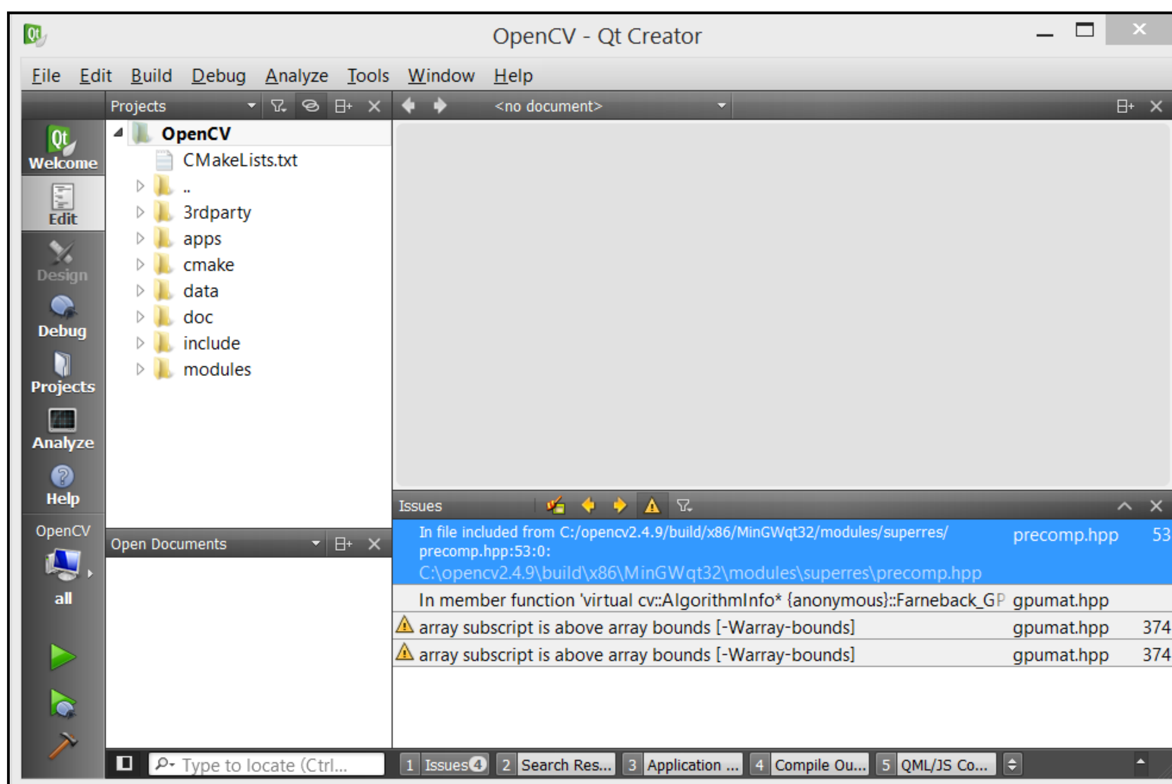
Using Qt for OpenCV developments

Qt is a cross-platform IDE for C++ applications developed as an open source project. It is offered under the **GNU Lesser General Public License** (**LGPL**) open source license as well as under a commercial (and paid) license for the development of proprietary projects. It is composed of two separate elements—a cross-platform IDE called Qt Creator, and a set of Qt class libraries and development tools. Using Qt to develop C++ applications has the following benefits:

- It is an open source initiative, developed by the Qt community, that gives you access to the source code of the different Qt components
- It is a cross-platform IDE, meaning that you can develop applications that can run on different operating systems, such as Windows, Linux, macOS, and so on
- It includes a complete and cross-platform GUI library that follows an effective object-oriented and event-driven model
- Qt also includes several cross-platform libraries that help you to develop multimedia, graphics, databases, multithreading, web applications, and many other interesting building blocks useful for designing advanced applications

You can download Qt from `https://www.qt.io/developers/`. When you install it, you will be offered the choice of different compilers. Under Windows, MinGW is an excellent alternative to the Visual Studio compilers.

Compiling the OpenCV library with Qt is particularly easy because it can read CMake files. Once OpenCV and CMake have been installed, simply select **Open File** or **Project...** from the Qt menu, and open the `CMakeLists.txt` file that you will find under the `sources` directory of OpenCV. This will create an OpenCV project that you will have built by clicking on **Build Project** in the Qt menu:



You might get a few warnings, but these can be overlooked without consequences.

The OpenCV developer site

OpenCV is an open source project that welcomes user contributions. You can access the developer site at `https://docs.opencv.org/`. Among other things, you can access the currently developed version of OpenCV. The community uses Git as its version control system. You then have to use it to check out the latest version of OpenCV. Git is also a free and open source software system; it is probably the best tool you can use to manage your own source code. You can download it from `https://git-scm.com/`.

See also

- The website of the author of this cookbook (`www.laganiere.name`) also presents step-by-step instructions on how to install the latest versions of the library.
- The *There's more...* section of the next recipe explains how to create an OpenCV project with Qt.

We've successfully learned how to install the OpenCV library. Now, let's move on to the next recipe!

Loading, displaying, and saving images

It is now time to run your first OpenCV application. Since OpenCV is about processing images, this task will show you how to perform the most fundamental operations needed in the development of imaging applications. These are loading an input image from a file, displaying an image on a window, applying a processing function, and storing an output image on a disk.

Getting ready

Using your favorite IDE (for example, MS Visual Studio or Qt), create a new console application with the `main` function that is ready to be filled.

How to do it...

Let's take a look at the following steps:

1. Include the header files, declaring the classes and functions you will use. Here, we simply want to display an image, so we need the `core` library that declares the image data structure and the `highgui` header file that contains all the graphical interface functions:

```
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
```

2. Our `main` function starts by declaring a variable that will hold the image. Under `OpenCV2`, define an object of the `cv::Mat` class:

```
cv::Mat image; // create an empty image
```

3. This definition creates an image sized *0 x 0*. This can be confirmed by accessing the `cv::Mat` size attributes:

```
std::cout << "This image is " << image.rows << " x "  << image.cols
<< std::endl;
```

4. Next, a simple call to the reading function will read an image from the file, decode it, and allocate the memory:

```
image= cv::imread("puppy.bmp"); // read an input image
```

5. You are now ready to use this image. However, you should first check whether the image has been correctly read (an error will occur if the file is not found, if the file is corrupted, or if it is not in a recognizable format) using the `empty()` function. The `empty` method returns `true` if no image data has been allocated:

```
if (image.empty()) {  // error handling
  // no image has been created...
  // possibly display an error message
  // and quit the application
  ...
}
```

6. The first thing you might want to do with this image is to display it. You can do this by using the functions of the `highgui` module. Start by declaring the window on which you want to display the images, and then specify the image to be shown on this special window:

```
// define the window (optional)
cv::namedWindow("Original Image");
// show the image
cv::imshow("Original Image", image);
```

As you can see, the window is identified by a name. You can reuse this window to display another image later, or you can create multiple windows with different names. When you run this application, you will see an image window as follows:

7. Now, you would normally apply some processing to the image. OpenCV offers a wide selection of processing functions, and several of them are explored in this book. Let's start with a very simple one that flips an image horizontally. Several image transformations in OpenCV can be performed **in-place**, meaning that the transformation is applied directly on the input image (no new image is created). This is the case with the flipping method. However, we can always create another matrix to hold the output result, and that is what we will do:

```
cv::Mat result; // we create another empty image
cv::flip(image,result,1); // positive for horizontal
                          // 0 for vertical,
                          // negative for both
```

8. We are going to display the result on another window:

```
cv::namedWindow("Output Image"); // the output window
cv::imshow("Output Image", result);
```

9. Since it is a console window that will terminate when it reaches the end of the `main` function, we add an extra `highgui` function to wait for a user keypress before ending the program:

```
cv::waitKey(0); // 0 to indefinitely wait for a key pressed
                // specifying a positive value will wait for
                // the given amount of msec
```

You can then see that the output image is displayed on a distinct window, as shown in the following screenshot:

10. Finally, you will probably want to save the processed image on your disk. This is done using the following `highgui` function:

```
cv::imwrite("output.bmp", result); // save result
```

The file extension determines which codec will be used to save the image. Other popular supported image formats are JPG, TIFF, and PNG.

How it works...

All classes and functions in the C++ API of OpenCV are defined within the `cv` namespace. You have two ways to access them. First, precede the `main` function's definition with the following declaration:

```
using namespace cv;
```

Alternatively, prefix all OpenCV class and function names by the namespace specification, that is, `cv::`, as we will do in this book. The use of the prefix makes the OpenCV classes and functions easier to identify.

The highgui module contains a set of functions that allows you to visualize and interact with your images easily. When you load an image with the imread function, you also have the option to read it as a gray-level image. This is very advantageous since several computer vision algorithms require gray-level images. Converting an input color image on the fly as you read it will save your time and minimize your memory usage. This can be done as follows:

```
// read the input image as a gray-scale image
image= cv::imread("puppy.bmp", cv::IMREAD_GRAYSCALE);
```

This will produce an image made of unsigned bytes (unsigned char in C++) that OpenCV designates with the CV_8U defined constant. Alternatively, it is sometimes necessary to read an image as a three-channel color image even if it has been saved as a gray-level image. This can be achieved by calling the imread function with a positive second argument:

```
// read the input image as a 3-channel color image
image= cv::imread("puppy.bmp", cv::IMREAD_COLOR);
```

This time, an image made of three bytes per pixel will be created, designated as CV_8UC3 in OpenCV. Of course, if your input image has been saved as a gray-level image, all three channels will contain the same value. Finally, if you wish to read the image in the format in which it has been saved, then simply input a negative value as the second argument. The number of channels in an image can be checked by using the channels method:

```
std::cout << "This image has " << image.channels() << " channel(s)";
```

Pay attention when you open an image with imread without specifying a full path (as we did here). In that case, the default directory will be used. When you run your application from the console, this directory is obviously one of your executable files. However, if you run the application directly from your IDE, the default directory will most often be the one that contains your project file. Consequently, make sure that your input image file is located in the right directory.

When you use imshow to display an image made up of integers (designated as CV_16U for 16-bit unsigned integers, or as CV_32S for 32-bit signed integers), the pixel values of this image will be divided by 256 first, in an attempt to make it displayable with 256 gray shades. Similarly, an image made of floating points will be displayed by assuming a range of possible values between 0.0 (displayed as black) and 1.0 (displayed as white). Values outside this defined range are displayed in white (for values above 1.0) or black (for values below 1.0).

The highghui module is very useful for building quick prototypal applications. When you are ready to produce a finalized version of your application, you will probably want to use the GUI module offered by your IDE in order to build an application with a more professional look.

Here, our application uses both input and output images. As an exercise, you should rewrite this simple program such that it takes advantage of the function's in-place processing, that is, by not declaring the output image and writing it instead:

```
cv::flip(image,image,1); // in-place processing
```

There's more...

The highgui module contains a rich set of functions that help you to interact with your images. Using these, your applications can react to mouse or key events. You can also draw shapes and write texts on images.

Clicking on images

You can program your mouse to perform specific operations when it is over one of the image windows you created. This is done by defining an appropriate **callback** function. A callback function is a function that you do not explicitly call but which is called by your application in response to specific events (here, the events that concern the mouse interacting with an image window). To be recognized by applications, callback functions need to have a specific signature and must be registered. In the case of the mouse event handler, the callback function must have the following signature:

```
void onMouse( int event, int x, int y, int flags, void* param);
```

The first parameter is an integer that is used to specify which type of mouse event has triggered the call to the callback function. The other two parameters are simply the pixel coordinates of the mouse location when the event occurred. The flags are used to determine which button was pressed when the mouse event was triggered. Finally, the last parameter is used to send an extra parameter to the function in the form of a pointer to an object. This callback function can be registered in the application through the following call:

```
cv::setMouseCallback("Original Image", onMouse,
reinterpret_cast<void*>(&image));
```

In this example, the `onMouse` function is associated with the image window called `Original Image`, and the address of the displayed image is passed as an extra parameter to the function. Now, if we define the `onMouse` callback function as shown in the following code, then each time the mouse is clicked, the value of the corresponding pixel will be displayed on the console (here, we assume that it is a gray-level image):

```
void onMouse( int event, int x, int y, int flags, void* param)  {

  cv::Mat *im= reinterpret_cast<cv::Mat*>(param);
  switch (event) {  // dispatch the event
  case cv::EVENT_LBUTTONDOWN: // left mouse button down event
        // display pixel value at (x,y)
        std::cout << "at (" << x << "," << y << ") value is: " <<
        static_cast<int>(im->at<uchar>(cv::Point(x,y))) << std::endl;
        break;
    }
}
```

Note that in order to obtain the pixel value at `(x,y)`, we used the `at` method of the `cv::Mat` object here; this is discussed in `Chapter 2`, *Manipulating the Pixels*. Other possible events that can be received by the mouse event callback function include `cv::EVENT_MOUSE_MOVE`, `cv::EVENT_LBUTTONUP`, `cv::EVENT_RBUTTONDOWN`, and `cv::EVENT_RBUTTONUP`.

Drawing on images

OpenCV also offers a few functions to draw shapes and write texts on images. The examples of basic shape-drawing functions are `circle`, `ellipse`, `line`, and `rectangle`. The following is an example of how to use the `circle` function:

```
cv::circle(image,  // destination image
            cv::Point(155,110), // center coordinate
            65,                 // radius
            0,                  // color (here black)
            3);                 // thickness
```

The `cv::Point` structure is often used in OpenCV methods and functions to specify a pixel coordinate. Note that here we assume that the drawing is done on a gray-level image; this is why the color is specified with a single integer. In the next recipe, you will learn how to specify a color value in the case of color images that use the `cv::Scalar` structure. It is also possible to write text on an image. This can be done as follows:

```
cv::putText(image,  // destination image
            "This is a dog.",// text
```

```
cv::Point(40,200), // text position
cv::FONT_HERSHEY_PLAIN,  // font type
2.0, // font scale
255, // text color (here white)
2);  // text thickness
```

Calling these two functions on our test image will then result in the following screenshot:

Let's see what happens when you run the example using Qt.

Running the example with Qt

If you wish to use Qt to run your OpenCV applications, you will need to create project files. For the example of this recipe, here is how the project file (`loadDisplaySave.pro`) will look:

```
QT += core
QT -= gui

TARGET = loadDisplaySave
CONFIG  += console
CONFIG  -= app_bundle

TEMPLATE = app

SOURCES += loadDisplaySave.cpp
INCLUDEPATH += C:\OpenCV4.0.0\build\include
LIBS += -LC:\OpenCV4.0.0\build\x86\MinGWqt32\lib \
-lopencv_core400 \
-lopencv_imgproc400 \
-lopencv_highgui400
```

This file shows you where to find the `include` and library files. It also lists the library modules that are used by the example. Make sure to use the library binaries compatible with the compiler that Qt is using. Note that if you download the source code of the examples for this book, you will find the `CMakeLists` files that you can open with Qt (or CMake) in order to create the associated projects.

See also

- The `cv::Mat` class is the data structure that is used to hold your images (and obviously, other matrix data). This data structure is at the core of all OpenCV classes and functions; the next recipe offers a detailed explanation of this data structure.
- You can download the source code of the examples for this book from `https://github.com/PacktPublishing/OpenCV-4-Computer-Vision-Application-Programming-Cookbook-Fourth-Edition`.

We've successfully learned how to load, display, and save images. Now, let's move on to the next recipe!

Exploring the cv::Mat data structure

In the previous recipe, you were introduced to the `cv::Mat` data structure. As mentioned, this is a key element of the library. It is used to manipulate images and matrices (in fact, an image is a matrix from a computational and mathematical point of view). Since you will be using this data structure extensively in your application developments, it is imperative that you become familiar with it. Notably, you will learn in this recipe that this data structure incorporates an elegant memory management mechanism, allowing efficient usage.

How to do it...

Let's write the following test program that will allow us to test the different properties of the `cv::Mat` data structure, as follows:

1. Include the `opencv` headers and a `c++ i/o stream` utility:

```
#include <iostream>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
```

2. We are going to create a function that generates a new gray image with a default value for all its pixels:

```
cv::Mat function() {
  // create image
  cv::Mat ima(500,500,CV_8U,50);
  // return it
  return ima;
}
```

3. In the `main` function, we are going to create six windows to show our results:

```
// define image windows
  cv::namedWindow("Image 1");
  cv::namedWindow("Image 2");
  cv::namedWindow("Image 3");
  cv::namedWindow("Image 4");
  cv::namedWindow("Image 5");
  cv::namedWindow("Image");
```

4. Now, we can start to create different mats (with different sizes, channels, and default values) and wait for the key to be pressed:

```
// create a new image made of 240 rows and 320 columns
  cv::Mat image1(240,320,CV_8U,100);

  cv::imshow("Image", image1); // show the image
  cv::waitKey(0); // wait for a key pressed

  // re-allocate a new image
  image1.create(200,200,CV_8U);
  image1= 200;

  cv::imshow("Image", image1); // show the image
  cv::waitKey(0); // wait for a key pressed

  // create a red color image
  // channel order is BGR
  cv::Mat image2(240,320,CV_8UC3,cv::Scalar(0,0,255));

  // or:
  // cv::Mat image2(cv::Size(320,240),CV_8UC3);
  // image2= cv::Scalar(0,0,255);

  cv::imshow("Image", image2); // show the image
  cv::waitKey(0); // wait for a key pressed
```

5. We are going to read an image with the `imread` function and copy it to another mat:

```
// read an image
  cv::Mat image3= cv::imread("puppy.bmp");

  // all these images point to the same data block
  cv::Mat image4(image3);
  image1= image3;

  // these images are new copies of the source image
  image3.copyTo(image2);
  cv::Mat image5= image3.clone();
```

6. Now, we are going to apply an image transformation (`flip`) to a copied image, show all images created, and wait for a keypress:

```
// transform the image for testing
  cv::flip(image3,image3,1);

  // check which images have been affected by the processing
  cv::imshow("Image 3", image3);
  cv::imshow("Image 1", image1);
  cv::imshow("Image 2", image2);
  cv::imshow("Image 4", image4);
  cv::imshow("Image 5", image5);
  cv::waitKey(0); // wait for a key pressed
```

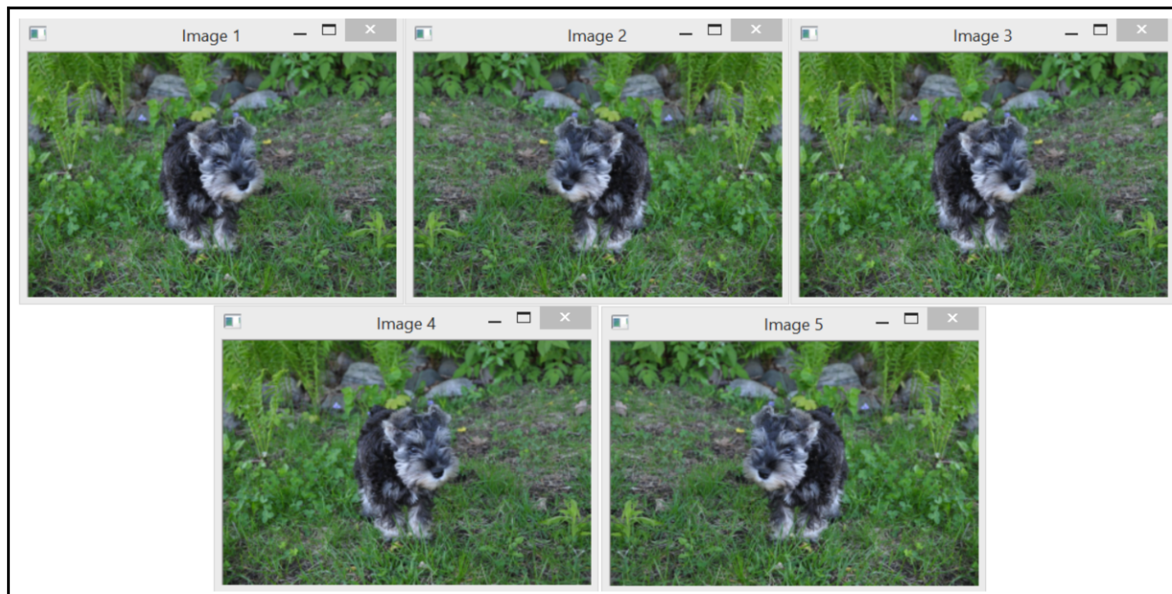
7. Now, we are going to use the function created before to generate a new gray mat:

```
// get a gray-level image from a function
  cv::Mat gray= function();

  cv::imshow("Image", gray); // show the image
  cv::waitKey(0); // wait for a key pressed
```

8. Finally, we are going to load a color image but convert it to gray in the loading process. Then, we will convert its values to float mat:

```
// read the image in gray scale
image1= cv::imread("puppy.bmp", IMREAD_GRAYSCALE);
image1.convertTo(image2,CV_32F,1/255.0,0.0);

cv::imshow("Image", image2); // show the image
cv::waitKey(0); // wait for a key pressed
```

Run this program and take a look at the following images produced:



Now, let's go behind the scenes to understand the code better.

How it works...

The `cv::Mat` data structure is essentially made up of two parts: a header and a data block. The header contains all the information associated with the matrix (size, number of channels, data type, and so on). The previous recipe showed you how to access some of the attributes of this structure contained in its header (for example, by using `cols`, `rows`, or `channels`). The data block holds all the pixel values of an image. The header contains a pointer variable that points to this data block; it is the `data` attribute. An important property of the `cv::Mat` data structure is the fact that the memory block is only copied when it is explicitly requested. Indeed, most operations will simply copy the `cv::Mat` header such that multiple objects will point to the same data block at the same time. This memory management model makes your applications more efficient while avoiding memory leaks, but its consequences have to be understood. The examples for this recipe illustrate this fact.

By default, the `cv::Mat` objects have a zero size when they are created, but you can also specify an initial size as follows:

```
// create a new image made of 240 rows and 320 columns
cv::Mat image1(240,320,CV_8U,100);
```

In this case, you also need to specify the type of each matrix element; `CV_8U` here, which corresponds to 1-byte pixel images. The letter `U` means it is unsigned. You can also declare signed numbers by using the letter `S`. For a color image, you would specify three channels (`CV_8UC3`). You can also declare integers (signed or unsigned) of size 16 and 32 (for example, `CV_16SC3`). You also have access to 32-bit and 64-bit floating-point numbers (for example, `CV_32F`).

Each element of an image (or a matrix) can be composed of more than one value (for example, the three channels of a color image); therefore, OpenCV has introduced a simple data structure that is used when pixel values are passed to functions. It is the `cv::Scalar` structure, which is generally used to hold one value or three values. For example, to create a color image initialized with red pixels, you will write the following code:

```
// create a red color image
// channel order is BGR
cv::Mat image2(240,320,CV_8UC3,cv::Scalar(0,0,255));
```

Similarly, the initialization of the gray-level image could also have been done using this structure by writing `cv::Scalar(100)`.

The image size also often needs to be passed to functions. We have already mentioned that the `cols` and `rows` attributes can be used to get the dimensions of a `cv::Mat` instance. The size information can also be provided through the `cv::Size` structure that simply contains the height and width of the matrix. The `size()` method allows you to obtain the current matrix size. It is the format that is used in many methods where a matrix size must be specified. For example, an image could be created as follows:

```
// create a non-initialized color image
cv::Mat image2(cv::Size(320,240),CV_8UC3);
```

The data block of an image can always be allocated or reallocated using the `create` method. When an image has been previously allocated, its old content is deallocated first. For reasons of efficiency, if the newly proposed size and type match the already existing size and type, then no new memory allocation is performed:

```
// re-allocate a new image
// (only if size or type are different)
image1.create(200,200,CV_8U);
```

When no more references point to a given `cv::Mat` object, the allocated memory is automatically released. This is very convenient because it avoids the common memory leak problems often associated with dynamic memory allocation in C++. This is a key mechanism in OpenCV 2 that is accomplished by having the `cv::Mat` class implement reference counting and shallow copying. Therefore, when an image is assigned to another one, the image data (that is, the pixels) is not copied; both the images will point to the same memory block. This also applies to images passed by value or returned by value. A reference count is kept, such that the memory will be released only when all the references to the image will be destroyed or assigned to another image:

```
// all these images point to the same data block
cv::Mat image4(image3);
image1= image3;
```

Any transformation applied to one of the preceding images will also affect the other images. If you wish to create a deep copy of the content of an image, use the `copyTo` method. In that case, the `create` method is called on the destination image. Another method that produces a copy of an image is the `clone` method, which creates an identical new image as follows:

```
// these images are new copies of the source image
image3.copyTo(image2);
cv::Mat image5= image3.clone();
```

If you need to copy an image into another image that does not necessarily have the same data type, you have to use the `convertTo` method:

```
// convert the image into a floating point image [0,1]
image1.convertTo(image2,CV_32F,1/255.0,0.0);
```

In this example, the source image is copied into a floating-point image. The method includes two optional parameters—a scaling factor and an offset. Note that both the images must, however, have the same number of channels.

The allocation model for the `cv::Mat` objects also allows you safely to write functions (or class methods) that return an image:

```
cv::Mat function() {
  // create image
  cv::Mat ima(240,320,CV_8U,cv::Scalar(100));
  // return it
  return ima;
}
```

We also call this function from our `main` function, as follows:

```
// get a gray-level image
cv::Mat gray= function();
```

If we do this, then the `gray` variable will now hold the image created by the function without extra memory allocation. Indeed, as we explained, only a shallow copy of the image will be transferred from the returned `cv::Mat` instance to the `gray` image. When the `ima` local variable goes out of scope, this variable is deallocated, but since the associated reference counter indicates that its internal image data is being referred to by another instance (that is, the `gray` variable), its memory block is not released.

It's worth noting that in the case of classes, you should be careful and not return image `class` attributes. Here is an example of an error-prone implementation:

```
class Test {
  // image attribute
  cv::Mat ima;
  public:
  // constructor creating a gray-level image
 Test() : ima(240,320,CV_8U,cv::Scalar(100)) {}

  // method return a class attribute, not a good idea...
  cv::Mat method() { return ima; }
};
```

Here, if a function calls the method of this class, it obtains a shallow copy of the image attributes. If later this copy is modified, the `class` attribute will also be surreptitiously modified, which can affect the subsequent behavior of the class (and vice versa). To avoid these kinds of errors, you should instead return a clone of the attribute.

There's more...

While you are manipulating the `cv::Mat` class, you will discover that OpenCV also includes several other related classes. It will be important for you to become familiar with them.

The input and output arrays

If you look at the OpenCV documentation, you will see that many methods and functions accept parameters of the `cv::InputArray` type as the input. This type is a simple proxy class introduced to generalize the concept of arrays in OpenCV, and thus, avoid the duplication of several versions of the same method or function with different input parameter types. It basically means that you can supply a `cv::Mat` object or other compatible types as an argument. This class is just an interface, so you should never declare it explicitly in your code. It is interesting to know that `cv::InputArray` can also be constructed from the popular `std::vector` class. This means that such objects can be used as the input to OpenCV methods and functions (as long as it makes sense to do so). Other compatible types are `cv::Scalar` and `cv::Vec`; this later structure will be presented in `Chapter 2`, *Manipulating the Pixels*. There is also a `cv::OutputArray` proxy class that is used to designate the arrays returned by some methods or functions.

See also

- The complete OpenCV documentation can be found at `https://docs.opencv.org/`.
- `Chapter 2`, *Manipulating the Pixels*, will show you how to access and modify the pixel values of an image represented by the `cv::Mat` class efficiently.

The next recipe will explain how to define a **region of interest** (**ROI**) inside an image.

Defining regions of interest

Sometimes, a processing function needs to be applied only to a portion of an image. OpenCV incorporates an elegant and simple mechanism to define a subregion in an image and manipulate it as a regular image. This recipe will teach you how to define an ROI inside an image.

Getting ready

Suppose we want to copy a small image onto a larger one. For example, let's say we want to insert the following small logo into our test image:

To do this, an ROI can be defined over which the copy operation can be applied. As we will see, the position of the ROI will determine where the logo will be inserted in the image.

How to do it...

Let's take a look at the following steps:

1. The first step consists of defining the ROI. We can use `Rect` to define the ROI:

```
cv::Rect myRoi= cv::Rect(image.cols-logo.cols, //ROI coordinates
                 image.rows-logo.rows,
                 logo.cols,logo.rows)
```

2. Once the ROI is defined, we can create a new mat applying the ROI to another mat and it can be manipulated as a regular `cv::Mat` instance. The key is that the ROI is indeed a `cv::Mat` object that points to the same data buffer as its parent image and has a header that specifies the coordinates of the ROI. Inserting the logo would then be accomplished as follows:

```
// define image ROI at image bottom-right
cv::Mat imageROI(image, myRoi);

// insert logo
logo.copyTo(imageROI);
```

Here, `image` is the destination image, and `logo` is the logo image (of a smaller size). The following image is then obtained by executing the previous code:

Now, let's go behind the scenes to understand the code better.

How it works...

One way to define an ROI is to use a `cv::Rect` instance. As the name indicates, it describes a rectangular region by specifying the position of the upper-left corner (the first two parameters of the constructor) and the size of the rectangle (the width and height are given in the last two parameters). In our example, we used the size of the image and the size of the logo in order to determine the position where the logo would cover the bottom-right corner of the image. Obviously, the ROI should always be completely inside the parent image.

The ROI can also be described using row and column ranges. A range is a continuous sequence from a start index to an end index (excluding both). The `cv::Range` structure is used to represent this concept. Therefore, an ROI can be defined from two ranges; in our example, the ROI could have been equivalently defined as follows:

```
imageROI= image(cv::Range(image.rows-logo.rows,image.rows),
                cv::Range(image.cols-logo.cols,image.cols));
```

In this case, the `operator()` function of `cv ::Mat` returns another `cv::Mat` instance that can then be used in subsequent calls. Any transformation of the ROI will affect the original image in the corresponding area because the image and the ROI share the same image data. Since the definition of an ROI does not include the copying of data, it is executed in a constant amount of time, no matter the size of the ROI.

If one wants to define an ROI made of some lines of an image, the following call could be used:

```
cv::Mat imageROI= image.rowRange(start,end);
```

Similarly, for an ROI made of some image columns, the following could be used:

```
cv::Mat imageROI= image.colRange(start,end);
```

There's more...

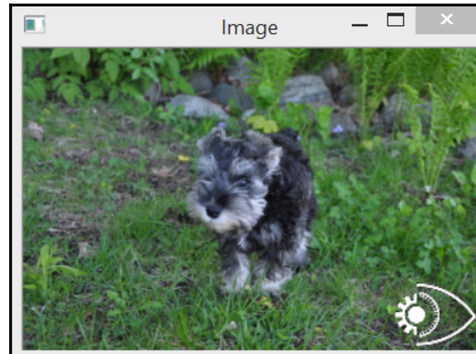
The OpenCV methods and functions include many optional parameters that are not discussed in the recipes of this book. When you wish to use a function for the first time, you should always take the time to look at the documentation to learn more about the possible options that this function offers. One very common option is the possibility to define image masks.

Using image masks

Some OpenCV operations allow you to define a mask that will limit the applicability of a given function or method, which is normally supposed to operate on all the image pixels. A mask is an 8-bit image that should be nonzero at all locations where you want an operation to be applied. At the pixel locations that correspond to the zero values of the mask, the image is untouched. For example, the `copyTo` method can be called with a mask. We can use it here to copy only the white portion of the logo shown previously, as follows:

```
// define image ROI at image bottom-right
imageROI= image(cv::Rect(image.cols-logo.cols,image.rows-logo.rows,
logo.cols,logo.rows));
// use the logo as a mask (must be gray-level)
cv::Mat mask(logo);

// insert by copying only at locations of non-zero mask
logo.copyTo(imageROI,mask);
```

The following image is obtained by executing the previous code:



The background of our logo was black (therefore, it had the value `0`); therefore, it was easy to use it as both the copied image and the mask. Of course, you can define the mask of your choice in your application; most OpenCV pixel-based operations give you the opportunity to use masks.

See also

- The `row` and `col` methods that will be used in the *Scanning an image with neighbor access* recipe of `Chapter 2`, *Manipulating the Pixels*. These are special cases of the `rowRange` and `colRange` methods in which the start and end indexes are equal in order to define a single-line or single-column ROI.