Software That Sees

Learning

OpenCV

Computer Vision with the OpenCV Library

O'REILLY®

Gary Bradski & Adrian Kaehler

O'REILLY®

Learning OpenCV

Learning OpenCV puts you in the middle of the rapidly expanding field of computer vision. Written by the creators of the free, open source OpenCV library, this book introduces you to computer vision and demonstrates how you can quickly build applications that enable computers to "see" and make decisions based on the data they acquire.

Computer vision is everywhere—in security systems, manufacturing inspection systems, medical image analysis, unmanned aerial vehicles, and more. It ties Google Maps and Google Earth together, checks the pixels on LCD screens, and makes sure the stitches in your shirt are sewn properly. OpenCV provides an easy-to-use computer vision framework and a comprehensive library with more than 500 functions that can run vision code in real time.

Learning OpenCV will teach any developer or hobbyist to use the framework quickly with the help of hands-on exercises in each chapter. This book includes:

- A thorough introduction to OpenCV
- Getting input from cameras
- Transforming images
- Segmenting images and shape matching
- Pattern recognition, including face detection
- Tracking and motion in two and three dimensions
- 3D reconstruction from stereo vision
- Machine learning algorithms

Getting machines to see is a challenging but entertaining goal. Whether you want to build simple or sophisticated vision applications, *Learning OpenCV* is the book you need to get started.

www.oreilly.com

US $49.99 **CAN $49.99**
ISBN: 978-0-596-51613-0

"This library is useful for practitioners, and is an excellent tool for those entering the field; it is a set of computer vision algorithms that work as advertised."

—William T. Freeman, Computer Science and Artificial Intelligence Laboratory, MIT

"Learning OpenCV *will likely occupy a prominent spot on the bookshelf of almost anyone working in computer vision."*

—David Lowe, Professor of Computer Science, University of British Columbia

Dr. Gary Rost Bradski is a consulting professor in the CS department at Stanford University's AI Lab and senior scientist at Willow Garage, a robotics research institute/incubator.

Dr. Adrian Kaehler, senior scientist at Applied Minds Corporation, conducts research in machine learning, statistical modeling, computer vision, and robotics.

Learning OpenCV

Gary Bradski and Adrian Kaehler

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Sebastopol · Taipei · Tokyo

Learning OpenCV
by Gary Bradski and Adrian Kaehler

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Editor: Mike Loukides
Production Editor: Rachel Monaghan
Production Services: Newgen Publishing and Data Services

Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:

September 2008: First Edition.

ISBN: 978-0-596-51613-0
[M]

Contents

Preface

This book provides a working guide to the Open Source Computer Vision Library (OpenCV) and also provides a general background to the field of computer vision sufficient to use OpenCV effectively.

Purpose

Computer vision is a rapidly growing field, partly as a result of both cheaper and more capable cameras, partly because of affordable processing power, and partly because vision algorithms are starting to mature. OpenCV itself has played a role in the growth of computer vision by enabling thousands of people to do more productive work in vision. With its focus on real-time vision, OpenCV helps students and professionals efficiently implement projects and jump-start research by providing them with a computer vision and machine learning infrastructure that was previously available only in a few mature research labs. The purpose of this text is to:

- Better document OpenCV—detail what function calling conventions really mean and how to use them correctly.
- Rapidly give the reader an intuitive understanding of how the vision algorithms work.
- Give the reader some sense of what algorithm to use and when to use it.
- Give the reader a boost in implementing computer vision and machine learning algorithms by providing many working coded examples to start from.
- Provide intuitions about how to fix some of the more advanced routines when something goes wrong.

Simply put, this is the text the authors wished we had in school and the coding reference book we wished we had at work.

This book documents a tool kit, OpenCV, that allows the reader to do interesting and fun things rapidly in computer vision. It gives an intuitive understanding as to how the algorithms work, which serves to guide the reader in designing and debugging vision

applications and also to make the formal descriptions of computer vision and machine learning algorithms in other texts easier to comprehend and remember.

After all, it is easier to understand complex algorithms and their associated math when you start with an intuitive grasp of how those algorithms work.

Who This Book Is For

This book contains descriptions, working coded examples, and explanations of the computer vision tools contained in the OpenCV library. As such, it should be helpful to many different kinds of users.

Professionals
: For those practicing professionals who need to rapidly implement computer vision systems, the sample code provides a quick framework with which to start. Our descriptions of the intuitions behind the algorithms can quickly teach or remind the reader how they work.

Students
: As we said, this is the text we wish had back in school. The intuitive explanations, detailed documentation, and sample code will allow you to boot up faster in computer vision, work on more interesting class projects, and ultimately contribute new research to the field.

Teachers
: Computer vision is a fast-moving field. We've found it effective to have the students rapidly cover an accessible text while the instructor fills in formal exposition where needed and supplements with current papers or guest lecturers from experts. The students can meanwhile start class projects earlier and attempt more ambitious tasks.

Hobbyists
: Computer vision is fun, here's how to hack it.

We have a strong focus on giving readers enough intuition, documentation, and working code to enable rapid implementation of real-time vision applications.

What This Book Is Not

This book is not a formal text. We do go into mathematical detail at various points,* but it is all in the service of developing deeper intuitions behind the algorithms or to make clear the implications of any assumptions built into those algorithms. We have not attempted a formal mathematical exposition here and might even incur some wrath along the way from those who do write formal expositions.

This book is not for theoreticians because it has more of an "applied" nature. The book will certainly be of general help, but is not aimed at any of the specialized niches in computer vision (e.g., medical imaging or remote sensing analysis).

* Always with a warning to more casual users that they may skip such sections.

That said, it is the belief of the authors that having read the explanations here first, a student will not only learn the theory better but remember it longer. Therefore, this book would make a good adjunct text to a theoretical course and would be a great text for an introductory or project-centric course.

About the Programs in This Book

All the program examples in this book are based on OpenCV version 2.0. The code should definitely work under Linux or Windows and probably under OS-X, too. Source code for the examples in the book can be fetched from this book's website (*http://www.oreilly.com/catalog/9780596516130*). OpenCV can be loaded from its source forge site (*http://sourceforge.net/projects/opencvlibrary*).

OpenCV is under ongoing development, with official releases occurring once or twice a year. As a rule of thumb, you should obtain your code updates from the source forge CVS server (*http://sourceforge.net/cvs/?group_id=22870*).

Prerequisites

For the most part, readers need only know how to program in C and perhaps some C++. Many of the math sections are optional and are labeled as such. The mathematics involves simple algebra and basic matrix algebra, and it assumes some familiarity with solution methods to least-squares optimization problems as well as some basic knowledge of Gaussian distributions, Bayes' law, and derivatives of simple functions.

The math is in support of developing intuition for the algorithms. The reader may skip the *math* and the algorithm descriptions, using only the function definitions and code examples to get vision applications up and running.

How This Book Is Best Used

This text need not be read in order. It can serve as a kind of user manual: look up the function when you need it; read the function's description if you want the gist of how it works "under the hood". The intent of this book is more tutorial, however. It gives you a basic understanding of computer vision along with details of how and when to use selected algorithms.

This book was written to allow its use as an adjunct or as a primary textbook for an undergraduate or graduate course in computer vision. The basic strategy with this method is for students to read the book for a rapid overview and then supplement that reading with more formal sections in other textbooks and with papers in the field. There are exercises at the end of each chapter to help test the student's knowledge and to develop further intuitions.

You could approach this text in any of the following ways.

Grab Bag
: Go through Chapters 1–3 in the first sitting, then just hit the appropriate chapters or sections as you need them. This book does not have to be read in sequence, except for Chapters 11 and 12 (Calibration and Stereo).

Good Progress
: Read just two chapters a week until you've covered Chapters 1–12 in six weeks (Chapter 13 is a special case, as discussed shortly). Start on projects and start in detail on selected areas in the field, using additional texts and papers as appropriate.

The Sprint
: Just cruise through the book as fast as your comprehension allows, covering Chapters 1–12. Then get started on projects and go into detail on selected areas in the field using additional texts and papers. This is probably the choice for professionals, but it might also suit a more advanced computer vision course.

Chapter 13 is a long chapter that gives a general background to machine learning in addition to details behind the machine learning algorithms implemented in OpenCV and how to use them. Of course, machine learning is integral to object recognition and a big part of computer vision, but it's a field worthy of its own book. Professionals should find this text a suitable launching point for further explorations of the literature—or for just getting down to business with the code in that part of the library. This chapter should probably be considered optional for a typical computer vision class.

This is how the authors like to teach computer vision: Sprint through the course content at a level where the students get the gist of how things work; then get students started on meaningful class projects while the instructor supplies depth and formal rigor in selected areas by drawing from other texts or papers in the field. This same method works for quarter, semester, or two-term classes. Students can get quickly up and running with a general understanding of their vision task and working code to match. As they begin more challenging and time-consuming projects, the instructor helps them develop and debug complex systems. For longer courses, the projects themselves can become instructional in terms of project management. Build up working systems first; refine them with more knowledge, detail, and research later. The goal in such courses is for each project to aim at being worthy of a conference publication and with a few project papers being published subsequent to further (postcourse) work.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, file extensions, path names, directories, and Unix utilities.

`Constant width`
: Indicates commands, options, switches, variables, attributes, keys, functions, types, classes, namespaces, methods, modules, properties, parameters, values, objects,

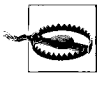
events, event handlers, XMLtags, HTMLtags, the contents of files, or the output from commands.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user. Also used for emphasis in code samples.

`Constant width italic`
: Shows text that should be replaced with user-supplied values.

[. . .]
: Indicates a reference to the bibliography.

Shows text that should be replaced with user-supplied values. his icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

OpenCV is free for commercial or research use, and we have the same policy on the code examples in the book. Use them at will for homework, for research, or for commercial products. We would very much appreciate referencing this book when you do, but it is not required. Other than how it helped with your homework projects (which is best kept a secret), we would like to hear how you are using computer vision for academic research, teaching courses, and in commercial products when you do use OpenCV to help you. Again, not required, but you are always invited to drop us a line.

Safari® Books Online

When you see a Safari® Books Online icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http://safari.oreilly.com*.

We'd Like to Hear from You

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list examples and any plans for future editions. You can access this information at:

http://www.oreilly.com/catalog/9780596516130/

You can also send messages electronically. To be put on the mailing list or request a catalog, send an email to:

info@oreilly.com

To comment on the book, send an email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

http://www.oreilly.com

Acknowledgments

A long-term open source effort sees many people come and go, each contributing in different ways. The list of contributors to this library is far too long to list here, but see the *.../opencv/docs/HTML/Contributors/doc_contributors.html* file that ships with OpenCV.

Thanks for Help on OpenCV

Intel is where the library was born and deserves great thanks for supporting this project the whole way through. Open source needs a champion and enough development support in the beginning to achieve critical mass. Intel gave it both. There are not many other companies where one could have started and maintained such a project through good times and bad. Along the way, OpenCV helped give rise to—and now takes (optional) advantage of—Intel's Integrated Performance Primitives, which are hand-tuned assembly language routines in vision, signal processing, speech, linear algebra, and more. Thus the lives of a great commercial product and an open source product are intertwined.

Mark Holler, a research manager at Intel, allowed OpenCV to get started by knowingly turning a blind eye to the inordinate amount of time being spent on an unofficial project back in the library's earliest days. As divine reward, he now grows wine up in Napa's Mt. Vieder area. Stuart Taylor in the Performance Libraries group at Intel enabled OpenCV by letting us "borrow" part of his Russian software team. Richard Wirt was key to its continued growth and survival. As the first author took on management responsibility at Intel, lab director Bob Liang let OpenCV thrive; when Justin Rattner became CTO, we were able to put OpenCV on a more firm foundation under Software Technology Lab—supported by software guru Shinn-Horng Lee and indirectly under his manager, Paul Wiley. Omid Moghadam helped advertise OpenCV in the early days. Mohammad Haghighat and Bill Butera were great as technical sounding boards. Nuriel Amir, Denver

Dash, John Mark Agosta, and Marzia Polito were of key assistance in launching the machine learning library. Rainer Lienhart, Jean-Yves Bouguet, Radek Grzeszczuk, and Ara Nefian were able technical contributors to OpenCV and great colleagues along the way; the first is now a professor, the second is now making use of OpenCV in some well-known Google projects, and the others are staffing research labs and start-ups. There were many other technical contributors too numerous to name.

On the software side, some individuals stand out for special mention, especially on the Russian software team. Chief among these is the Russian lead programmer Vadim Pisarevsky, who developed large parts of the library and also managed and nurtured the library through the lean times when boom had turned to bust; he, if anyone, is the true hero of the library. His technical insights have also been of great help during the writing of this book. Giving him managerial support and protection in the lean years was Valery Kuriakin, a man of great talent and intellect. Victor Eruhimov was there in the beginning and stayed through most of it. We thank Boris Chudinovich for all of the contour components.

Finally, very special thanks go to Willow Garage [WG], not only for its steady financial backing to OpenCV's future development but also for supporting one author (and providing the other with snacks and beverages) during the final period of writing this book.

Thanks for Help on the Book

While preparing this book, we had several key people contributing advice, reviews, and suggestions. Thanks to John Markoff, Technology Reporter at the *New York Times* for encouragement, key contacts, and general writing advice born of years in the trenches. To our reviewers, a special thanks go to Evgeniy Bart, physics postdoc at CalTech, who made many helpful comments on every chapter; Kjerstin Williams at Applied Minds, who did detailed proofs and verification until the end; John Hsu at Willow Garage, who went through all the example code; and Vadim Pisarevsky, who read each chapter in detail, proofed the function calls and the code, and also provided several coding examples. There were many other partial reviewers. Jean-Yves Bouguet at Google was of great help in discussions on the calibration and stereo chapters. Professor Andrew Ng at Stanford University provided useful early critiques of the machine learning chapter. There were numerous other reviewers for various chapters—our thanks to all of them. Of course, any errors result from our own ignorance or misunderstanding, not from the advice we received.

Finally, many thanks go to our editor, Michael Loukides, for his early support, numerous edits, and continued enthusiasm over the long haul.

Gary Adds . . .

With three young kids at home, my wife Sonya put in more work to enable this book than I did. Deep thanks and love—even OpenCV gives her recognition, as you can see in the face detection section example image. Further back, my technical beginnings started with the physics department at the University of Oregon followed by undergraduate years at

UC Berkeley. For graduate school, I'd like to thank my advisor Steve Grossberg and Gail Carpenter at the Center for Adaptive Systems, Boston University, where I first cut my academic teeth. Though they focus on mathematical modeling of the brain and I have ended up firmly on the engineering side of AI, I think the perspectives I developed there have made all the difference. Some of my former colleagues in graduate school are still close friends and gave advice, support, and even some editing of the book: thanks to Frank Guenther, Andrew Worth, Steve Lehar, Dan Cruthirds, Allen Gove, and Krishna Govindarajan.

I specially thank Stanford University, where I'm currently a consulting professor in the AI and Robotics lab. Having close contact with the best minds in the world definitely rubs off, and working with Sebastian Thrun and Mike Montemerlo to apply OpenCV on Stanley (the robot that won the $2M DARPA Grand Challenge) and with Andrew Ng on STAIR (one of the most advanced personal robots) was more technological fun than a person has a right to have. It's a department that is currently hitting on all cylinders and simply a great environment to be in. In addition to Sebastian Thrun and Andrew Ng there, I thank Daphne Koller for setting high scientific standards, and also for letting me hire away some key interns and students, as well as Kunle Olukotun and Christos Kozyrakis for many discussions and joint work. I also thank Oussama Khatib, whose work on control and manipulation has inspired my current interests in visually guided robotic manipulation. Horst Haussecker at Intel Research was a great colleague to have, and his own experience in writing a book helped inspire my effort.

Finally, thanks once again to Willow Garage for allowing me to pursue my lifelong robotic dreams in a great environment featuring world-class talent while also supporting my time on this book and supporting OpenCV itself.

Adrian Adds . . .

Coming from a background in theoretical physics, the arc that brought me through supercomputer design and numerical computing on to machine learning and computer vision has been a long one. Along the way, many individuals stand out as key contributors. I have had many wonderful teachers, some formal instructors and others informal guides. I should single out Professor David Dorfan of UC Santa Cruz and Hartmut Sadrozinski of SLAC for their encouragement in the beginning, and Norman Christ for teaching me the fine art of computing with the simple edict that "if you can not make the computer do it, you don't know what you are talking about". Special thanks go to James Guzzo, who let me spend time on this sort of thing at Intel—even though it was miles from what I was supposed to be doing—and who encouraged my participation in the Grand Challenge during those years. Finally, I want to thank Danny Hillis for creating the kind of place where all of this technology can make the leap to wizardry and for encouraging my work on the book while at Applied Minds.

I also would like to thank Stanford University for the extraordinary amount of support I have received from them over the years. From my work on the Grand Challenge team with Sebastian Thrun to the STAIR Robot with Andrew Ng, the Stanford AI Lab was always

generous with office space, financial support, and most importantly ideas, enlightening conversation, and (when needed) simple instruction on so many aspects of vision, robotics, and machine learning. I have a deep gratitude to these people, who have contributed so significantly to my own growth and learning.

No acknowledgment or thanks would be meaningful without a special thanks to my lady Lyssa, who never once faltered in her encouragement of this project or in her willingness to accompany me on trips up and down the state to work with Gary on this book. My thanks and my love go to her.

CHAPTER 1

Overview

What Is OpenCV?

OpenCV [OpenCV] is an open source (see *http://opensource.org*) computer vision library available from *http://SourceForge.net/projects/opencvlibrary*. The library is written in C and C++ and runs under Linux, Windows and Mac OS X. There is active development on interfaces for Python, Ruby, Matlab, and other languages.

OpenCV was designed for computational efficiency and with a strong focus on real-time applications. OpenCV is written in optimized C and can take advantage of multicore processors. If you desire further automatic optimization on Intel architectures [Intel], you can buy Intel's Integrated Performance Primitives (IPP) libraries [IPP], which consist of low-level optimized routines in many different algorithmic areas. OpenCV automatically uses the appropriate IPP library at runtime if that library is installed.

One of OpenCV's goals is to provide a simple-to-use computer vision infrastructure that helps people build fairly sophisticated vision applications quickly. The OpenCV library contains over 500 functions that span many areas in vision, including factory product inspection, medical imaging, security, user interface, camera calibration, stereo vision, and robotics. Because computer vision and machine learning often go hand-in-hand, OpenCV also contains a full, general-purpose Machine Learning Library (MLL). This sublibrary is focused on statistical pattern recognition and clustering. The MLL is highly useful for the vision tasks that are at the core of OpenCV's mission, but it is general enough to be used for any machine learning problem.

Who Uses OpenCV?

Most computer scientists and practical programmers are aware of some facet of the role that computer vision plays. But few people are aware of all the ways in which computer vision is used. For example, most people are somewhat aware of its use in surveillance, and many also know that it is increasingly being used for images and video on the Web. A few have seen some use of computer vision in game interfaces. Yet few people realize that most aerial and street-map images (such as in Google's Street View) make heavy

use of camera calibration and image stitching techniques. Some are aware of niche applications in safety monitoring, unmanned flying vehicles, or biomedical analysis. But few are aware how pervasive machine vision has become in manufacturing: virtually everything that is mass-produced has been automatically inspected at some point using computer vision.

The open source license for OpenCV has been structured such that you can build a commercial product using all or part of OpenCV. You are under no obligation to open-source your product or to return improvements to the public domain, though we hope you will. In part because of these liberal licensing terms, there is a large user community that includes people from major companies (IBM, Microsoft, Intel, SONY, Siemens, and Google, to name only a few) and research centers (such as Stanford, MIT, CMU, Cambridge, and INRIA). There is a Yahoo groups forum where users can post questions and discussion at *http://groups.yahoo.com/group/OpenCV*; it has about 20,000 members. OpenCV is popular around the world, with large user communities in China, Japan, Russia, Europe, and Israel.

Since its alpha release in January 1999, OpenCV has been used in many applications, products, and research efforts. These applications include stitching images together in satellite and web maps, image scan alignment, medical image noise reduction, object analysis, security and intrusion detection systems, automatic monitoring and safety systems, manufacturing inspection systems, camera calibration, military applications, and unmanned aerial, ground, and underwater vehicles. It has even been used in sound and music recognition, where vision recognition techniques are applied to sound spectrogram images. OpenCV was a key part of the vision system in the robot from Stanford, "Stanley", which won the $2M DARPA Grand Challenge desert robot race [Thrun06].

What Is Computer Vision?

Computer vision* is the transformation of data from a still or video camera into either a decision or a new representation. All such transformations are done for achieving some particular goal. The input data may include some contextual information such as "the camera is mounted in a car" or "laser range finder indicates an object is 1 meter away". The decision might be "there is a person in this scene" or "there are 14 tumor cells on this slide". A new representation might mean turning a color image into a grayscale image or removing camera motion from an image sequence.

Because we are such visual creatures, it is easy to be fooled into thinking that computer vision tasks are easy. How hard can it be to find, say, a car when you are staring at it in an image? Your initial intuitions can be quite misleading. The human brain divides the vision signal into many channels that stream different kinds of information into your brain. Your brain has an attention system that identifies, in a task-dependent

* Computer vision is a vast field. This book will give you a basic grounding in the field, but we also recommend texts by Trucco [Trucco98] for a simple introduction, Forsyth [Forsyth03] as a comprehensive reference, and Hartley [Hartley06] and Faugeras [Faugeras93] for how 3D vision really works.

way, important parts of an image to examine while suppressing examination of other areas. There is massive feedback in the visual stream that is, as yet, little understood. There are widespread associative inputs from muscle control sensors and all of the other senses that allow the brain to draw on cross-associations made from years of living in the world. The feedback loops in the brain go back to all stages of processing including the hardware sensors themselves (the eyes), which mechanically control lighting via the iris and tune the reception on the surface of the retina.

In a machine vision system, however, a computer receives a grid of numbers from the camera or from disk, and that's it. For the most part, there's no built-in pattern recognition, no automatic control of focus and aperture, no cross-associations with years of experience. For the most part, vision systems are still fairly naïve. Figure 1-1 shows a picture of an automobile. In that picture we see a side mirror on the driver's side of the car. What the *computer* "sees" is just a grid of numbers. Any given number within that grid has a rather large noise component and so by itself gives us little information, but this grid of numbers is all the computer "sees". Our task then becomes to turn this noisy grid of numbers into the perception: "side mirror". Figure 1-2 gives some more insight into why computer vision is so hard.

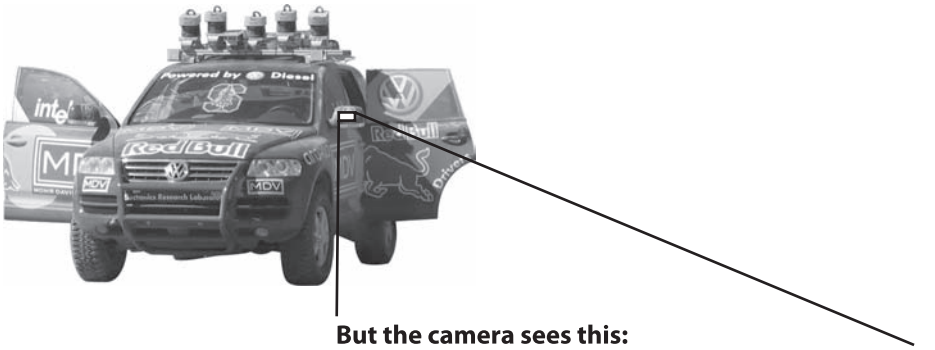


Figure 1-1. To a computer, the car's side mirror is just a grid of numbers

In fact, the problem, as we have posed it thus far, is worse than hard; it is formally impossible to solve. Given a two-dimensional (2D) view of a 3D world, there is no unique way to reconstruct the 3D signal. Formally, such an ill-posed problem has no unique or definitive solution. The same 2D image could represent any of an infinite combination of 3D scenes, even if the data were perfect. However, as already mentioned, the data is

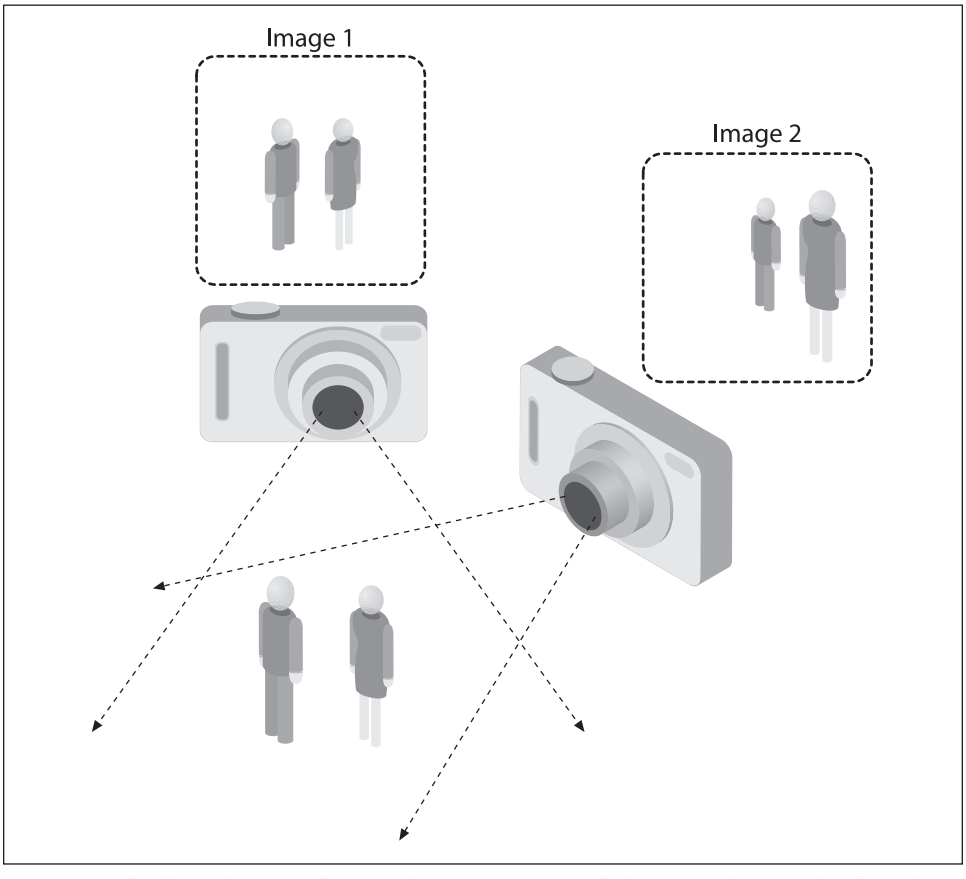


Figure 1-2. The ill-posed nature of vision: the 2D appearance of objects can change radically with viewpoint

corrupted by noise and distortions. Such corruption stems from variations in the world (weather, lighting, reflections, movements), imperfections in the lens and mechanical setup, finite integration time on the sensor (motion blur), electrical noise in the sensor or other electronics, and compression artifacts after image capture. Given these daunting challenges, how can we make any progress?

In the design of a practical system, additional contextual knowledge can often be used to work around the limitations imposed on us by visual sensors. Consider the example of a mobile robot that must find and pick up staplers in a building. The robot might use the facts that a desk is an object found inside offices and that staplers are mostly found on desks. This gives an implicit size reference; staplers must be able to fit on desks. It also helps to eliminate falsely "recognizing" staplers in impossible places (e.g., on the ceiling or a window). The robot can safely ignore a 200-foot advertising blimp shaped like a stapler because the blimp lacks the prerequisite wood-grained background of a desk. In contrast, with tasks such as image retrieval, all stapler images in a database

may be of real staplers and so large sizes and other unusual configurations may have been implicitly precluded by the assumptions of those who took the photographs. That is, the photographer probably took pictures only of real, normal-sized staplers. People also tend to center objects when taking pictures and tend to put them in characteristic orientations. Thus, there is often quite a bit of unintentional implicit information within photos taken by people.

Contextual information can also be modeled explicitly with machine learning techniques. Hidden variables such as size, orientation to gravity, and so on can then be correlated with their values in a labeled training set. Alternatively, one may attempt to measure hidden bias variables by using additional sensors. The use of a laser range finder to measure depth allows us to accurately measure the size of an object.

The next problem facing computer vision is noise. We typically deal with noise by using statistical methods. For example, it may be impossible to detect an edge in an image merely by comparing a point to its immediate neighbors. But if we look at the statistics over a local region, edge detection becomes much easier. A real edge should appear as a string of such immediate neighbor responses over a local region, each of whose orientation is consistent with its neighbors. It is also possible to compensate for noise by taking statistics over time. Still other techniques account for noise or distortions by building explicit models learned directly from the available data. For example, because lens distortions are well understood, one need only learn the parameters for a simple polynomial model in order to describe—and thus correct almost completely—such distortions.

The actions or decisions that computer vision attempts to make based on camera data are performed in the context of a specific purpose or task. We may want to remove noise or damage from an image so that our security system will issue an alert if someone tries to climb a fence or because we need a monitoring system that counts how many people cross through an area in an amusement park. Vision software for robots that wander through office buildings will employ different strategies than vision software for stationary security cameras because the two systems have significantly different contexts and objectives. As a general rule: the more constrained a computer vision context is, the more we can rely on those constraints to simplify the problem and the more reliable our final solution will be.

OpenCV is aimed at providing the basic tools needed to solve computer vision problems. In some cases, high-level functionalities in the library will be sufficient to solve the more complex problems in computer vision. Even when this is not the case, the basic components in the library are complete enough to enable creation of a complete solution of your own to almost any computer vision problem. In the latter case, there are several tried-and-true methods of using the library; all of them start with solving the problem using as many available library components as possible. Typically, after you've developed this first-draft solution, you can see where the solution has weaknesses and then fix those weaknesses using your own code and cleverness (better known as "solve the problem you actually have, not the one you imagine"). You can then use your draft

solution as a benchmark to assess the improvements you have made. From that point, whatever weaknesses remain can be tackled by exploiting the context of the larger system in which your problem solution is embedded.

The Origin of OpenCV

OpenCV grew out of an Intel Research initiative to advance CPU-intensive applications. Toward this end, Intel launched many projects including real-time ray tracing and 3D display walls. One of the authors working for Intel at that time was visiting universities and noticed that some top university groups, such as the MIT Media Lab, had well-developed and internally open computer vision infrastructures—code that was passed from student to student and that gave each new student a valuable head start in developing his or her own vision application. Instead of reinventing the basic functions from scratch, a new student could begin by building on top of what came before.

Thus, OpenCV was conceived as a way to make computer vision infrastructure universally available. With the aid of Intel's Performance Library Team,* OpenCV started with a core of implemented code and algorithmic specifications being sent to members of Intel's Russian library team. This is the "where" of OpenCV: it started in Intel's research lab with collaboration from the Software Performance Libraries group together with implementation and optimization expertise in Russia.

Chief among the Russian team members was Vadim Pisarevsky, who managed, coded, and optimized much of OpenCV and who is still at the center of much of the OpenCV effort. Along with him, Victor Eruhimov helped develop the early infrastructure, and Valery Kuriakin managed the Russian lab and greatly supported the effort. There were several goals for OpenCV at the outset:

- 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 commercial applications to be open or free themselves.

Those goals constitute the "why" of OpenCV. Enabling computer vision applications would increase the need for fast processors. Driving upgrades to faster processors would generate more income for Intel than selling some extra software. Perhaps that is why this open and free code arose from a hardware vendor rather than a software company. In some sense, there is more room to be innovative at software within a hardware company.

In any open source effort, it's important to reach a critical mass at which the project becomes self-sustaining. There have now been approximately two million downloads

* Shinn Lee was of key help.

of OpenCV, and this number is growing by an average of 26,000 downloads a month. The user group now approaches 20,000 members. OpenCV receives many user contributions, and central development has largely moved outside of Intel.* OpenCV's past timeline is shown in Figure 1-3. Along the way, OpenCV was affected by the dot-com boom and bust and also by numerous changes of management and direction. During these fluctuations, there were times when OpenCV had no one at Intel working on it at all. However, with the advent of multicore processors and the many new applications of computer vision, OpenCV's value began to rise. Today, OpenCV is an active area of development at several institutions, so expect to see many updates in multicamera calibration, depth perception, methods for mixing vision with laser range finders, and better pattern recognition as well as a lot of support for robotic vision needs. For more information on the future of OpenCV, see Chapter 14.

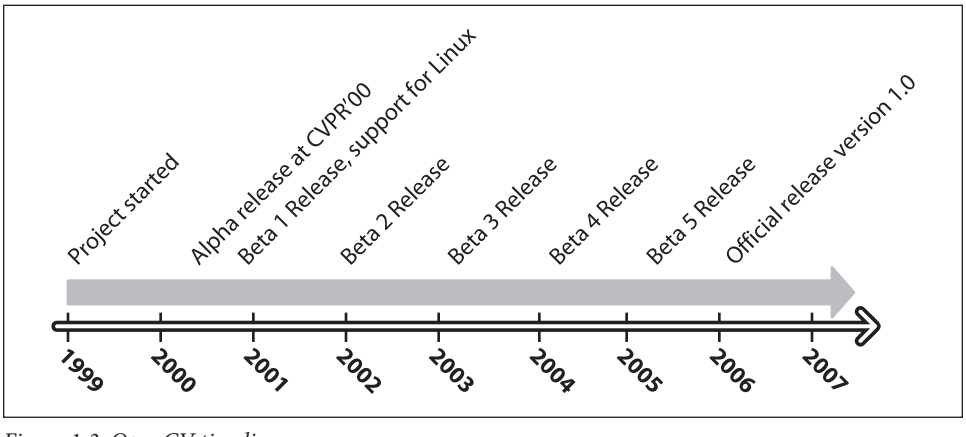


Figure 1-3. OpenCV timeline

Speeding Up OpenCV with IPP

Because OpenCV was "housed" within the Intel Performance Primitives team and several primary developers remain on friendly terms with that team, OpenCV exploits the hand-tuned, highly optimized code in IPP to speed itself up. The improvement in speed from using IPP can be substantial. Figure 1-4 compares two other vision libraries, LTI [LTI] and VXL [VXL], against OpenCV and OpenCV using IPP. Note that performance was a key goal of OpenCV; the library needed the ability to run vision code in real time.

OpenCV is written in performance-optimized C and C++ code. It does *not* depend in any way on IPP. If IPP is present, however, OpenCV will automatically take advantage of IPP by loading IPP's dynamic link libraries to further enhance its speed.

* As of this writing, Willow Garage [WG] (*www.willowgarage.com*), a robotics research institute and incubator, is actively supporting general OpenCV maintenance and new development in the area of robotics applications.

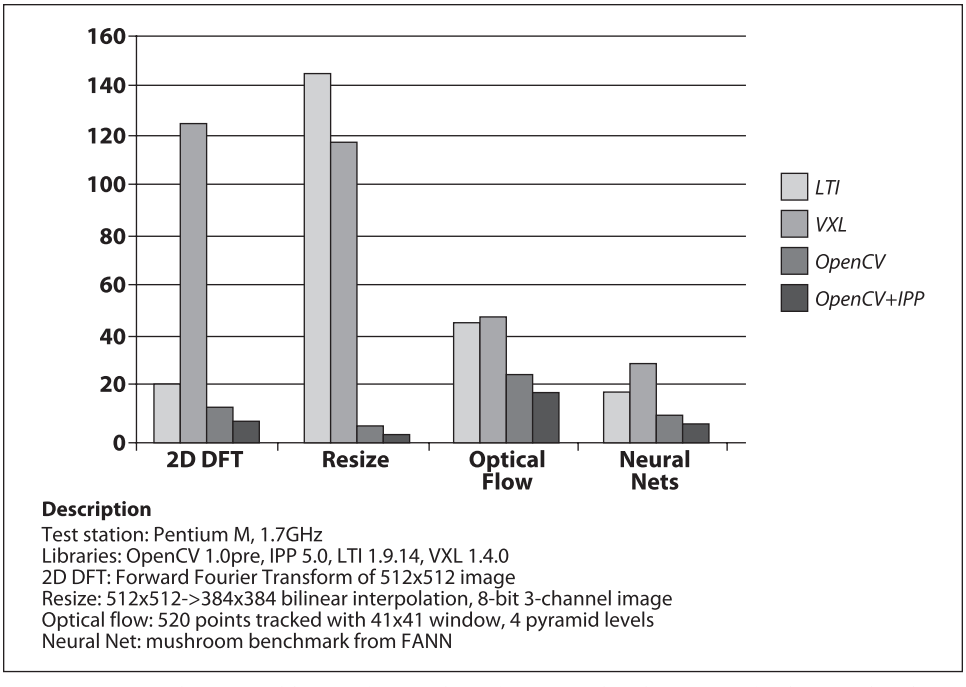


Figure 1-4. Two other vision libraries (LTI and VXL) compared with OpenCV (without and with IPP) on four different performance benchmarks: the four bars for each benchmark indicate scores proportional to run time for each of the given libraries; in all cases, OpenCV outperforms the other libraries and OpenCV with IPP outperforms OpenCV without IPP

Who Owns OpenCV?

Although Intel started OpenCV, the library is and always was intended to promote commercial and research use. It is therefore open and free, and the code itself may be used or embedded (in whole or in part) in other applications, whether commercial or research. It does not force your application code to be open or free. It does not require that you return improvements back to the library—but we hope that you will.

Downloading and Installing OpenCV

The main OpenCV site is on SourceForge at *http://SourceForge.net/projects/opencvlibrary* and the OpenCV Wiki [OpenCV Wiki] page is at *http://opencvlibrary.SourceForge.net*. For Linux, the source distribution is the file *opencv-1.0.0.tar.gz*; for Windows, you want *OpenCV_1.0.exe*. However, the most up-to-date version is always on the CVS server at SourceForge.

Install

Once you download the libraries, you must install them. For detailed installation instructions on Linux or Mac OS, see the text file named *INSTALL* directly under the

.../opencv/ directory; this file also describes how to build and run the OpenCV testing routines. *INSTALL* lists the additional programs you'll need in order to become an OpenCV developer, such as *autoconf*, *automake*, *libtool*, and *swig*.

Windows

Get the executable installation from SourceForge and run it. It will install OpenCV, register DirectShow filters, and perform various post-installation procedures. You are now ready to start using OpenCV. You can always go to the *.../opencv/_make* directory and open *opencv.sln* with MSVC++ or MSVC.NET 2005, or you can open *opencv.dsw* with lower versions of MSVC++ and build debug versions or rebuild release versions of the library.*

To add the commercial IPP performance optimizations to Windows, obtain and install IPP from the Intel site (*http://www.intel.com/software/products/ipp/index.htm*); use version 5.1 or later. Make sure the appropriate binary folder (e.g., *c:/program files/intel/ipp/5.1/ia32/bin*) is in the system path. IPP should now be automatically detected by OpenCV and loaded at runtime (more on this in Chapter 3).

Linux

Prebuilt binaries for Linux are not included with the Linux version of OpenCV owing to the large variety of versions of GCC and GLIBC in different distributions (SuSE, Debian, Ubuntu, etc.). If your distribution doesn't offer OpenCV, you'll have to build it from sources as detailed in the *.../opencv/INSTALL* file.

To build the libraries and demos, you'll need GTK+ 2.x or higher, including headers. You'll also need *pkgconfig*, *libpng*, *zlib*, *libjpeg*, *libtiff*, and *libjasper* with development files. You'll need Python 2.3, 2.4, or 2.5 with headers installed (developer package). You will also need *libavcodec* and the other *libav** libraries (including headers) from *ffmpeg* `0.4.9-pre1` or later (*svn checkout svn://svn.mplayerhq.hu/ffmpeg/trunk ffmpeg*).

Download ffmpeg from *http://ffmpeg.mplayerhq.hu/download.html*.† The ffmpeg program has a lesser general public license (LGPL). To use it with non-GPL software (such as OpenCV), build and use a shared ffmpg library:

```
$> ./configure --enable-shared
$> make
$> sudo make install
```

You will end up with: */usr/local/lib/libavcodec.so.**, */usr/local/lib/libavformat.so.**, */usr/local/lib/libavutil.so.**, and include files under various */usr/local/include/libav**.

To build OpenCV once it is downloaded:‡

* It is important to know that, although the Windows distribution contains binary libraries for release builds, it does not contain the debug builds of these libraries. It is therefore likely that, before developing with OpenCV, you will want to open the solution file and build these libraries for yourself.

† You can check out ffmpeg by: *svn checkout svn://svn.mplayerhq.hu/ffmpeg/trunk ffmpeg*.

‡ To build OpenCV using Red Hat Package Managers (RPMs), use **`rpmbuild -ta OpenCV-x.y.z.tar.gz`** (for RPM 4.x or later), or **`rpm -ta OpenCV-x.y.z.tar.gz`** (for earlier versions of RPM), where *OpenCV-x.y.z.tar.gz* should be put in */usr/src/redhat/SOURCES/* or a similar directory. Then install OpenCV using **`rpm -i OpenCV-x.y.z.*.rpm`**.

```
$> ./configure
$> make
$> sudo make install
$> sudo ldconfig
```

After installation is complete, the default installation path is */usr/local/lib/* and */usr/local/include/opencv/*. Hence you need to add */usr/local/lib/* to */etc/ld.so.conf* (and run `ldconfig` afterwards) *or* add it to the `LD_LIBRARY_PATH` environment variable; then you are done.

To add the commercial IPP performance optimizations to Linux, install IPP as described previously. Let's assume it was installed in */opt/intel/ipp/5.1/ia32/*. Add *<your install_path>/bin/* and *<your install_path>/bin/linux32* `LD_LIBRARY_PATH` in your initialization script (*.bashrc* or similar):

```
LD_LIBRARY_PATH=/opt/intel/ipp/5.1/ia32/bin:/opt/intel/ipp/5.1
/ia32/bin/linux32:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
```

Alternatively, you can add *<your install_path>/bin* and *<your install_path>/bin/linux32*, one per line, to */etc/ld.so.conf* and then run *ldconfig* as root (or use sudo).

That's it. Now OpenCV should be able to locate IPP shared libraries and make use of them on Linux. See *.../opencv/INSTALL* for more details.

MacOS X

As of this writing, full functionality on MacOS X is a priority but there are still some limitations (e.g., writing AVIs); these limitations are described in *.../opencv/INSTALL*.

The requirements and building instructions are similar to the Linux case, with the following exceptions:

- By default, Carbon is used instead of GTK+.
- By default, QuickTime is used instead of ffmpeg.
- pkg-config is optional (it is used explicitly only in the *samples/c/build_all.sh* script).
- RPM and ldconfig are not supported by default. Use `configure+make+sudo make install` to build and install OpenCV, update `LD_LIBRARY_PATH` (unless `./configure --prefix=/usr` is used).

For full functionality, you should install *libpng*, *libtiff*, *libjpeg* and *libjasper* from *darwinports* and/or *fink* and make them available to *./configure* (see `./configure --help`). For the most current information, see the OpenCV Wiki at *http://opencvlibrary.SourceForge.net/* and the Mac-specific page *http://opencvlibrary.SourceForge.net/Mac_OS_X_OpenCV_Port*.

Getting the Latest OpenCV via CVS

OpenCV is under active development, and bugs are often fixed rapidly when bug reports contain accurate descriptions and code that demonstrates the bug. However,

official OpenCV releases occur only once or twice a year. If you are seriously developing a project or product, you will probably want code fixes and updates as soon as they become available. To do this, you will need to access OpenCV's Concurrent Versions System (CVS) on SourceForge.

This isn't the place for a tutorial in CVS usage. If you've worked with other open source projects then you're probably familiar with it already. If you haven't, check out *Essential CVS* by Jennifer Vesperman (O'Reilly). A command-line CVS client ships with Linux, OS X, and most UNIX-like systems. For Windows users, we recommend TortoiseCVS (*http://www.tortoisecvs.org/*), which integrates nicely with Windows Explorer.

On Windows, if you want the latest OpenCV from the CVS repository then you'll need to access the CVSROOT directory:

```
:pserver:anonymous@opencvlibrary.cvs.sourceforge.net:2401/cvsroot/opencvlibrary
```

On Linux, you can just use the following two commands:

```
cvs -d:pserver:anonymous@opencvlibrary.cvs.sourceforge.net:/cvsroot/opencvlibrary
login
```

When asked for password, hit return. Then use:

```
cvs -z3 -d:pserver:anonymous@opencvlibrary.cvs.sourceforge.net:/cvsroot/opencvlibrary
co -P opencv
```

More OpenCV Documentation

The primary documentation for OpenCV is the HTML documentation that ships with the source code. In addition to this, the OpenCV Wiki and the older HTML documentation are available on the Web.

Documentation Available in HTML

OpenCV ships with html-based user documentation in the *.../opencv/docs* subdirectory. Load the *index.htm* file, which contains the following links.

CXCORE
: Contains data structures, matrix algebra, data transforms, object persistence, memory management, error handling, and dynamic loading of code as well as drawing, text and basic math.

CV
: Contains image processing, image structure analysis, motion and tracking, pattern recognition, and camera calibration.

Machine Learning (ML)
: Contains many clustering, classification and data analysis functions.

HighGUI
: Contains user interface GUI and image/video storage and recall.

CVCAM
: Camera interface.

Haartraining
: How to train the boosted cascade object detector. This is in the *.../opencv/apps/HaarTraining/doc/haartraining.htm* file.

The *.../opencv/docs* directory also contains *IPLMAN.pdf*, which was the original manual for OpenCV. It is now defunct and should be used with caution, but it does include detailed descriptions of algorithms and of what image types may be used with a particular algorithm. Of course, the first stop for such image and algorithm details is the book you are reading now.

Documentation via the Wiki

OpenCV's documentation Wiki is more up-to-date than the html pages that ship with OpenCV and it also features additional content as well. The Wiki is located at *http://opencvlibrary.SourceForge.net*. It includes information on:

- Instructions on compiling OpenCV using Eclipse IDE
- Face recognition with OpenCV
- Video surveillance library
- Tutorials
- Camera compatibility
- Links to the Chinese and the Korean user groups

Another Wiki, located at *http://opencvlibrary.SourceForge.net/CvAux*, is the only documentation of the auxiliary functions discussed in "OpenCV Structure and Content" (next section). CvAux includes the following functional areas:

- Stereo correspondence
- View point morphing of cameras
- 3D tracking in stereo
- Eigen object (PCA) functions for object recognition
- Embedded hidden Markov models (HMMs)

This Wiki has been translated into Chinese at *http://www.opencv.org.cn/index.php/%E9%A6%96%E9%A1%B5*.

Regardless of your documentation source, it is often hard to know:

- Which image type (floating, integer, byte; 1–3 channels) works with which function
- Which functions work in place
- Details of how to call the more complex functions (e.g., contours)

- Details about running many of the examples in the *…/opencv/samples/c/* directory
- *What* to do, not just how
- How to set parameters of certain functions

One aim of this book is to address these problems.

OpenCV Structure and Content

OpenCV is broadly structured into five main components, four of which are shown in Figure 1-5. The CV component contains the basic image processing and higher-level computer vision algorithms; ML is the machine learning library, which includes many statistical classifiers and clustering tools. HighGUI contains I/O routines and functions for storing and loading video and images, and CXCore contains the basic data structures and content.

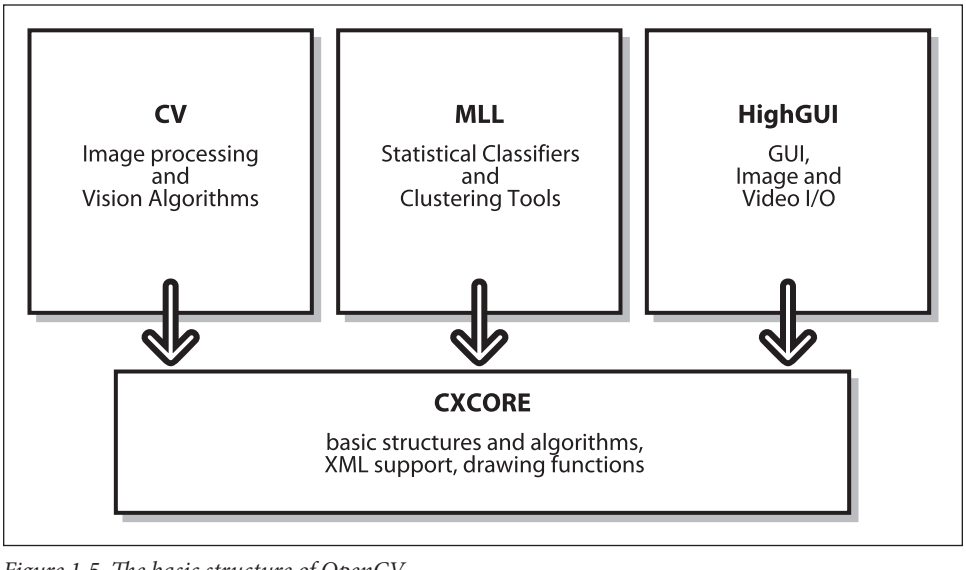


Figure 1-5. The basic structure of OpenCV

Figure 1-5 does not include CvAux, which contains both defunct areas (embedded HMM face recognition) and experimental algorithms (background/foreground segmentation). CvAux is not particularly well documented in the Wiki and is not documented at all in the *.../opencv/docs* subdirectory. CvAux covers:

- Eigen objects, a computationally efficient recognition technique that is, in essence, a template matching procedure
- 1D and 2D hidden Markov models, a statistical recognition technique solved by dynamic programming
- Embedded HMMs (the observations of a parent HMM are themselves HMMs)

- Gesture recognition from stereo vision support
- Extensions to Delaunay triangulation, sequences, and so forth
- Stereo vision
- Shape matching with region contours
- Texture descriptors
- Eye and mouth tracking
- 3D tracking
- Finding skeletons (central lines) of objects in a scene
- Warping intermediate views between two camera views
- Background-foreground segmentation
- Video surveillance (see Wiki FAQ for more documentation)
- Camera calibration C++ classes (the C functions and engine are in CV)

Some of these features may migrate to CV in the future; others probably never will.

Portability

OpenCV was designed to be portable. It was originally written to compile across Borland C++, MSVC++, and the Intel compilers. This meant that the C and C++ code had to be fairly standard in order to make cross-platform support easier. Figure 1-6 shows the platforms on which OpenCV is known to run. Support for 32-bit Intel architecture (IA32) on Windows is the most mature, followed by Linux on the same architecture. Mac OS X portability became a priority only after Apple started using Intel processors. (The OS X port isn't as mature as the Windows or Linux versions, but this is changing rapidly.) These are followed by 64-bit support on extended memory (EM64T) and the 64-bit Intel architecture (IA64). The least mature portability is on Sun hardware and other operating systems.

If an architecture or OS doesn't appear in Figure 1-6, this doesn't mean there are no OpenCV ports to it. OpenCV has been ported to almost every commercial system, from PowerPC Macs to robotic dogs. OpenCV runs well on AMD's line of processors, and even the further optimizations available in IPP will take advantage of multimedia extensions (MMX) in AMD processors that incorporate this technology.

	IA32	EM64T	IA64	Other (PPC, Sparc)
Windows	✓ (w. IPP; MSVC6, .NET2005+OMP, ICC, GCC, BCC)	✓ (w. IPP; MSVC6+PSDK.NET2005+OMP, PSDK)	± (w. IPP; PSDK, some tests fail)	N/A
Linux	✓ (w. IPP; GCC, BCC)	✓ (w. IPP; GCC, BCC)	✓ (GCC, ICC)	✗
MacOSX	✓ (w. IPP, GCC, native APIs)	? (not tested)	N/A	✓ (iMac G5, GCC, native APIs)
Others (BSD, Solaris…)	✗	✗	✗	Reported to build on UltraSparc Solaris

Figure 1-6. OpenCV portability guide for release 1.0: operating systems are shown on the left; computer architecture types across top

Exercises

1. Download and install the latest release of OpenCV. Compile it in debug and release mode.
2. Download and build the latest CVS update of OpenCV.
3. Describe at least three ambiguous aspects of converting 3D inputs into a 2D representation. How would you overcome these ambiguities?

CHAPTER 2

Introduction to OpenCV

Getting Started

After installing the OpenCV library, our first task is, naturally, to get started and make something interesting happen. In order to do this, we will need to set up the programming environment.

In Visual Studio, it is necessary to create a project and to configure the setup so that (a) the libraries *highgui.lib*, *cxcore.lib*, *ml.lib*, and *cv.lib* are linked* and (b) the preprocessor will search the OpenCV *…/opencv/*/include* directories for header files. These "include" directories will typically be named something like *C:/program files/opencv/cv/include*,† *…/opencv/cxcore/include*, *…/opencv/ml/include*, and *…/opencv/otherlibs/highgui*. Once you've done this, you can create a new C file and start your first program.

Certain key header files can make your life much easier. Many useful macros are in the header files *…/opencv/cxcore/include/cxtypes.h* and *cxmisc.h*. These can do things like initialize structures and arrays in one line, sort lists, and so on. The most important headers for compiling are *.../cv/include/cv.h* and *…/cxcore/include/cxcore.h* for computer vision, *…/otherlibs/highgui/highgui.h* for I/O, and *…/ml/include/ml.h* for machine learning.

First Program—Display a Picture

OpenCV provides utilities for reading from a wide array of image file types as well as from video and cameras. These utilities are part of a toolkit called HighGUI, which is included in the OpenCV package. We will use some of these utilities to create a simple program that opens an image and displays it on the screen. See Example 2-1.

* For debug builds, you should link to the libraries *highguid.lib*, *cxcored.lib*, *mld.lib*, and *cvd.lib*.

† *C:/program files/* is the default installation of the OpenCV directory on Windows, although you can choose to install it elsewhere. To avoid confusion, from here on we'll use "*…/opencv/*" to mean the path to the opencv directory on your system.

Example 2-1. A simple OpenCV program that loads an image from disk and displays it on the screen

```
#include "highgui.h"

int main( int argc, char** argv ) {
    IplImage* img = cvLoadImage( argv[1] );
    cvNamedWindow( "Example1", CV_WINDOW_AUTOSIZE );
    cvShowImage( "Example1", img );
    cvWaitKey(0);
    cvReleaseImage( &img );
    cvDestroyWindow( "Example1" );
}
```

When compiled and run from the command line with a single argument, this program loads an image into memory and displays it on the screen. It then waits until the user presses a key, at which time it closes the window and exits. Let's go through the program line by line and take a moment to understand what each command is doing.

```
IplImage* img = cvLoadImage( argv[1] );
```

This line loads the image.* The function `cvLoadImage()` is a high-level routine that determines the file format to be loaded based on the file name; it also automatically allocates the memory needed for the image data structure. Note that `cvLoadImage()` can read a wide variety of image formats, including BMP, DIB, JPEG, JPE, PNG, PBM, PGM, PPM, SR, RAS, and TIFF. A pointer to an allocated image data structure is then returned. This structure, called `IplImage`, is the OpenCV construct with which you will deal the most. OpenCV uses this structure to handle all kinds of images: single-channel, multichannel, integer-valued, floating-point-valued, et cetera. We use the pointer that `cvLoadImage()` returns to manipulate the image and the image data.

```
cvNamedWindow( "Example1", CV_WINDOW_AUTOSIZE );
```

Another high-level function, `cvNamedWindow()`, opens a window on the screen that can contain and display an image. This function, provided by the HighGUI library, also assigns a name to the window (in this case, "`Example1`"). Future HighGUI calls that interact with this window will refer to it by this name.

The second argument to `cvNamedWindow()` defines window properties. It may be set either to 0 (the default value) or to `CV_WINDOW_AUTOSIZE`. In the former case, the size of the window will be the same regardless of the image size, and the image will be scaled to fit within the window. In the latter case, the window will expand or contract automatically when an image is loaded so as to accommodate the image's true size.

```
cvShowImage( "Example1", img );
```

Whenever we have an image in the form of an `IplImage*` pointer, we can display it in an existing window with `cvShowImage()`. The `cvShowImage()` function requires that a named window already exist (created by `cvNamedWindow()`). On the call to `cvShowImage()`, the

* A proper program would check for the existence of `argv[1]` and, in its absence, deliver an instructional error message for the user. We will abbreviate such necessities in this book and assume that the reader is cultured enough to understand the importance of error-handling code.

window will be redrawn with the appropriate image in it, and the window will resize itself as appropriate if it was created using the CV_WINDOW_AUTOSIZE flag.

```
cvWaitKey(0);
```

The cvWaitKey() function asks the program to stop and wait for a keystroke. If a positive argument is given, the program will wait for that number of milliseconds and then continue even if nothing is pressed. If the argument is set to 0 or to a negative number, the program will wait indefinitely for a keypress.

```
cvReleaseImage( &img );
```

Once we are through with an image, we can free the allocated memory. OpenCV expects a pointer to the IplImage* pointer for this operation. After the call is completed, the pointer img will be set to NULL.

```
cvDestroyWindow( "Example1" );
```

Finally, we can destroy the window itself. The function cvDestroyWindow() will close the window and de-allocate any associated memory usage (including the window's internal image buffer, which is holding a copy of the pixel information from *img). For a simple program, you don't really have to call cvDestroyWindow() or cvReleaseImage() because all the resources and windows of the application are closed automatically by the operating system upon exit, but it's a good habit anyway.

Now that we have this simple program we can toy around with it in various ways, but we don't want to get ahead of ourselves. Our next task will be to construct a very simple—almost as simple as this one—program to read in and display an AVI video file. After that, we will start to tinker a little more.

Second Program—AVI Video

Playing a video with OpenCV is almost as easy as displaying a single picture. The only new issue we face is that we need some kind of loop to read each frame in sequence; we may also need some way to get out of that loop if the movie is too boring. See Example 2-2.

Example 2-2. A simple OpenCV program for playing a video file from disk

```
#include "highgui.h"

int main( int argc, char** argv ) {
    cvNamedWindow( "Example2", CV_WINDOW_AUTOSIZE );
    CvCapture* capture = cvCreateFileCapture( argv[1] );
    IplImage* frame;
    while(1) {
        frame = cvQueryFrame( capture );
        if( !frame ) break;
        cvShowImage( "Example2", frame );
        char c = cvWaitKey(33);
        if( c == 27 ) break;
    }
    cvReleaseCapture( &capture );
    cvDestroyWindow( "Example2" );
}
```

Here we begin the function `main()` with the usual creation of a named window, in this case "`Example2`". Things get a little more interesting after that.

```
CvCapture* capture = cvCreateFileCapture( argv[1] );
```

The function `cvCreateFileCapture()` takes as its argument the name of the AVI file to be loaded and then returns a pointer to a `CvCapture` structure. This structure contains all of the information about the AVI file being read, including state information. When created in this way, the `CvCapture` structure is initialized to the beginning of the AVI.

```
frame = cvQueryFrame( capture );
```

Once inside of the `while(1)` loop, we begin reading from the AVI file. `cvQueryFrame()` takes as its argument a pointer to a `CvCapture` structure. It then grabs the next video frame into memory (memory that is actually part of the `CvCapture` structure). A pointer is returned to that frame. Unlike `cvLoadImage`, which actually allocates memory for the image, `cvQueryFrame` uses memory already allocated in the `CvCapture` structure. Thus it will not be necessary (or wise) to call `cvReleaseImage()` for this "frame" pointer. Instead, the frame image memory will be freed when the `CvCapture` structure is released.

```
c = cvWaitKey(33);
if( c == 27 ) break;
```

Once we have displayed the frame, we then wait for 33 ms.* If the user hits a key, then `c` will be set to the ASCII value of that key; if not, then it will be set to –1. If the user hits the Esc key (ASCII 27), then we will exit the read loop. Otherwise, 33 ms will pass and we will just execute the loop again.

It is worth noting that, in this simple example, we are not explicitly controlling the speed of the video in any intelligent way. We are relying solely on the timer in `cvWaitKey()` to pace the loading of frames. In a more sophisticated application it would be wise to read the actual frame rate from the `CvCapture` structure (from the AVI) and behave accordingly!

```
cvReleaseCapture( &capture );
```

When we have exited the read loop—because there was no more video data or because the user hit the Esc key—we can free the memory associated with the `CvCapture` structure. This will also close any open file handles to the AVI file.

Moving Around

OK, that was great. Now it's time to tinker around, enhance our toy programs, and explore a little more of the available functionality. The first thing we might notice about the AVI player of Example 2-2 is that it has no way to move around quickly within the video. Our next task will be to add a slider bar, which will give us this ability.

* You can wait any amount of time you like. In this case, we are simply assuming that it is correct to play the video at 30 frames per second and allow user input to interrupt between each frame (thus we pause for input 33 ms between each frame). In practice, it is better to check the `CvCapture` structure returned by `cvCaptureFromCamera()` in order to determine the actual frame rate (more on this in Chapter 4).

The HighGUI toolkit provides a number of simple instruments for working with images and video beyond the simple display functions we have just demonstrated. One especially useful mechanism is the slider, which enables us to jump easily from one part of a video to another. To create a slider, we call `cvCreateTrackbar()` and indicate which window we would like the trackbar to appear in. In order to obtain the desired functionality, we need only supply a callback that will perform the relocation. Example 2-3 gives the details.

Example 2-3. Program to add a trackbar slider to the basic viewer window: when the slider is moved, the function onTrackbarSlide() is called and then passed to the slider's new value

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

int        g_slider_position = 0;
CvCapture* g_capture         = NULL;

void onTrackbarSlide(int pos) {
    cvSetCaptureProperty(
        g_capture,
        CV_CAP_PROP_POS_FRAMES,
        pos
    );
}

int main( int argc, char** argv ) {
    cvNamedWindow( "Example3", CV_WINDOW_AUTOSIZE );
    g_capture = cvCreateFileCapture( argv[1] );
    int frames = (int) cvGetCaptureProperty(
        g_capture,
        CV_CAP_PROP_FRAME_COUNT
    );
    if( frames!= 0 ) {
      cvCreateTrackbar(
          "Position",
          "Example3",
          &g_slider_position,
          frames,
          onTrackbarSlide
      );
    }
    IplImage* frame;
    // While loop (as in Example 2) capture & show video
    …
    // Release memory and destroy window
    …
    return(0);
}
```

In essence, then, the strategy is to add a global variable to represent the slider position and then add a callback that updates this variable and relocates the read position in the

video. One call creates the slider and attaches the callback, and we are off and running.* Let's look at the details.

```
int g_slider_position = 0;
CvCapture* g_capture  = NULL;
```

First we define a global variable for the slider position. The callback will need access to the capture object, so we promote that to a global variable. Because we are nice people and like our code to be readable and easy to understand, we adopt the convention of adding a leading g_ to any global variable.

```
void onTrackbarSlide(int pos) {
    cvSetCaptureProperty(
    g_capture,
    CV_CAP_PROP_POS_FRAMES,
    pos
 );
```

Now we define a callback routine to be used when the user pokes the slider. This routine will be passed to a 32-bit integer, which will be the slider position.

The call to `cvSetCaptureProperty()` is one we will see often in the future, along with its counterpart `cvGetCaptureProperty()`. These routines allow us to configure (or query in the latter case) various properties of the `CvCapture` object. In this case we pass the argument `CV_CAP_PROP_POS_FRAMES`, which indicates that we would like to set the read position in units of frames. (We can use `AVI_RATIO` instead of `FRAMES` if we want to set the position as a fraction of the overall video length). Finally, we pass in the new value of the position. Because HighGUI is highly civilized, it will automatically handle such issues as the possibility that the frame we have requested is not a key-frame; it will start at the previous key-frame and fast forward up to the requested frame without us having to fuss with such details.

```
int frames = (int) cvGetCaptureProperty(
    g_capture,
    CV_CAP_PROP_FRAME_COUNT
);
```

As promised, we use `cvGetCaptureProperty()`when we want to query some data from the `CvCapture` structure. In this case, we want to find out how many frames are in the video so that we can calibrate the slider (in the next step).

```
if( frames!= 0 ) {
  cvCreateTrackbar(
      "Position",
      "Example3",
      &g_slider_position,
      frames,
      onTrackbarSlide
  );
}
```

* This code does not update the slider position as the video plays; we leave that as an exercise for the reader. Also note that some mpeg encodings do not allow you to move backward in the video.

The last detail is to create the trackbar itself. The function `cvCreateTrackbar()` allows us to give the trackbar a label* (in this case `Position`) and to specify a window to put the trackbar in. We then provide a variable that will be bound to the trackbar, the maximum value of the trackbar, and a callback (or NULL if we don't want one) for when the slider is moved. Observe that we do not create the trackbar if `cvGetCaptureProperty()` returned a zero frame count. This is because sometimes, depending on how the video was encoded, the total number of frames will not be available. In this case we will just play the movie without providing a trackbar.

It is worth noting that the slider created by HighGUI is not as full-featured as some sliders out there. Of course, there's no reason you can't use your favorite windowing toolkit instead of HighGUI, but the HighGUI tools are quick to implement and get us off the ground in a hurry.

Finally, we did not include the extra tidbit of code needed to make the slider move as the video plays. This is left as an exercise for the reader.

A Simple Transformation

Great, so now you can use OpenCV to create your own video player, which will not be much different from countless video players out there already. But we are interested in computer vision, and we want to do some of that. Many basic vision tasks involve the application of filters to a video stream. We will modify the program we already have to do a simple operation on every frame of the video as it plays.

One particularly simple operation is the smoothing of an image, which effectively reduces the information content of the image by convolving it with a Gaussian or other similar kernel function. OpenCV makes such convolutions exceptionally easy to do. We can start by creating a new window called "`Example4-out`", where we can display the results of the processing. Then, after we have called `cvShowImage()` to display the newly captured frame in the input window, we can compute and display the smoothed image in the output window. See Example 2-4.

Example 2-4. Loading and then smoothing an image before it is displayed on the screen

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

void example2_4( IplImage* image )

    // Create some windows to show the input
    // and output images in.
    //
    cvNamedWindow( "Example4-in" );
```

* Because HighGUI is a lightweight and easy-to-use toolkit, `cvCreateTrackbar()` does not distinguish between the name of the trackbar and the label that actually appears on the screen next to the trackbar. You may already have noticed that `cvNamedWindow()` likewise does not distinguish between the name of the window and the label that appears on the window in the GUI.

Example 2-4. Loading and then smoothing an image before it is displayed on the screen (continued)

```
    cvNamedWindow( "Example4-out" );

    // Create a window to show our input image
    //
    cvShowImage( "Example4-in", image );

    // Create an image to hold the smoothed output
    //
    IplImage* out = cvCreateImage(
        cvGetSize(image),
        IPL_DEPTH_8U,
        3
    );

    // Do the smoothing
    //
    cvSmooth( image, out, CV_GAUSSIAN, 3, 3 );

    // Show the smoothed image in the output window
    //
    cvShowImage( "Example4-out", out );

    // Be tidy
    //
    cvReleaseImage( &out );

    // Wait for the user to hit a key, then clean up the windows
    //
    cvWaitKey( 0 );
    cvDestroyWindow( "Example4-in" );
    cvDestroyWindow( "Example4-out" );

}
```

The first call to `cvShowImage()` is no different than in our previous example. In the next call, we allocate another image structure. Previously we relied on `cvCreateFileCapture()` to allocate the new frame for us. In fact, that routine actually allocated only one frame and then wrote over that data each time a capture call was made (so it actually returned the same pointer every time we called it). In this case, however, we want to allocate our own image structure to which we can write our smoothed image. The first argument is a `CvSize` structure, which we can conveniently create by calling `cvGetSize(image)`; this gives us the size of the existing structure `image`. The second argument tells us what kind of data type is used for each channel on each pixel, and the last argument indicates the number of channels. So this image is three channels (with 8 bits per channel) and is the same size as `image`.

The smoothing operation is itself just a single call to the OpenCV library: we specify the input image, the output image, the smoothing method, and the parameters for the smooth. In this case we are requesting a Gaussian smooth over a 3×3 area centered on each pixel. It is actually allowed for the output to be the same as the input image, and

this would work more efficiently in our current application, but we avoided doing this because it gave us a chance to introduce cvCreateImage()!

Now we can show the image in our new second window and then free it: cvReleaseImage() takes a pointer to the IplImage* pointer and then de-allocates all of the memory associated with that image.

A Not-So-Simple Transformation

That was pretty good, and we are learning to do more interesting things. In Example 2-4 we chose to allocate a new IplImage structure, and into this new structure we wrote the output of a single transformation. As mentioned, we could have applied the transformation in such a way that the output overwrites the original, but this is not always a good idea. In particular, some operators do not produce images with the same size, depth, and number of channels as the input image. Typically, we want to perform a *sequence* of operations on some initial image and so produce a chain of transformed images.

In such cases, it is often useful to introduce simple wrapper functions that both allocate the output image and perform the transformation we are interested in. Consider, for example, the reduction of an image by a factor of 2 [Rosenfeld80]. In OpenCV this is accomplished by the function cvPyrDown(), which performs a Gaussian smooth and then removes every other line from an image. This is useful in a wide variety of important vision algorithms. We can implement the simple function described in Example 2-5.

Example 2-5. Using cvPyrDown() to create a new image that is half the width and height of the input image

```
IplImage* doPyrDown(
  IplImage* in,
  int       filter = IPL_GAUSSIAN_5x5
) {

    // Best to make sure input image is divisible by two.
    //
    assert( in->width%2 == 0 && in->height%2 == 0 );

    IplImage* out = cvCreateImage(
        cvSize( in->width/2, in->height/2 ),
        in->depth,
        in->nChannels
    );
    cvPyrDown( in, out );
    return( out );
};
```

Notice that we allocate the new image by reading the needed parameters from the old image. In OpenCV, all of the important data types are implemented as structures and passed around as structure pointers. There is no such thing as private data in OpenCV!

Let's now look at a similar but slightly more involved example involving the *Canny edge detector* [Canny86] (see Example 2-6). In this case, the edge detector generates an image that is the full size of the input image but needs only a single channel image to write to.

Example 2-6. The Canny edge detector writes its output to a single channel (grayscale) image

```
IplImage* doCanny(
    IplImage* in,
    double    lowThresh,
    double    highThresh,
    double    aperture
) {
    If(in->nChannels != 1)
        return(0); //Canny only handles gray scale images

    IplImage* out = cvCreateImage(
        cvSize( cvGetSize( in ),
        IPL_DEPTH_8U,
        1
    );
    cvCanny( in, out, lowThresh, highThresh, aperture );
    return( out );
};
```

This allows us to string together various operators quite easily. For example, if we wanted to shrink the image twice and then look for lines that were present in the twice-reduced image, we could proceed as in Example 2-7.

Example 2-7. Combining the pyramid down operator (twice) and the Canny subroutine in a simple image pipeline

```
IplImage* img1 = doPyrDown( in, IPL_GAUSSIAN_5x5 );
IplImage* img2 = doPyrDown( img1, IPL_GAUSSIAN_5x5 );
IplImage* img3 = doCanny( img2, 10, 100, 3 );

// do whatever with 'img3'
//
…
cvReleaseImage( &img1 );
cvReleaseImage( &img2 );
cvReleaseImage( &img3 );
```

It is important to observe that nesting the calls to various stages of our filtering pipeline is not a good idea, because then we would have no way to free the images that we are allocating along the way. If we are too lazy to do this cleanup, we could opt to include the following line in each of the wrappers:

```
    cvReleaseImage( &in );
```

This "self-cleaning" mechanism would be very tidy, but it would have the following disadvantage: if we actually did want to do something with one of the intermediate images, we would have no access to it. In order to solve that problem, the preceding code could be simplified as described in Example 2-8.

Example 2-8. Simplifying the image pipeline of Example 2-7 by making the individual stages release their intermediate memory allocations

```
IplImage* out;
out = doPyrDown( in, IPL_GAUSSIAN_5x5 );
out = doPyrDown( out, IPL_GAUSSIAN_5x5 );
out = doCanny( out, 10, 100, 3 );

// do whatever with 'out'
//
…
cvReleaseImage ( &out );
```

One final word of warning on the self-cleaning filter pipeline: in OpenCV we must always be certain that an image (or other structure) being de-allocated is one that was, in fact, explicitly allocated previously. Consider the case of the `IplImage*` pointer returned by `cvCreateFileCapture()`. Here the pointer points to a structure allocated as part of the `CvCapture` structure, and the target structure is allocated only once when the `CvCapture` is initialized and an AVI is loaded. De-allocating this structure with a call to `cvReleaseImage()` would result in some nasty surprises. The moral of this story is that, although it's important to take care of garbage collection in OpenCV, we should only clean up the garbage that we have created.

Input from a Camera

Vision can mean many things in the world of computers. In some cases we are analyzing still frames loaded from elsewhere. In other cases we are analyzing video that is being read from disk. In still other cases, we want to work with real-time data streaming in from some kind of camera device.

OpenCV—more specifically, the HighGUI portion of the OpenCV library—provides us with an easy way to handle this situation. The method is analogous to how we read AVIs. Instead of calling `cvCreateFileCapture()`, we call `cvCreateCameraCapture()`. The latter routine does not take a file name but rather a camera ID number as its argument. Of course, this is important only when multiple cameras are available. The default value is –1, which means "just pick one"; naturally, this works quite well when there is only one camera to pick (see Chapter 4 for more details).

The `cvCreateCameraCapture()` function returns the same `CvCapture*` pointer, which we can hereafter use exactly as we did with the frames grabbed from a video stream. Of course, a lot of work is going on behind the scenes to make a sequence of camera images look like a video, but we are insulated from all of that. We can simply grab images from the camera whenever we are ready for them and proceed as if we did not know the difference. For development reasons, most applications that are intended to operate in real time will have a video-in mode as well, and the universality of the `CvCapture` structure makes this particularly easy to implement. See Example 2-9.

Example 2-9. After the capture structure is initialized, it no longer matters whether the image is from a camera or a file

```
CvCapture* capture;

if( argc==1 ) {
    capture = cvCreateCameraCapture(0);
} else {
    capture = cvCreateFileCapture( argv[1] );
}
assert( capture != NULL );

// Rest of program proceeds totally ignorant
…
```

As you can see, this arrangement is quite ideal.

Writing to an AVI File

In many applications we will want to record streaming input or even disparate captured images to an output video stream, and OpenCV provides a straightforward method for doing this. Just as we are able to create a capture device that allows us to grab frames one at a time from a video stream, we are able to create a writer device that allows us to place frames one by one into a video file. The routine that allows us to do this is `cvCreateVideoWriter()`.

Once this call has been made, we may successively call `cvWriteFrame()`, once for each frame, and finally `cvReleaseVideoWriter()` when we are done. Example 2-10 describes a simple program that opens a video file, reads the contents, converts them to a log-polar format (something like what your eye actually sees, as described in Chapter 6), and writes out the log-polar image to a new video file.

Example 2-10. A complete program to read in a color video and write out the same video in grayscale

```
// Convert a video to grayscale
 // argv[1]: input video file
 // argv[2]: name of new output file
 //
#include “cv.h”
#include “highgui.h”
main( int argc, char* argv[] ) {
    CvCapture* capture = 0;
    capture = cvCreateFileCapture( argv[1] );
    if(!capture){
        return -1;
    }
    IplImage *bgr_frame=cvQueryFrame(capture);//Init the video read
    double fps = cvGetCaptureProperty (
        capture,
        CV_CAP_PROP_FPS
    );
```

Example 2-10. A complete program to read in a color video and write out the same video in grayscale (continued)

```
    CvSize size = cvSize(
       (int)cvGetCaptureProperty( capture, CV_CAP_PROP_FRAME_WIDTH),
       (int)cvGetCaptureProperty( capture, CV_CAP_PROP_FRAME_HEIGHT)
    );
    CvVideoWriter *writer = cvCreateVideoWriter(
        argv[2],
        CV_FOURCC('M','J','P','G'),
        fps,
        size
    );
    IplImage* logpolar_frame = cvCreateImage(
        size,
        IPL_DEPTH_8U,
        3
    );
    while( (bgr_frame=cvQueryFrame(capture)) != NULL ) {
        cvLogPolar( bgr_frame, logpolar_frame,
                    cvPoint2D32f(bgr_frame->width/2,
                    bgr_frame->height/2),
                    40,
                    CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS );
        cvWriteFrame( writer, logpolar_frame );
    }
    cvReleaseVideoWriter( &writer );
    cvReleaseImage( &logpolar_frame );
    cvReleaseCapture( &capture );
    return(0);
}
```

Looking over this program reveals mostly familiar elements. We open one video; start reading with `cvQueryFrame()`, which is necessary to read the video properties on some systems; and then use `cvGetCaptureProperty()` to ascertain various important properties of the video stream. We then open a video file for writing, convert the frame to logpolar format, and write the frames to this new file one at a time until there are none left. Then we close up.

The call to `cvCreateVideoWriter()` contains several parameters that we should understand. The first is just the filename for the new file. The second is the *video codec* with which the video stream will be compressed. There are countless such codecs in circulation, but whichever codec you choose must be available on your machine (codecs are installed separately from OpenCV). In our case we choose the relatively popular *MJPG* codec; this is indicated to OpenCV by using the macro `CV_FOURCC()`, which takes four characters as arguments. These characters constitute the "four-character code" of the codec, and every codec has such a code. The four-character code for *motion jpeg* is MJPG, so we specify that as `CV_FOURCC('M','J','P','G')`.

The next two arguments are the replay frame rate, and the size of the images we will be using. In our case, we set these to the values we got from the original (color) video.

Onward

Before moving on to the next chapter, we should take a moment to take stock of where we are and look ahead to what is coming. We have seen that the OpenCV API provides us with a variety of easy-to-use tools for loading still images from files, reading video from disk, or capturing video from cameras. We have also seen that the library contains primitive functions for manipulating these images. What we have not yet seen are the powerful elements of the library, which allow for more sophisticated manipulation of the entire set of abstract data types that are important to practical vision problem solving.

In the next few chapters we will delve more deeply into the basics and come to understand in greater detail both the interface-related functions and the image data types. We will investigate the primitive image manipulation operators and, later, some much more advanced ones. Thereafter, we will be ready to explore the many specialized services that the API provides for tasks as diverse as camera calibration, tracking, and recognition. Ready? Let's go!

Exercises

Download and install OpenCV if you have not already done so. Systematically go through the directory structure. Note in particular the *docs* directory; there you can load *index.htm*, which further links to the main documentation of the library. Further explore the main areas of the library. *Cvcore* contains the basic data structures and algorithms, *cv* contains the image processing and vision algorithms, *ml* includes algorithms for machine learning and clustering, and *otherlibs/highgui* contains the I/O functions. Check out the *_make* directory (containing the OpenCV build files) and also the samples directory, where example code is stored.

1. Go to the *…/opencv/_make* directory. On Windows, open the solution file *opencv.sln*; on Linux, open the appropriate makefile. Build the library in both the debug and the release versions. This may take some time, but you will need the resulting library and *dll* files.
2. Go to the *…/opencv/samples/c/* directory. Create a project or make file and then import and build *lkdemo.c* (this is an example motion tracking program). Attach a camera to your system and run the code. With the display window selected, type "r" to initialize tracking. You can add points by clicking on video positions with the mouse. You can also switch to watching only the points (and not the image) by typing "n". Typing "n" again will toggle between "night" and "day" views.
3. Use the capture and store code in Example 2-10, together with the `doPyrDown()` code of Example 2-5 to create a program that reads from a camera and stores downsampled color images to disk.

4. Modify the code in exercise 3 and combine it with the window display code in Example 2-1 to display the frames as they are processed.
5. Modify the program of exercise 4 with a slider control from Example 2-3 so that the user can dynamically vary the pyramid downsampling reduction level by factors of between 2 and 8. You may skip writing this to disk, but you should display the results.